

The Web's Security Model

Philippe De Ryck
philippe.deryck@cs.kuleuven.be

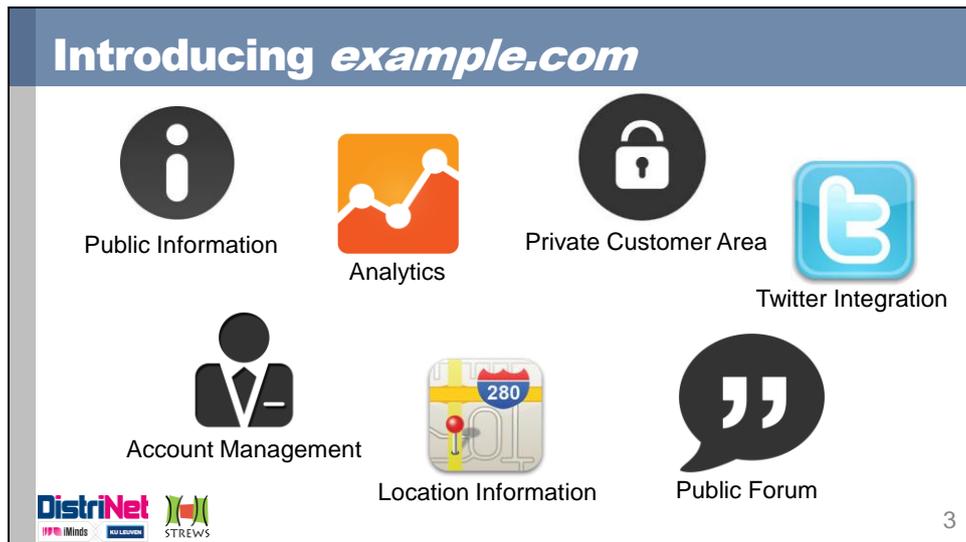


Who am I?

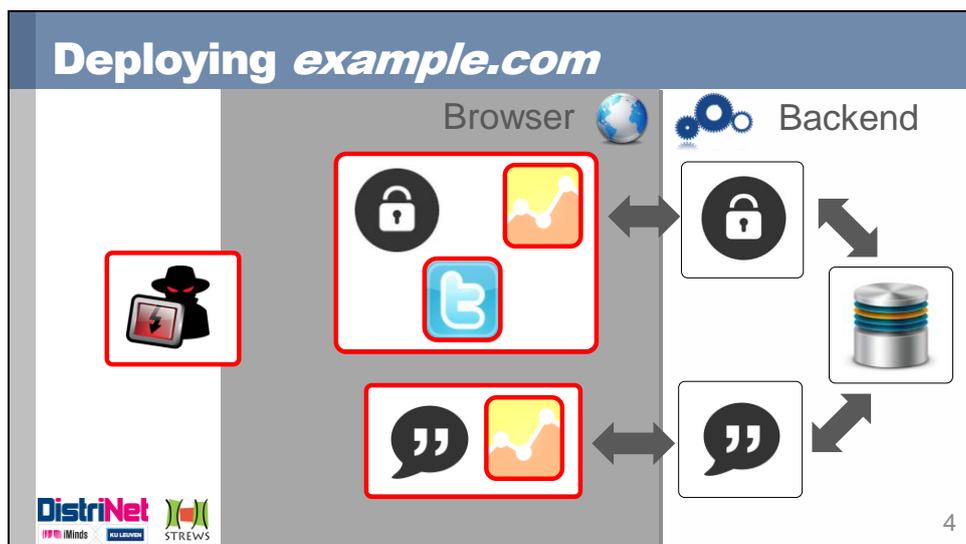


- Philippe De Ryck
 - Final year PhD student at iMinds-Distrinet, KU Leuven
 - Research on Cross-Site Request Forgery and Session Management
 - Author of CsFire, a browser add-on protecting against CSRF
- Contributor to the STREWS project
 - Roadmapping activity for Web Security
 - Main author of the **Web-platform security guide: Security Assessment of the Web Ecosystem**





Modern web applications are often composed of several distinct features, all integrated into one visual entity. This includes the incorporation of third-party code, such as social integration (like buttons, trending gadgets, etc.), analytics or other useful gadgets.



When web applications are deployed, they have a server-side backend, which is responsible for sending content to the client, and processing incoming requests from the client. At the client-side, the browser hosts several contexts (e.g. windows, tabs, ...) which run client-side code, such as *example.com*'s forum page, or a completely different page, with potentially malicious content. One might expect that these contexts are isolated in the browser, and that third-party code within a context is subject to certain constraints. Unfortunately, the Web's security model differs from these expectations, resulting in common web vulnerabilities.

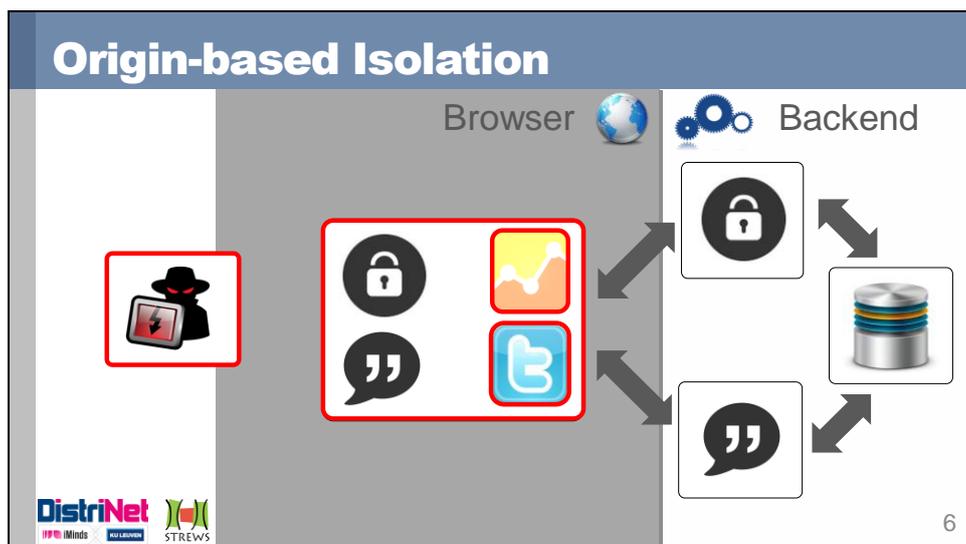
Deploying *example.com* in the Web

It can't be that simple, right?



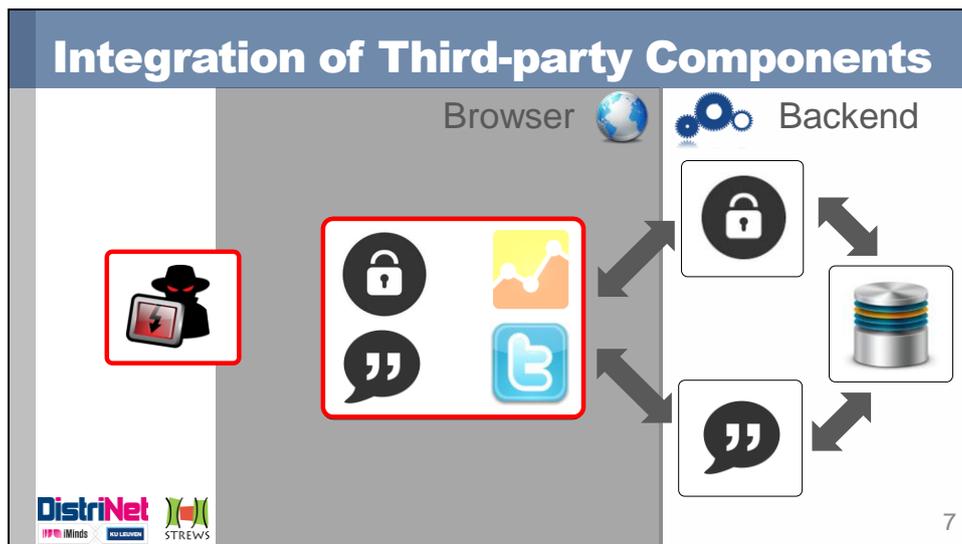

5

It isn't ☹️



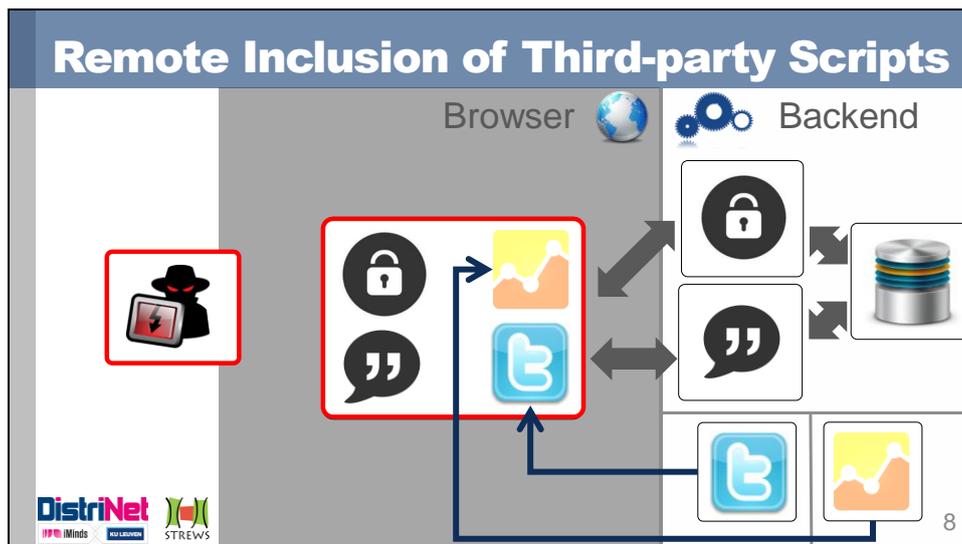
At the client side, most security policies are enforced on origins, which consist of the triple $\langle \text{scheme}, \text{host}, \text{port} \rangle$ (e.g. $\langle \text{http}, \text{example.com}, 80 \rangle$). The most prominent example is the Same-Origin Policy (SOP), which isolates contexts from different origins within the browser. Additionally, script-based fetching mechanisms (e.g. XMLHttpRequest), permission systems (e.g. Geolocation), etc. enforce their security policy on the level of origins.

