Analysis of Hypertext Isolation Techniques for Cross-site Scripting Prevention

Mike Ter Louw Prithvi Bisht V.N. Venkatakrishnan

UIC UNIVERSITY OF ILLINOIS AT CHICAGO
Department of Computer Science
COLLEGE OF ENGINEERING

Outline

Quote

"Cross-site scripting (XSS) attacks would always fail if the browser could know for absolute certain which scripts were legitimate and which were malicious."

Gervase Markham
Mozilla Foundation
`http://www.gerv.net/security/content-restrictions/`

Many web users invest a large degree of trust into sites they visit.

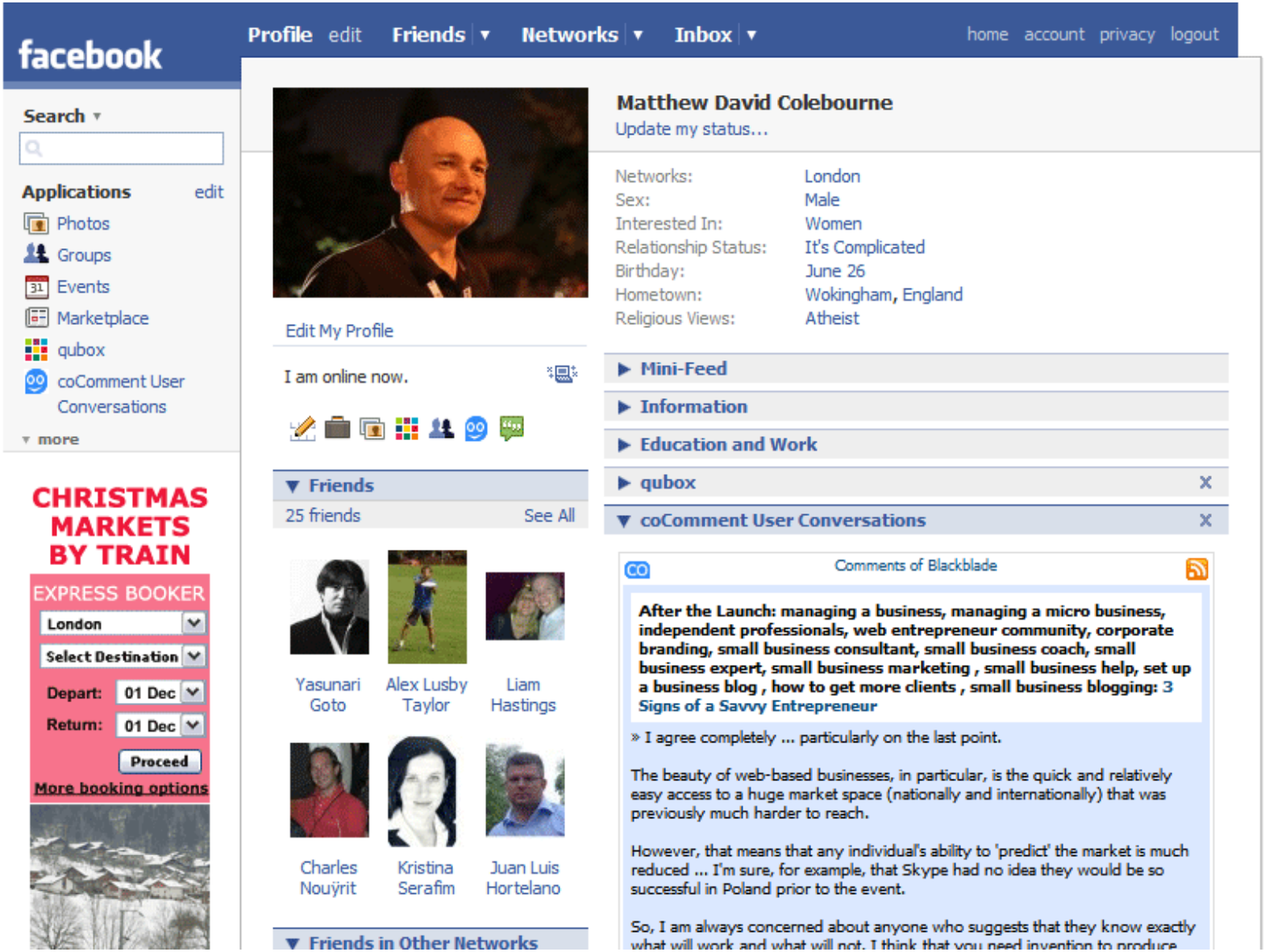
- People expect web applications to:
  - preserve data confidentiality
  - provide accurate information
  - protect authentication credentials
- Lack of assurances forces trust model upon us

Modern websites blend code and data from many sources into a single execution environment.

Content types commonly found in a web application:

- First-party user-generated content
- Second-party web services and information
- Third-party advertisements
- Peer-authored content

Ex: Facebook.com aggregates content from many sources.