For our example application, this means that there is no effective isolation between the client-side contexts. One consequence is the escalation impact of a vulnerability in one part of the site, potentially giving the attacker unrestricted access to the entire web application.



Integration of third-party components is a common practice in modern web development. Code can be included as JavaScript, in which case it runs within the security context of the including page (e.g. `example.com`), where it runs unrestricted. An alternative is including an entire document in an iframe, where a developer can benefit from the same-origin policy to enforce some boundaries, if the included code lives in another origin than the including page.

In our example, this means that the analytics code and the twitter gadget, both included JavaScript code, actually run within the security context of `example.com`. The included code has unrestricted access to `example.com`'s client-side data and APIs, and can also send requests to the backend, impersonating a legitimate user. While giving trusted code such unrestricted access might be an acceptable risk, it certainly opens the door for abuse, either willfully or through compromise of the third-party provider.



As said before, the integration of third-party components is a common practice. Very often, these components are included directly from the third-party provider, by using the URL of the remote script as the source of the script tag (e.g. `<script src="http://analytics.com/...">`). If including third-party code posed a security risk, including it directly from the provider increases this risk, due to lack of control over the third-party provider's operational context. A compromise of the third-party provider automatically escalates towards a compromise of your web application.

In our example, the analytics code and the twitter gadget are loaded directly from the provider, posing a potential security risk in case of compromise.

Compromise of Third-party Providers

If you downloaded the qTip2 library between 8th December 2011 and 10th of January 2012, please make sure to re-download the library as the site was compromised between these dates due to malicious code injected via a Wordpress bug. Apologies for any inconvenience caused by this, but as usual vulnerabilities like this can only be pro-actively remedied as they occur.

Download latest: 1.0.0-rc3

What package would you like?

If you downloaded the qTip2 library between 8th December 2011 and 10th of January 2012, please make sure to re-download the library as the site was compromised between these dates due to malicious code injected via a Wordpress bug. Apologies for any inconvenience caused by this, but as usual vulnerabilities like this can only be pro-actively remedied as they occur.

9

One example of the compromise of a third-party code provider. The qTip2 library, a jQuery plugin, has been compromised for **32 days (!)** before it has been noticed, causing malware insertions on numerous websites during this period.

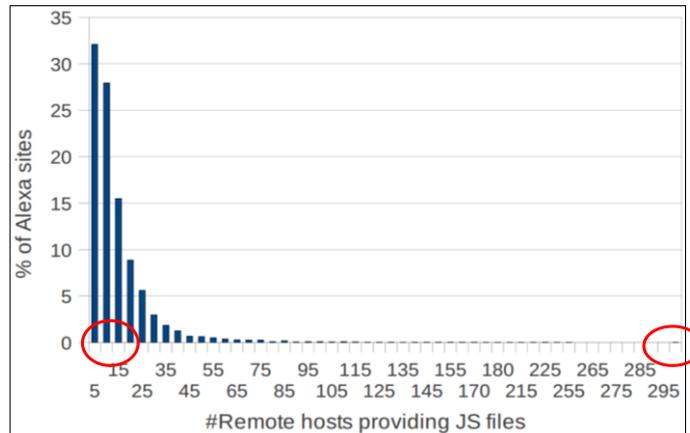
Large-scale Study of Remote JS Inclusions

“88.45% of the Alexa top 10,000 web sites included at least one remote JavaScript library”

10

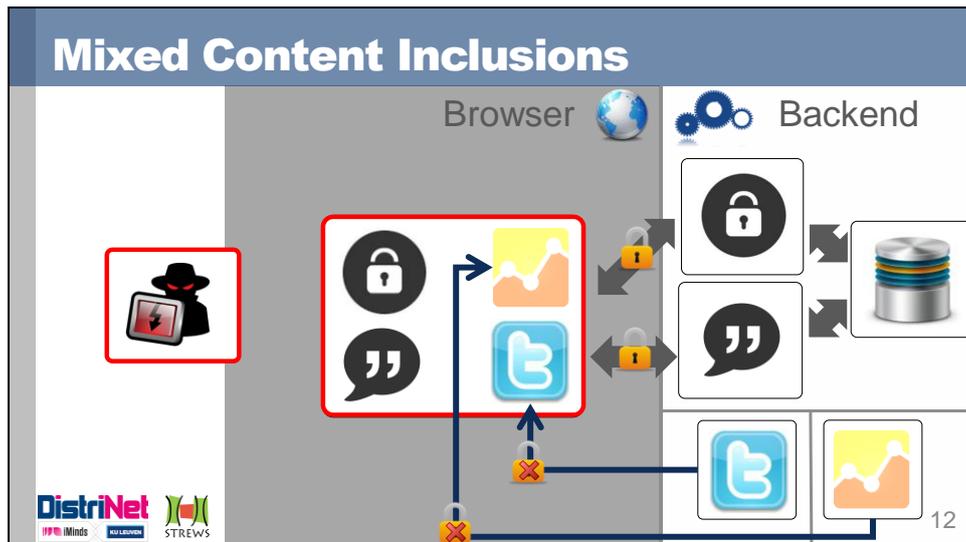
Reference: *You Are what You Include: Large-scale Evaluation of Remote JavaScript Inclusions*, N. Nikiforakis et al.

Large-scale Study of Remote JS Inclusions



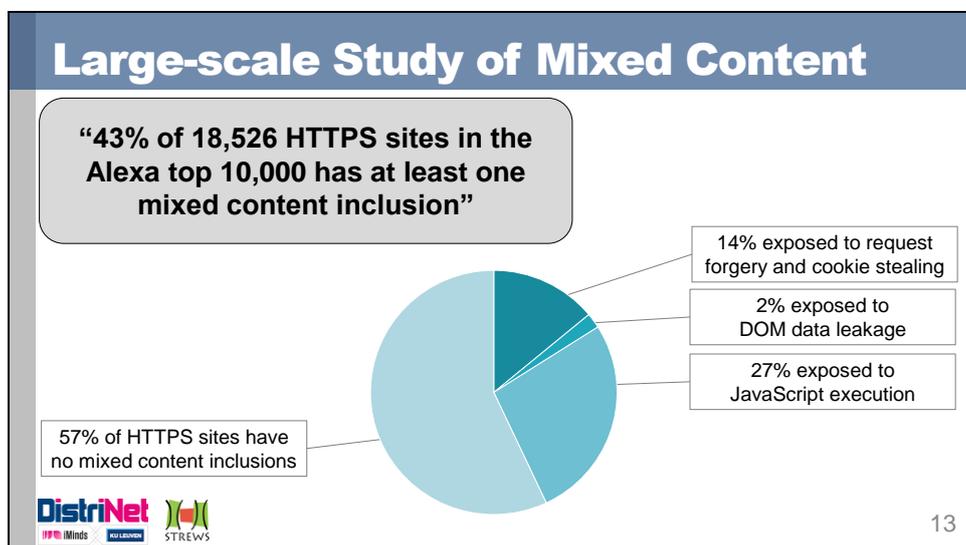
The study shows that the majority of sites includes a few JavaScript libraries (the left part of the graph), but that some sites include scripts from numerous distinct hosts. The most extreme example included scripts from 295 distinct hosts. Additionally, the study proposed a metric to measure the *security-consciousness* of sites, and concluded that 12% of sites considered security-conscious include at least one script from providers considered to be not security-conscious.

Reference: *You Are what You Include: Large-scale Evaluation of Remote JavaScript Inclusions*, N. Nikiforakis et al.



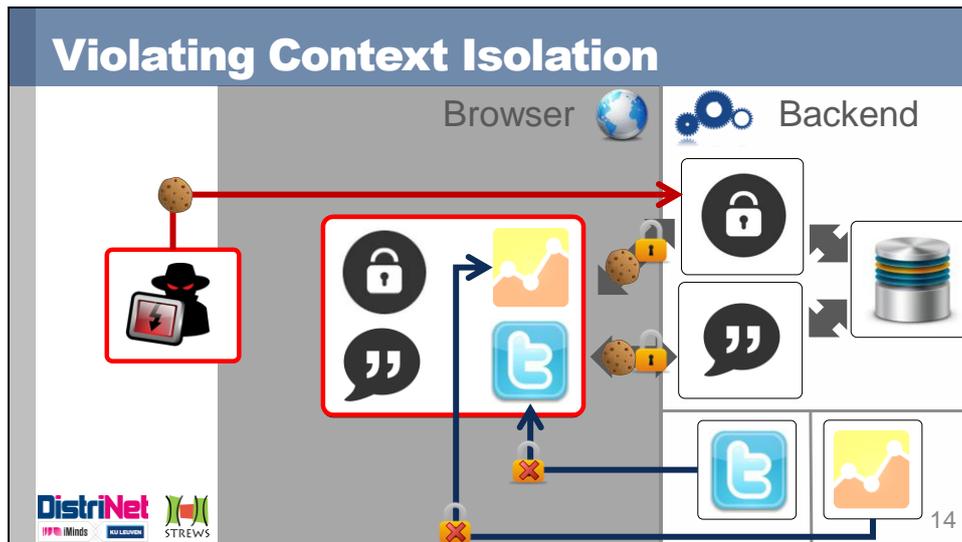
Deploying web applications over a TLS-secured channel (HTTPS) is a good practice, but introduces a certain complexity. HTTPS-pages that include content from third-party content providers often fetch these resources over a plain HTTP channel, leaving those resources vulnerable to attacker manipulation.

In our example, we accidentally load the analytics code and twitter gadget over an insecure channel, allowing an attacker to provide a compromised version, as well as to snoop on the data transmitted with the request.



A study of the top 10,000 domains discovered 18,526 HTTPS sites, of which 43% include at least one type of mixed content. While offering their site over HTTPS is an important step forward, including non-HTTPS resources essentially nullifies a lot of the efforts.

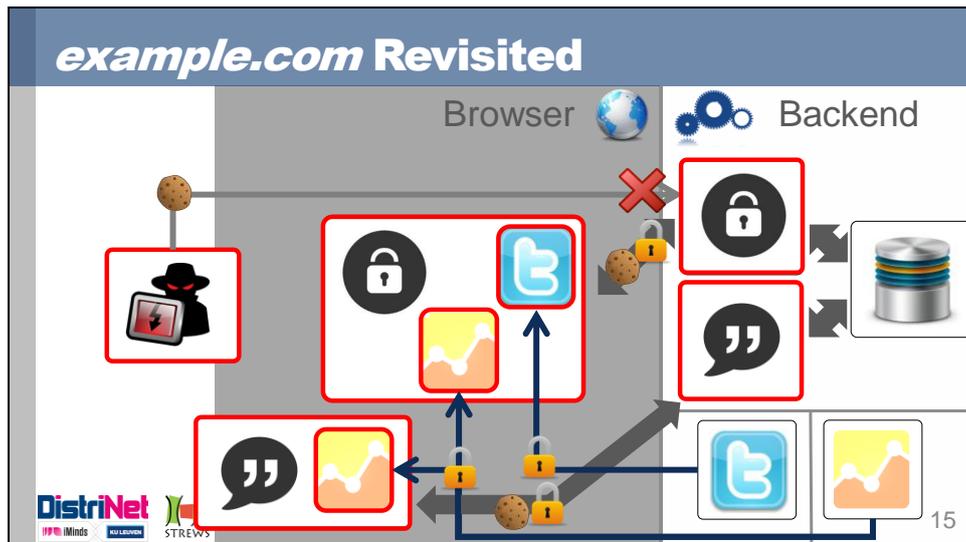
Reference: *Large-scale Analysis of Mixed-content Websites*, P. Cheng et al.



While client-side contexts are separated by origin, the Web's security model does not prevent cross-origin communication between a client-side context and a server-side resource. In itself, this communication is not a problem, and is often used in legitimate scenarios (e.g. loading a remote resource, using a server-side API, ...). However, due to the implicit authentication mechanisms used today, the browser actually adds authentication information to potentially illegitimate requests, causing a vulnerability known as Cross-Site Request Forgery. While defenses against CSRF have been known for a while, many websites fail to deploy them correctly, leaving vulnerabilities open for attackers. Examples of CSRF victims are Google, Facebook, Youtube, online banking systems, webshops, etc.

In our example, an attacker-controlled context can easily send requests towards the backend. The problem however is that the browser adds the authentication cookie used for *example.com*, allowing the attacker-controlled context to create requests that appear to come from the user. Using such forged requests, an attacker can execute any action in the user's name which is not explicitly protected against such attacks.

Reference: *Cross-site request forgeries: Exploitation and prevention*, W. Zeller et al.



In an ideal web, we'd like to deploy the example.com application within separate origins, effectively leveraging the current security model to separate the private customer part from the public forum. Additionally, within each client-side context, remote code only has restricted access, thereby minimizing the risk of potential misbehavior by the included code. Naturally, all the included resources are transferred over an HTTPS channel, preventing any eavesdropping and manipulation attacks. Finally, we'd like to prevent an "untrusted" context from making requests to the application's backend, thus eradicating CSRF attacks.

Challenges for this Session

- **Compartmentalization using origins**
 - Leverage the same-origin policy to isolate sensitive parts
- **Sharing information and authentication**
 - Share authentication information between contexts
 - Interact and exchange information between contexts
- **Managing third-party code inclusion**
 - Managing the risk associated with potentially untrusted code
 - Preventing mixed-content warnings
- **Communication with the backend**
 - Enable legitimate communication from HTML and JavaScript
 - Mitigate illegitimate requests from untrusted contexts

DistriNet | IIMinds | KUL LEUVEN | STREWS

16

Compartmentalization

- Separation based on origin
 - Naturally enforced by the Same-Origin Policy
 - Allows you to separate sensitive parts and non-sensitive parts
 - Prevents unintended sharing of information
 - Prevents escalation of successful attack

ORIGIN

The triple <scheme, host, port> derived from the document's URL. For *http://example.org/forum/*, the origin is <*http, example.org, 80*>

SAME-ORIGIN POLICY

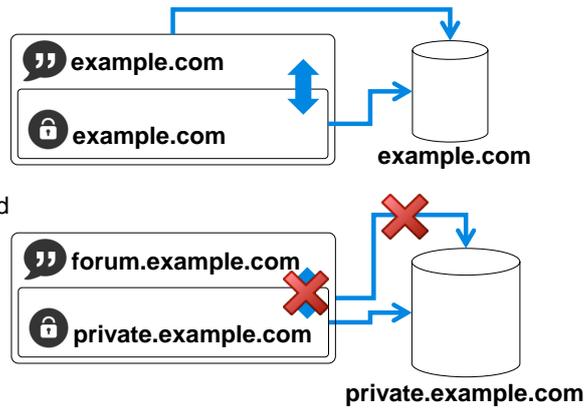
Content retrieved from one origin can freely interact with other content from that origin, but interactions with content from other origins are restricted

Deploying different parts of the application in different origins leverages the power of the same-origin policy, the cornerstone security policy deployed in every modern browser. Should an attacker succeed in compromising a non-sensitive part of the application, the same-origin policy will help in preventing a quick escalation of the attack towards the sensitive parts of the application.

Examples of the Same-Origin Policy

SAME-ORIGIN POLICY

Content retrieved from one origin can freely interact with other content from that origin, but interactions with content from other origins are restricted



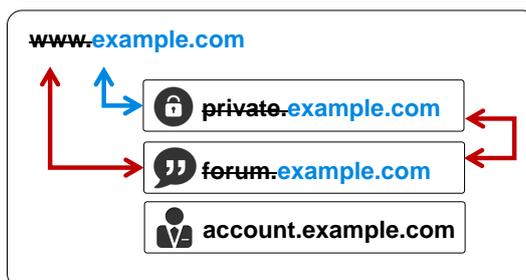
The top example shows both parts of the application residing in the same origin, and the bottom example shows a compartmentalized application using subdomains. In both cases, the public forum page is compromised by an attacker, who uses cross-site scripting techniques to embed an iframe, which in turn loads the private part of the web application. In the top example, the same-origin policy will not restrict the forum page from accessing, inspecting and modifying the private page in the iframe. In the bottom example, the same-origin policy will restrict this behavior, as the origin of both documents differs ($\text{forum.example.com} \neq \text{private.example.com}$).

Note that any data stored within the origin (e.g. Web Storage, IndexedDB, API permissions, cookies, ...) is also available to any resource within that origin (the forum page in the top example). Correct compartmentalization prevents such unintended leaks, as shown in the bottom example.

Domains vs Subdomains

- Subdomains
 - E.g. **private.example.com** vs **forum.example.com**
 - Considered different origin
 - Possibility to relax the origin to **example.com** using *document.domain*
 - Possibility to use cookies on **example.com**
- Completely separate domains
 - E.g. **private.example.com** vs **exampleforum.com**
 - Considered different origin, without possibility of relaxation
 - No possibility of shared cookies

Subdomains and Domain Relaxation



DOMAIN RELAXATION

```
document.domain = "example.com";
```

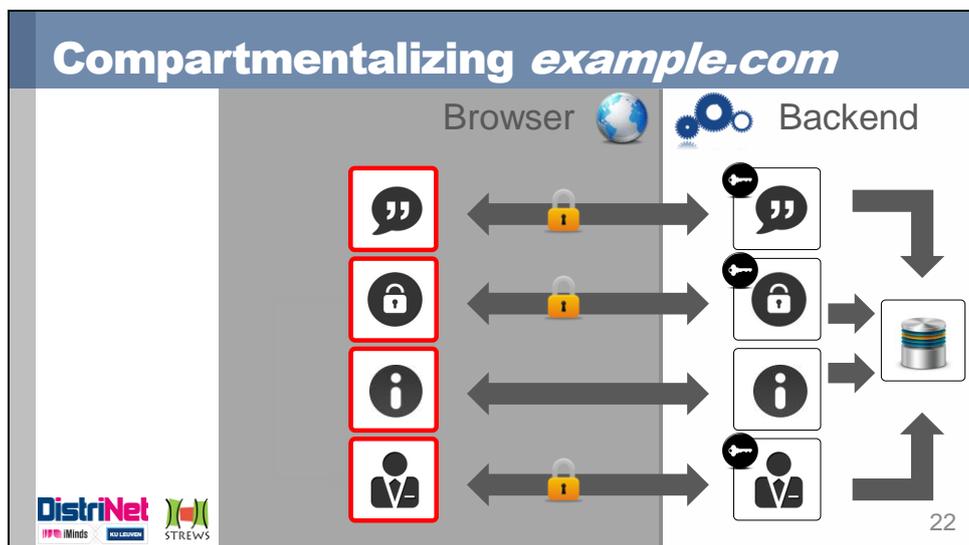
Documents that have the same parent domain (*example.com*) can choose to relax their effective origin to *example.com*. Once they have done so, they can interact freely with other documents that have the *example.com* origin, but not with documents that have not relaxed their origin. In this example, the main document and the forum document have relaxed their origin to enable interaction.

Note that once an origin is relaxed, there is no turning back (except for reloading the document). There is also no restriction on which other documents under a subdomain of *example.com* can join the club, which may be undesired. In this example, the forum page relaxed its origin as well, enabling it to inspect its parent document, as well as the private sibling document.

Compartmentalizing *example.com*

	 Public Information	 Private Customer	 Account Management	 Public Forum
Sensitive Content	no	yes	yes	no
Requires authentication	no	yes	yes	yes
Deploy over HTTPS	preferable	yes	yes	yes
Needs cooperation	no	account	private	no
Origin	www.example.com		account.example.com	
		private.example.com	exampleforum.com	

   21



Separating the different parts of the *example.com* application into subdomains or alternate domains effectively leverages the same-origin policy to provide client-side isolation between contexts. As indicated, the account management and private customer area still need to communicate, which will be covered later. Additionally, several parts require authentication, implying that they also need to be deployed over an HTTPS channel.

In the next part, we will investigate how several isolated contexts can share authentication information, preventing a user from having to authenticate separately for each part of the application. Additionally, we will discover how isolated contexts can exchange information via an opt-in messaging mechanism.

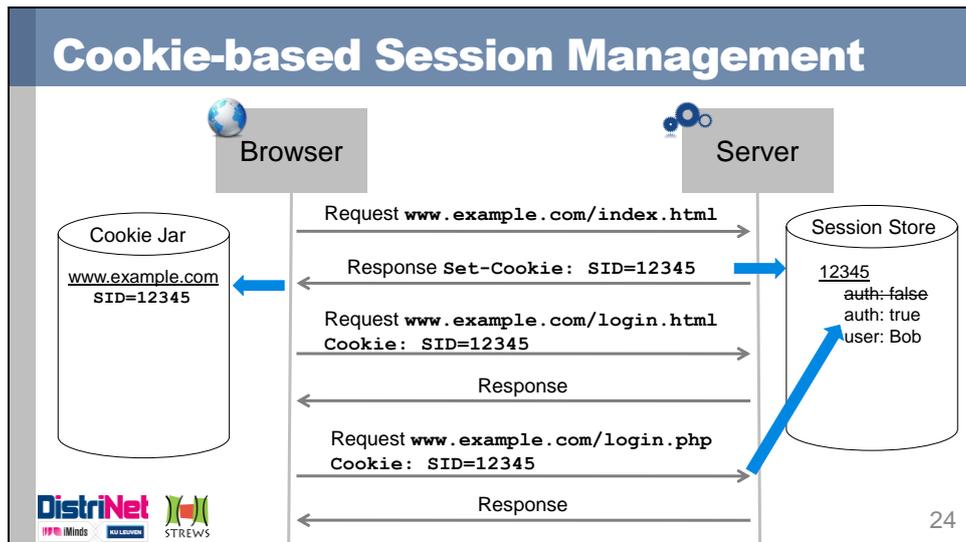
Authentication on the Web

- Typical authentication consists of two steps
 - Entity authentication
 - Maintaining the session associated with the authenticated user
- Entity authentication
 - Exchanging username and password
 - Challenge/response systems are also used
- Session management
 - De facto standard is cookie-based session management
 - Cookie contains unique identifier, associated with server-side state

Authentication on the Web consists of two important aspects:

- **Entity authentication:** The process of identifying the authenticating party. On the Web, this is typically done by sending a username and password to the server. Alternative systems use client certificates for a secure TLS connection (e.g. with the Belgian EID), a challenge/response system (e.g. the Belgian banking systems), etc.
- **Session management:** Session management is the process of maintaining a session specific to a certain user, allowing the web server to tie multiple requests together in a single session. If a user authenticates himself to the web application, this information is typically stored within the session. Note that a session can also exist for an anonymous user, who has not yet authenticated himself. The de facto session management mechanism on the web is based on cookies. Often, parameter-based session management (a variable in the URL) is used as a fallback.

These topics are covered in detail in the session “Entity Authentication and Session Management” by Jim Manico



The browser has a cookie-jar, where cookies are stored per domain. When a browser sends a request to a server without a session identifier, typically the server will create a new session with a random identifier, and send the identifier in a cookie to the browser. The browser stores this cookie, and will include it in any future request towards the associated domain. When the server receives a request with a session cookie attached, the server will first lookup the associated session, after which the request is handled, while potentially using information from the session (e.g. authentication state, ...).

Modifying Cookie Behavior

- **Domain**
 - Allows to broaden the applicability of the cookie
 - E.g. *example.com* applies cookie to **.example.com*
- **Path**
 - Associates a cookie with a specific path
 - E.g. */admin/* associates a cookie with */admin/**
 - Conflicts with the same-origin policy
- **HttpOnly**
 - Restricts a cookie from being accessed through JavaScript

http://user.example.com/attacker/

http://user.example.com/victim/

The SOP allows direct access to the iframe, exposing *document.cookie*

25

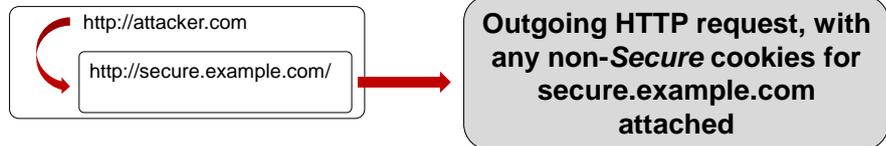
Cookie behavior can be modified using a set of predefined attributes, which are added to the *Set-Cookie* header by the server. The available attributes are Domain, Path, Expires, HttpOnly and Secure.

- **Domain:** The domain attribute allows to broaden the applicability of the cookie. By setting this parameter to a parent domain of the current document, the browser will send the cookie to all child domains of this parent domain. For example, setting a cookie with *Domain=example.com* on a document hosted on *foo.example.com* will cause the browser to send the cookie to any child domain of *example.com*.
- **Path:** The path attribute allows to limit the cookie to a specific path within the domain. The browser will only send the cookie if the path value matches the URL of the document being fetched. Note however that a mismatch between the cookie security policy and the Same-Origin Policy allows to steal the cookie through JavaScript. If an attacker page on *http://user.example.com/attacker/* includes an iframe, with a document from *http://user.example.com/victim/*, it is allowed by the SOP to directly interact with the context of this iframe, since both documents have the same origin. The attacker page can now use JavaScript to access the iframe and request the value of the *document.cookie* attribute, which contains the cookie associated with the */victim/* path.
- **Expires:** Allows the specification of an expiration date for the cookie, after which the browser will no longer use it.
- **HttpOnly:** Specifying this attribute (without value) tells the browser to use this cookie when making requests, but to not disclose this cookie through the *document.cookie* property in JavaScript. This effectively protects the cookie against script-based cookie-stealing attacks, such as session hijacking.
- **Secure:** The secure attribute tells the browser to only send this cookie over an HTTPS connection. The next slide covers this attribute in detail.

Cookies and HTTPS deployments

- Why the *Secure* flag matters

- Cookies are associated with a domain, not an origin
- No separation between cookies used on HTTP and HTTPS requests

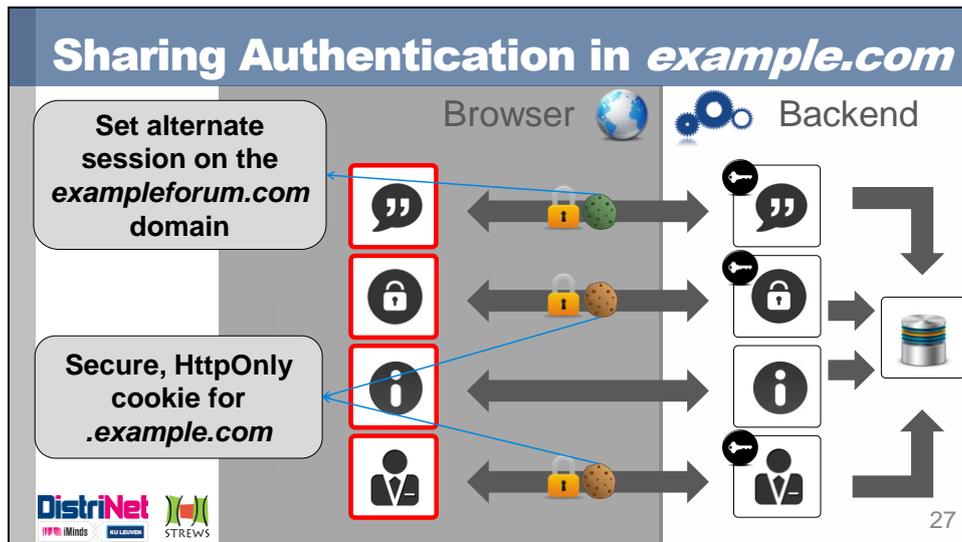


- Use separate cookies for HTTP and HTTPS

- Associate different security levels to each cookie
- Require HTTPS cookie to be present for sensitive operations

The *Secure* attribute for cookies ensures that cookies issued on HTTPS connections will never be sent on plain HTTP connections. This is important, since omitting this flag effectively enables leaking of cookies. An attacker can simply load a subresource of *secure.example.com* over an insecure connection, causing the browser to attach any non-secure cookie. If an attacker is eavesdropping on the network traffic, this suffices to lead to a session hijacking attack.

If cookies are used on both the HTTP part and the HTTPS part of the site, the developer needs to take care that the HTTPS cookies are marked as *Secure*, and the HTTP part uses an alternative set of cookies. Additionally, any sensitive operation at the server-side should check the presence of the HTTPS session identifier, in order to prevent a session hijacking attack.



In our example application, both the private customer part and the account management part require authentication. This authentication can be shared by setting the session cookie on the *example.com* domain with the *Secure* flag, causing it to be sent to every subdomain of *example.com*, but only over secure connections. Additionally, setting the *HttpOnly* flag prevents any JavaScript-based cookie-stealing attacks.

The *exampleforum.com* part also requires authentication, but since it lives on a different domain, the already-existing session cookie can not be easily shared. One solution is to issue an alternate session ID for the *exampleforum.com* domain, which is set after the user has authenticated himself. At the server-side, both session IDs refer to the same session. At the client-side, issuing an alternate session identifier fits within the compartmentalization strategy, since a stolen session identifier from *exampleforum.com* will not be valid on a subdomain of *example.com* and vice versa.

Setting the alternate session identifier after login can be done with a simple request to a server-side cookie-setting script. The new identifier is included as a URL parameter, and bounced by the script as a session cookie (e.g. a request to <https://exampleforum.com/setSession.php?ALTSID=abcde> will issue a response with a *Set-Cookie: SID=abcde; Secure; HttpOnly* header). This technique is commonly deployed on the Web, for example by the Google services.

Interaction between Contexts

- Related contexts
 - Documents can open popup windows, embed frames, etc.
 - Related cross-origin contexts are isolated by default
 - Limited interactions possible (navigation, messaging APIs, ...)
- Navigation
 - Navigate child frame to different resource
 - Navigate parent frame, reloading the entire document
- Exposed APIs
 - Prime example: **Web Messaging API**, to support interaction

By default, different windows or tabs in a browser are isolated from each other, regardless of the same-origin policy. However, documents embedded by means of iframes, or popup windows opened from within a page are capable of interaction, if allowed by the same-origin policy. For example, a document embedding an iframe from the same origin has full access to the frame's context. Embedding an iframe from a different origin triggers the SOP, which limits interaction to navigation of the frame and access to a limited set of exposed APIs. All other interactions will result in an exception, typically seen as a *DOM Exception* explaining the violation of the SOP

Web Messaging API

- Messaging mechanism between contexts
 - Used for iframes, Web Workers, etc.
 - Event listener for receiving messages (opt-in mechanism)
 - API function for sending data (text, objects, etc.)
- Security considerations
 - Specify origin of receiver to prevent leaking of content
 - Check origin of sender to prevent malicious use
 - Validate incoming content before using data to prevent injection attacks

The Web Messaging API supports sending messages between contexts. A message can consist of a simple String, or a cloneable object. Receiving messages is implemented by means of an event handler, which listens to incoming messages. If an event handler is not registered, incoming messages are simply discarded. Sending a message is done using the *postMessage* function, an API call exposed on the *window* object of the context. This API call is also accessible from cross-origin contexts, as long as a reference to the context can be obtained.

When using Web Messaging, several security considerations need to be taken into account. First, when sending a message, you need to specify the origin of the receiver, to ensure that the message is delivered to the right party. Second, when receiving messages, you need to verify the origin of the sender of the message. Failing to do so enables an attacker to send arbitrary messages to your context, potentially causing harmful effects. Finally, when receiving content which is processed further, validate the content before using it, in order to prevent injection attacks.

Reference: *Web Messaging API*, W3C

Web Messaging API



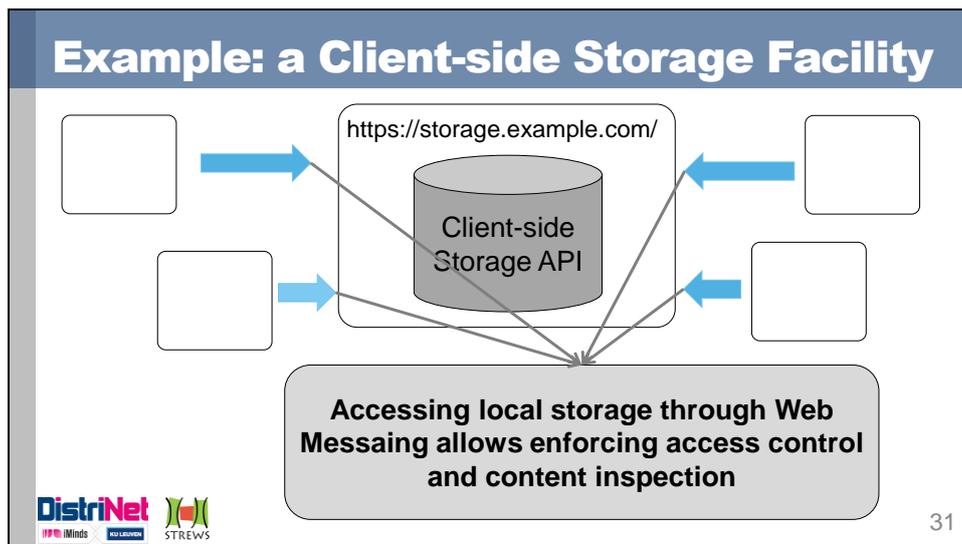
SENDING MESSAGES

```
myframe.postMessage(data, 'http://test.example.com');
```

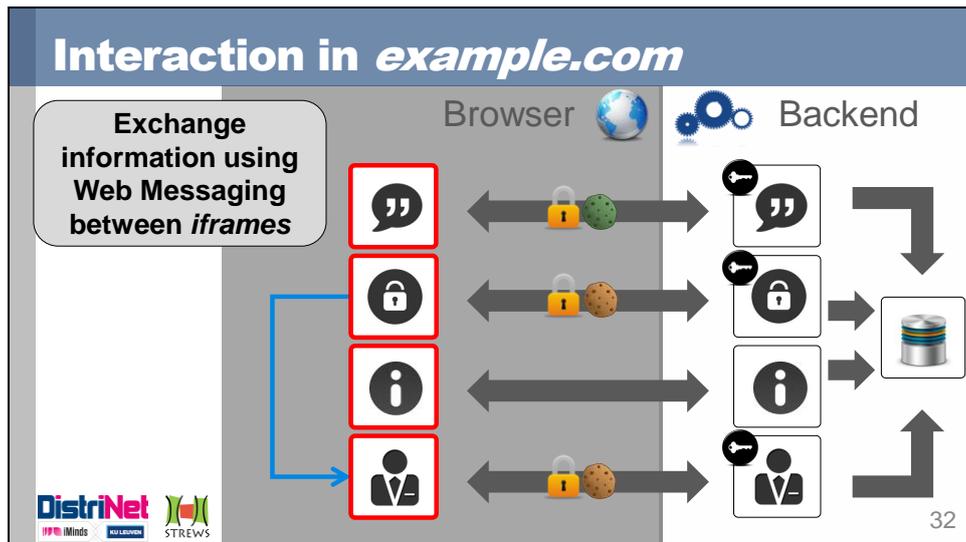


RECEIVING MESSAGES

```
var handler = function(event) {  
  if(event.origin ==  
    'http://www.example.com') {  
    alert(event.data);  
  }  
}  
window.addEventListener('message', handler, false);
```



The example in this slide offers a client-side storage facility, enabling the storage of application data within the browser, while keeping control over who access what data, and what content is stored in the data. The code managing the locally stored data (which is bound to an origin) is hosted on a designated origin, which hosts no other application code. Any context that needs access (read or write) to locally stored data can include the storage code in an iframe. Within this frame, this storage code has access to the data stored within its origin in the browser. Using the Web Messaging API, the context can send messages to the storage context, asking to read or write some data. The storage context can check the origin of the sender, and decide whether the operation is actually allowed or not. Additionally, the storage context can enforce additional restrictions, such as the data format, the length of the data, etc.



In our example, the private customer area needs to communicate with the account management area, to fetch account-related information. Making both contexts same-origin using the *document.domain* attribute is one option, but this would allow other subdomains of *example.com* to do the same, and gain undesired access to the private and account management contexts. Therefore, we choose to embed a communication module of the account management into an iframe, which we can then contact using the Web Messaging API. The communication module is able to enforce a policy, exposing only the functionality and data that it chooses.

Including Remote Content

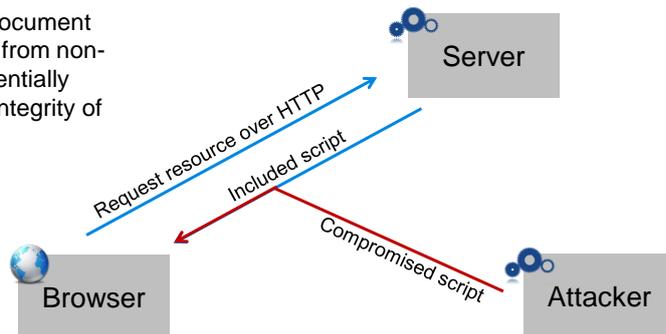
- Types of remote content
 - Images
 - JavaScript
 - CSS Styles
 - HTML documents
 - SVG images
 - Audio/video
 - Plugin content (Flash, Java)
 - ...
- Including remote content
 - Identified by a URL
 - Fetched by the browser, and subsequently integrated
 - For active content (e.g. JavaScript), the included code is typically executed in the context of the including page

Remote content can be found everywhere on the Web, ranging from a simple image inclusion to loading entire frameworks through third-party JavaScript files. Numerous HTML tags support the inclusion of remote content by specifying a URL for the source. The browser will fetch the resource identified by the URL, and process it accordingly. For an image, this means merely displaying the image in the dedicated location. For a script, this effectively means executing the script code within the context of the page.

Mixed Content Problems

MIXED CONTENT INCLUSION

When an HTTPS-document includes resources from non-HTTP sources, potentially compromising the integrity of the document



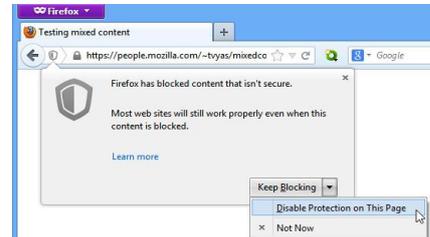
Loaded HTTPS page which requires additional resources

Deploying a site over HTTPS results in several security benefits, but also has an impact on the web application itself. One example is the problem of mixed content inclusion, where an HTTPS page includes content from non-HTTPS sources. Since such content can be compromised by network attackers (e.g. a Man-in-the-Middle attack), it might compromise the integrity of the entire page. In the example on this slide, an attacker succeeds in compromising a included script from an HTTP source, giving him full control over the client-side context of the including page.

Reference: *Large-scale Analysis of Mixed-content Websites*, P. Cheng et al.

Solving Mixed Content Problems

- Browsers blocking mixed content inclusion
 - IE 7 started with prompting users, other browsers are following
 - **Active mixed content** is typically blocked, passive content is allowed



- Localize remote resources
 - Host remote resources locally within the application's HTTPS domain

Surprisingly, the solution to avoiding mixed content problems is to avoid including non-HTTPS resources in HTTPS documents. Popular resources (e.g. popular libraries, ...) are available on secure links, or hosted by providers such as Google. Alternatively, the non-secure resources can be placed within the application's HTTPS domain, allowing for a secure inclusion in HTTPS pages. The downside of this approach is the need to keep the imported resource up to date with its original repository.

At the browser-side, action is being taken to avoid mixed content security issues. IE7 was the first browser to warn about mixed content inclusions, and recently, other browsers are following. Typically, active mixed content (script, stylesheets, Flash) is blocked, but passive mixed content (e.g. images) are allowed to be loaded.

Reference: <https://blog.mozilla.org/tanvi/2013/04/10/mixed-content-blocking-enabled-in-firefox-23/>

Integration of Remote Code

- Two mechanisms to integrate code
 - Directly including JavaScript code using the `<script>` tag
 - Including code through an *iframe*, which hosts a separate document
- Scripts
 - Straightforward integration in the context, without restrictions
 - Violates the security boundaries of a document
- Iframes
 - Depending on the origin, the SOP restrictions apply
 - Preserves the security boundaries, but may hinder interaction

Integration of remote code is a common practice on the modern Web. Examples are the inclusion of libraries such as Google Analytics, advertisements, etc. HTML supports two integration mechanisms: direct integration through the script tag, or the integration of a sub-document using an *iframe*. The former loads the remote script in the security context of the including page, effectively importing the script's functions and variables into the current execution context. Script integration does not offer any security boundaries, but also does not restrict interaction between the included script and the hosting page. The latter solution, *iframe* integration, requires an entire document to be loaded, which is instantiated with its own execution context and security boundaries. If the origin of the *iframe* differs from the including page, the SOP applies and the origin-based security boundaries are enforced. In this case, free interaction between both contexts is no longer possible, but communication mechanisms such as Web Messaging can be used to achieve controlled interaction.

Script-based Content Integration

- No security boundaries offered by browser
 - Combination with remote providers is potentially dangerous
 - Full access to the client-side context, including local resources
- Existing techniques to constrain scripts
 - Localizing scripts → requires effort to update
 - Safe subsets of JavaScript → requires compatibility with existing scripts
 - Browser-based sandboxing → requires modifications to the browser
 - Server-side rewriting → requires control over the scripts
 - JavaScript-based sandboxing → upcoming technology

When integrating remote code through script inclusion, the remote code is imported into the security context of the including page. This effectively grants the remote code the same privileges as the rest of the page, giving the code full access to the page's contents, cookies, data stored at the client side, permissions granted to the page's origin, etc. These considerations need to be taken into account when including a script into the page, as it implies a certain trust in the included script. Not only does the script have to be trusted, but the provider also needs to be trusted to be legitimate, and to remain free of compromise in the future.

Currently, there is no silver bullet solution to safely integrate remote code using script tags. However, several different approaches have been proposed to mitigate the risks associated with direct script inclusion:

- **Localizing scripts:** by hosting a controlled copy of the remote script, the risk of a compromise of the provider can be severely limited. However, this approach requires a developer to keep up to date with the remote scripts. Research suggests that a weekly update schedule should suffice for this approach [1].
- **Safe subsets of JavaScript:** By limiting a script to a specific subset, one can ensure that the script can only access the features that are exposed, preventing it from gaining full access to the page's context. One example of this technique is Adsafe, a subset specifically aimed at the safe inclusion of JavaScript advertisements [2]
- **Browser-based sandboxing:** Several solutions modify the browser to isolate scripts and enforce fine-grained security policies on them. This effectively restrains a script to a specific set of accessible features. Unfortunately, these approaches require browser modifications, which make deployment of such security measures difficult [3]
- **Server-side rewriting:** By rewriting JavaScript code at the server-side, one can verify whether a script adheres to a predefined security policy. One example of this technique is Google Caja [4], which transforms a compliant script into a secure module, assuring that the script will be constrained. The disadvantage of these

techniques is that they require control over the server-side JavaScript code, in order to support the rewriting process.

- **JavaScript-based Sandboxing**: this approach brings the technique from server-side rewriting to the client-side, ensuring that a script is contained in a sandbox, where it can only access the features that it's permitted to use. These techniques are promising, but still subject to active research [5]. Consequently, they are not ready for production use yet.

References:

[1] *You Are what You Include: Large-scale Evaluation of Remote JavaScript Inclusions*, N. Nikiforakis et al.

[2] <http://www.adsafe.org/>

[3] *WebJail: least-privilege integration of third-party components in web mashups*, S. Van Acker et al.

[4] <http://code.google.com/p/google-caja/>

[5] *JSand: Complete client-side sandboxing of third-party JavaScript without browser modifications*, P. Agten et al.

Iframe-based Content Integration

- Iframes are controlled by the same-origin policy
 - Documents with different origins are isolated by the SOP
 - Well-suited to integrate separate components (e.g. advertisements)
 - More difficult to achieve dynamic interaction
- HTML5 introduces the *sandbox* attribute
 - Gives coarse-grained control over capabilities in an iframe
 - Supports disabling scripts, plugins, forms, etc.
 - Supports a unique origin, alienating the iframe from any other origin
 - Well-suited for the integration of untrusted content

Iframes are a natural boundary within the browser, as the Same-Origin Policy effectively isolates documents with different origins. Iframes are commonly used to include content that does not require interaction with the hosting page. Examples are advertisements, social media buttons, etc. One disadvantage of using iframes with different origins is the lack of direct interaction. Interaction is however possible using the recently introduced Web Messaging API, which will be covered in the next slide.

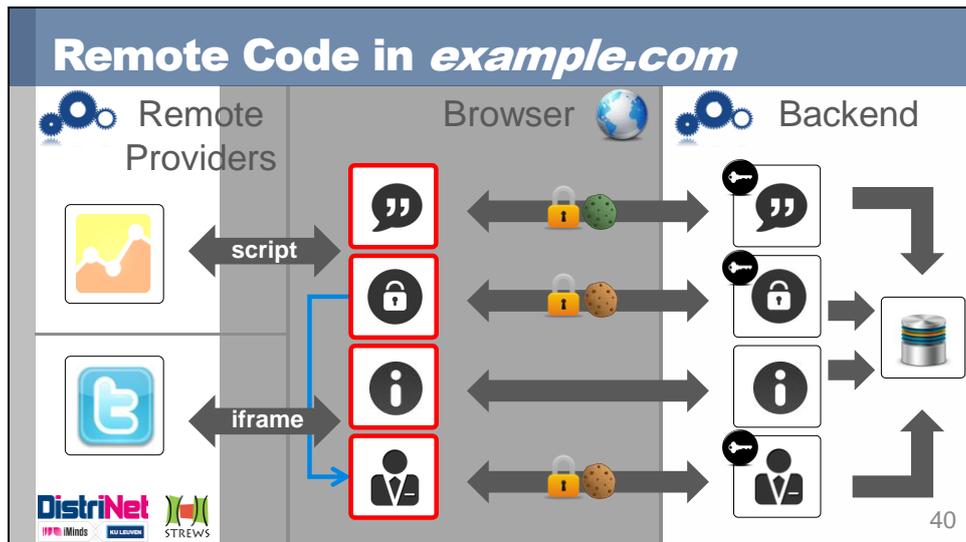
To enhance the security properties of iframes, HTML5 introduces the *sandbox* attribute. The attribute can be used to configure further restrictions on the content within the iframe. For example, by default, a sandboxed iframe is prevented from running scripts, running plugin content, submitting forms, etc. Using the attribute, these permissions can be re-enabled if desired. Additionally, the *sandbox* attribute supports the assignment of a unique origin to the iframe, effectively alienating the iframe from any other content that originated from the same origin.

One ideal use case for the *sandbox* attribute is the inclusion of potentially untrusted content. If you want to display user-provided content, which might contain potential XSS attacks, you can simply load it in a sandboxed iframe with scripting disabled, effectively preventing the attack from ever being executed.

The HTML5 *sandbox* attribute is covered in more detail in the session on *Recent Web Security Technology* by Lieven Desmet

Best Practices for Integrating Code

- If possible, isolate the content in an iframe
 - Use the *sandbox* attribute to enforce even more restrictions
 - Especially true for untrusted content (e.g. user-provided)
- Only include code from trusted providers
 - Google often provides mirrors of popular libraries
- Localize scripts for crucial applications
 - Keep scripts regularly up-to-date
 - Perform code reviews of the differences between versions



In our example, we have to include the analytics code as a script, since it requires full access to the page, in order to monitor the user's behavior. In this case, the script comes from Google, and can be considered to be trusted. However, we need to be aware of the risk associated with the direct inclusion of the analytics script. Should future JS-based sandboxing technologies fully support the containment of the analytics script, giving it only access to the page and not to the entire context, this approach is preferable.

The twitter gadget does not require interaction with the page, and can be loaded in an iframe. Depending on the code being loaded in the iframe, it can either be loaded directly, or needs to be integrated in a self-hosted HTML page, which is then loaded in the iframe. In both cases, the *sandbox* attribute can be used to limit the features available to the framed content.

Interacting with Remote Services

- Ways to interact remotely
 - Triggered from HTML elements (image loads, forum submissions, ...)
 - Programmatically from JavaScript (XMLHttpRequest)
 - Using alternative protocols (Web Sockets, WebRTC, ...)
- Challenges with remote interaction
 - Difficult to determine which context a request originated from
 - Difficult to determine if a request was intended by the user

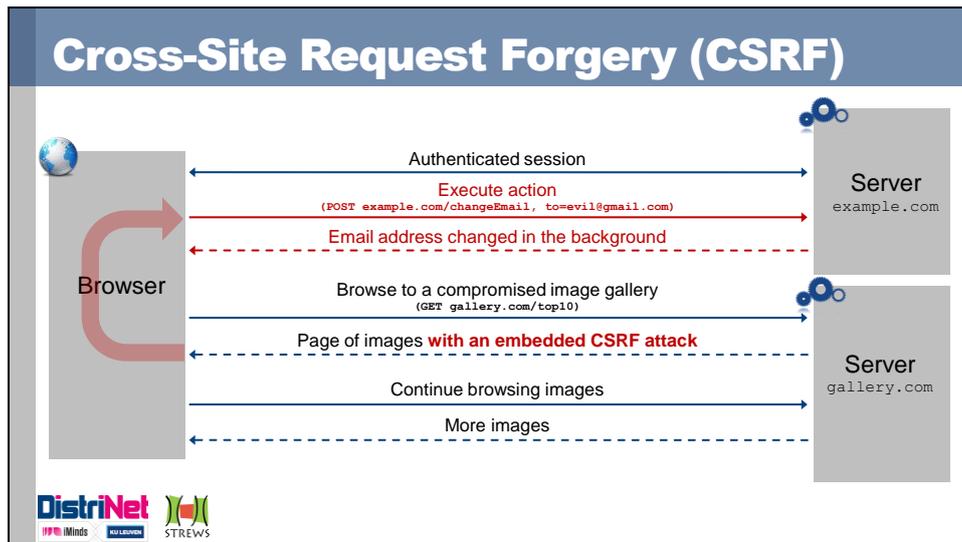
Remote interaction is an important concept in the Web, and several ways to trigger remote interaction are available. A first method is through existing HTML elements, like an image, a form, an iframe, etc. All of these elements trigger a request being sent from the browser, resulting in remote interaction with a backend service. A second way is by using JavaScript, more specifically the XMLHttpRequest API, to create and send custom requests. Finally, several alternate protocols are being developed and deployed, such as Web Sockets, WebRTC, etc.

Remote interaction originates in the browser, within some context, but the server typically does not possess this information. Therefore, the difficulty with remote interaction is to determine where a request originated from, and whether the request was actually intended to be sent, or part of an attack like Cross-Site Request Forgery.

HTML-based Remote Interaction

- Several types of requests can be triggered
 - GET requests from ``, `<script>`, ...
 - POST requests with control over body content from `<form>`
- Not affected by the Same-Origin Policy
 - GET and POST requests can be sent to other origin
 - Browser attaches available cookies to the request
- Session cookies are implicit authentication!
 - Results in an attack known as Cross-Site Request Forgery

HTML elements embedded in a document can trigger the browser to send a GET or a POST request to the target specified by the element. These requests are not restricted by the SOP, so it is possible to send requests to a different origin than that of the document where the request originates. Even more, the browser happily attaches any known cookies for the target origin to that request. This behavior can be legitimate (e.g. embedding part of a Facebook profile), but can also be malicious. The latter is known as Cross-Site Request Forgery, and basically allows an attacker to execute actions with a vulnerable service in the user's name.



In a CSRF attack, an attacker tricks the user's browser into sending a request in the user's name, causing unintended changes at the server side. In this example, the user has an authenticated session with *example.com*, and visits another site afterwards. This site has been compromised, and an attacker injected code to trigger a request towards *example.com*, to change the email address of the authenticated user. The moment the browser loaded the *gallery.com* page, with the embedded code of the attacker, a request to *example.com* is sent, with the available cookie attached. Key to the success of a CSRF attack is an existing authenticated session with the target application.

Reference: *Cross-Site Request Forgeries: Exploitation and Prevention*, W. Zeller et al.

Mitigating Cross-Site Request Forgery

- Mitigation techniques need to be explicitly present
 - Token-based approaches
 - Origin header



TOKEN-BASED APPROACH

example.com

```
<form action="submit.php">
  <input type="hidden" name="token"
    value="qasfj8j12adsjadu2223" />
  ...
</form>
```

When developing an application, the developer needs to explicitly take CSRF into account. Two common mitigation techniques are available

- **Token-based approaches:** embedding a token into pages that will trigger a request resulting in some action at the server-side is the most common approach. Any page of the site embeds such a token, and requires it to be present when the action is sent to the server. Since the attacker triggers the request from his page, no token will be present, and the request can be ignored. Stealing a token from a legitimate page within the users browser is not possible either, due to the restrictions enforced by the same origin policy.
- **Origin header checking:** Modern browser attach an *Origin* header to cross-origin requests. This header contains the origin where the request originated from, allowing the server to check whether the request is legitimate. If it comes from an unknown or unexpected origin, it can safely be ignored.

Programmatic Remote Interaction

- Sending requests with XMLHttpRequest
 - Supports different types of requests
 - Possibility to modify/manipulate “safe” headers
 - Response can be processed from within JavaScript



SENDING REQUESTS

```
var url = "http://test.example.com/api.php";
var req = new XMLHttpRequest();
req.open("GET", url, true);
req.onload = function(e) { ... }
req.send();
```

XMLHttpRequest is a JavaScript API allowing to create request objects, which can be configured and subsequently sent to a remote server. The XHR API supports different HTTP methods (GET, POST, PUT, DELETE), the use of custom headers, the use of custom body formats, etc. Some restrictions are imposed to prevent the manipulation of browser-provided security features, such as the Host header, the Origin header, etc. Other headers can be modified, such as the Cookie header, or custom created header fields.

XHR requests can be sent synchronously and asynchronously, and when the response is received, it can be processed from JavaScript. Response processing is entirely up to the code making the request. For example, when receiving HTML code, it can be placed inside an existing element on the page. When receiving data, for example in the JSON format, it can be parsed into a JavaScript variable. Other examples are receiving encrypted data from a secure service, which can be decrypted at the client-side, before being presented to the user

XMLHttpRequest and the SOP

- Same-origin requests
 - No restrictions imposed on the use of XMLHttpRequest
 - Custom headers, use of credentials, etc.
- Cross-origin requests
 - Required to enable remote interaction (e.g. APIs) without hacks
 - Enables capabilities not found in traditional HTML (e.g. PUT, DELETE)
 - Legacy server code does not expect such cross-origin requests
- New security policy: Cross-Origin Resource Sharing

Traditionally, the use of XMLHttpRequest was limited to the same origin as the origin of the document where the request originated. This functionality enabled the so-called AJAX capabilities, allowing a site to send requests in the background, to load additional information, contact a server-side API, etc.

With the growth of available online services, the need to use cross-origin APIs quickly emerged, followed by ad-hoc solutions that enabled this functionality within the existing Web security models. One way of achieving this is by loading a remote script file, with the data embedded in it, along with a function to process the data. The problem with this approach however, is that the remote site is not limited to only providing data, but can also add additional code, opening a new range of injection attacks.

In response to these valid needs and security concerns, the XMLHttpRequest API has been expanded to enable cross-origin requests. One important attention point in opening up the powerful features of XMLHttpRequest across origins are the security assumptions made by legacy servers. If a server provides an API with PUT and DELETE methods, which could previously only be used within the same origin, it now becomes vulnerable to cross-origin attacks. Therefore, the Cross-Origin Resource Sharing (CORS) security policy was introduced, aiming to ensure that cross-origin XHR requests have no more unwanted side effects than traditional HTML elements.

Cross-Origin Resource Sharing (CORS)

- Enables client-side cross-origin requests
 - Opt-in mechanism to grant other origins access to certain resources
 - Allows the easy use of online APIs without hacks
- Preventing additional attack vectors
 - Configurable security policy to determine who can access response
 - Preflight request to approve “dangerous” requests up front
 - Attacker capabilities with CORS largely correspond to HTML elements
- Already used beyond XMLHttpRequest
 - Regulating access to cross-origin HTML elements (canvas, ...)

The Cross-Origin Resource Sharing specification describes the security policy that is used for regulating access to cross-origin context. This specification is used by the XMLHttpRequest Level 2 specification to implement the cross-origin behavior. CORS is an opt-in mechanism for granting other origins access to certain resources or content. Legacy servers not aware of CORS will not provide the required headers, which leads to a default deny decision, like it would be in the pre-CORS Web.

The goals of the CORS specification is to prevent additional attack vectors beyond what is possible with traditional HTML elements. For example, with a *form* element, you can trigger a cross-origin POST request to any server. CORS does not prevent you from doing so, but does requires additional security headers when using content types that can not be used in combination with the *form* element. Similarly, cross-origin PUT and DELETE is not possible with traditional HTML elements, so will be subject to additional security headers.

This topic is covered in more detail in the session about *Recent Web Security Technology* by Lieven Desmet

Cross-Origin Resource Sharing (CORS)



SENDING CORS REQUESTS

```
var url = "http://api.provider.com/api.php";  
var req = new XMLHttpRequest();  
req.open("GET", url, true);  
xhr.withCredentials = true;  
req.onload = function(e) { ... }  
req.send();
```

CORS RESPONSE HEADERS

Access-Control-Allow-Origin: *http://www.example.com*

Access-Control-Allow-Credentials: *true*

Access-Control-Expose-Headers: *APIVersion*

Sending cross-origin XHR requests does not differ from sending same-origin XHR requests. The additional CORS security headers are added by the browser when the request is sent. The only additional client-side feature is the explicit flag *withCredentials*, telling the browser to use available credentials on the cross-origin request. Setting this flag can cause the browser to explicitly request permission from the server to send a request with credentials, to avoid new CSRF attack vectors.

CORS itself is implemented using request and response headers. For requests that are considered to be harmless (i.e. also possible with HTML elements), the browser simply makes the request. The server is responsible for checking whether this request is allowed to be made (i.e. check the origin header), and if so, process the request accordingly. When sending the response, the server can include several headers telling the browser how to proceed:

- **Access-Control-Allow-Origin:** This header contains the origin that is allowed to access the response. If this does not match the origin that made the request, the browser will prevent access to the response of the CORS request
- **Access-Control-Allow-Credentials:** Indicates whether the request can be made with credentials or not. If this does not match the credentials flag set when making the request, the browser will prevent access to the request
- **Access-Control-Expose-Headers:** Allows the server to specify a list of custom headers that can be exposed to the script making the request. Any headers not listed here will be hidden from the script processing the response.

For requests that are traditionally not possible with HTML elements, the browser will send a preflight request asking for permission to make the request. The preflight is an OPTIONS request, which has no side-effects in case a legacy server does not expect it. By responding with the appropriate CORS headers (there are more headers available to use with the preflight), the server can instruct the browser to either allow or deny the request about to be made. This approach effectively prevents any potentially dangerous requests from being sent to an unsuspecting server.

Sharing an API with CORS

PUBLIC CORS API (/API/PUBLIC/)

- Allow wildcard origin

Access-Control-Allow-Origin: *

RESTRICTED CORS API (/API/ACCOUNTS/)

- Allow the customer area origin
- Allow the use of credentials
- Expose the *X-Version* header

Access-Control-Allow-Origin: https://private.example.com

Access-Control-Allow-Credentials: true

Access-Control-Expose-Headers: X-Version

CORS PROCESSING CHECKLIST

- Check origin of request
- Check used method
- Perform traditional access control
- Execute request
- Add appropriate response headers

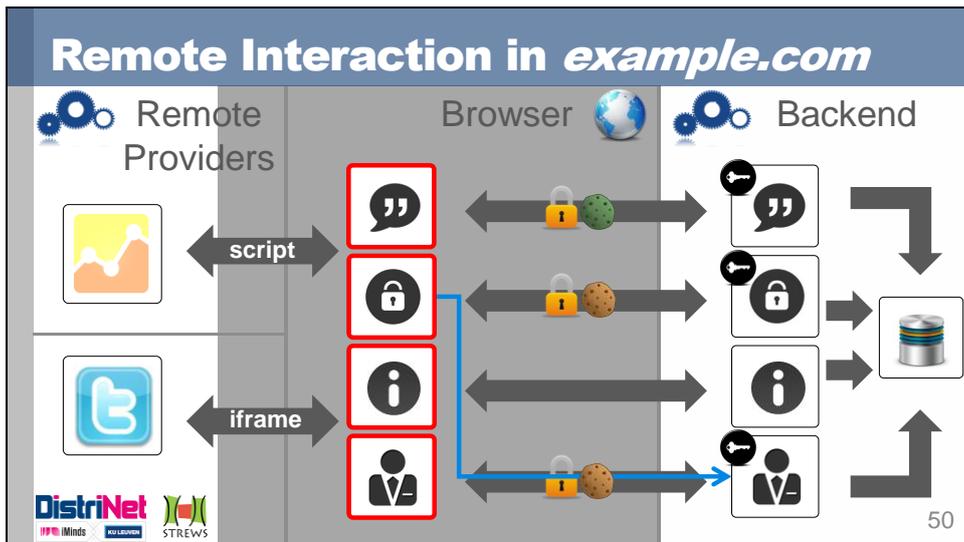


49

When processing a CORS request, it is important to check the properties of the request, before deciding to process it. First, check the origin where the request originated from, to be sure that it comes from a trusted origin (or any origin for public APIs). Second, check which HTTP method is used, and only allow those you expect. Third, you should apply your traditional access controls, such as checking the session for authentication status, etc. Once the request is executed, it is important to set the necessary CORS response headers, to allow the browser to grant access to the response.

In this slide, we give two examples of exposed CORS API. The first API is a publicly accessible API, where no restrictions are imposed. If you don't use credentials, and only serve simple requests, you can use a wildcard for the allowed origin. This API is accessible to anyone, and should therefore not disclose any private information.

The second API is a restricted API, disclosing account information. In our example application, this information is used by the private customer area, and hence the policies should only allow this origin to access this API. Furthermore, we allow the use of credentials and also allow access to the custom *X-Version* header, which denotes the version of the accounts API.



In our example, we previously enabled the private customer area to use an *iframe* and Web Messaging to interact with the account management code. With the introduction of CORS, we can alternatively directly use the account management API at the server-side to request account information and perform account-related actions. The account management API needs to expose the required operations, protected by the necessary CORS security checks and headers.

Wrap Up

DistriNet **iMinds** **KU Leuven** **STREWS**

51

Take-home Message

- The *origin* is a core concept in web security
- Compartmentalize where possible
- Treat incoming messages as potentially untrustworthy
- Consider the trust relationship with external parties

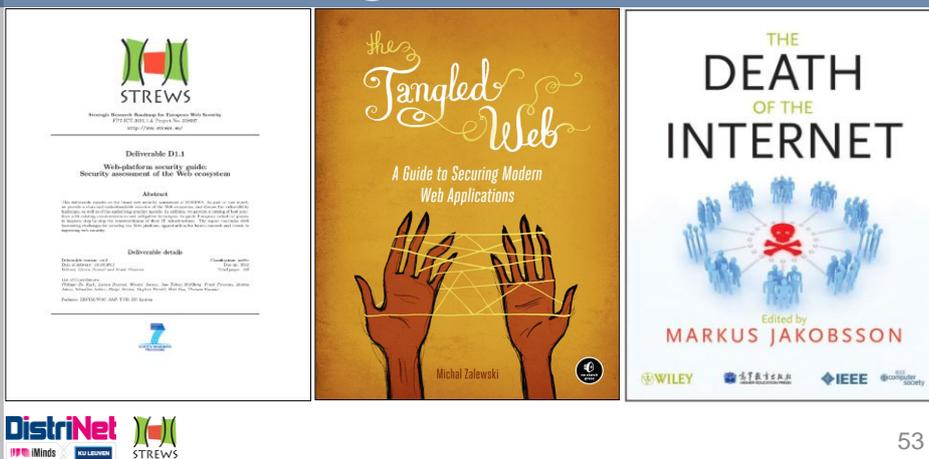
An origin is the core concept in the Web and a driving force in web security. Your origin hosts your content within the browser, has access to locally stored data and APIs and has associated user-granted permissions. Furthermore, the origin is key to determining remote access permissions, for example with the use of XHR and CORS.

A first advantage of compartmentalizing your web application is the introduction of natural barriers, which prevent a quick escalation when a part of the application is compromised. Additionally, it allows the necessary flexibility in deploying your application, such as deploying part over HTTPS with separate session cookies, exposing part of your application via a CORS API, etc.

An important advice, almost as old as the Web itself, is to treat incoming messages as potentially untrustworthy. Every incoming message, everywhere, should be treated with care, and should be validated. Three important questions need to be asked: where did the message come from? Does it contain valid content? Do I expect and allow this message? These questions hold for client-side interaction mechanisms such as Web Messaging, for remote communication messages such as CORS, or for server-side code such as dealing with SQL queries.

Finally, you should always consider the trust relationship with external parties. This is relevant when choosing providers for code components, but also when deciding how to integrate them into your application. If you are aware of the trust you place in these parties, you are in a position to re-evaluate this trust at anytime, preventing the continued use of potentially insecure code providers.

Further Reading



Further information can be found in the following documents:

- **STREWS Web-platform security guide: Security Assessment of the Web Ecosystem:** An asset-centric overview of the current state of the security of the Web. Discussion of all major vulnerabilities, countermeasures and best practices.
- **The Tangled Web:** This book covers all aspects of currently relevant security policies, including the numerous browser quirks and tiny differences. The book also gives a good overview of recently introduced technologies (CORS, sandbox, ...)
- **The Death of the Internet:** This book covers a wide range of currently existing attack scenarios, and discusses the technicalities, the attacker's motivations, as well as potential defenses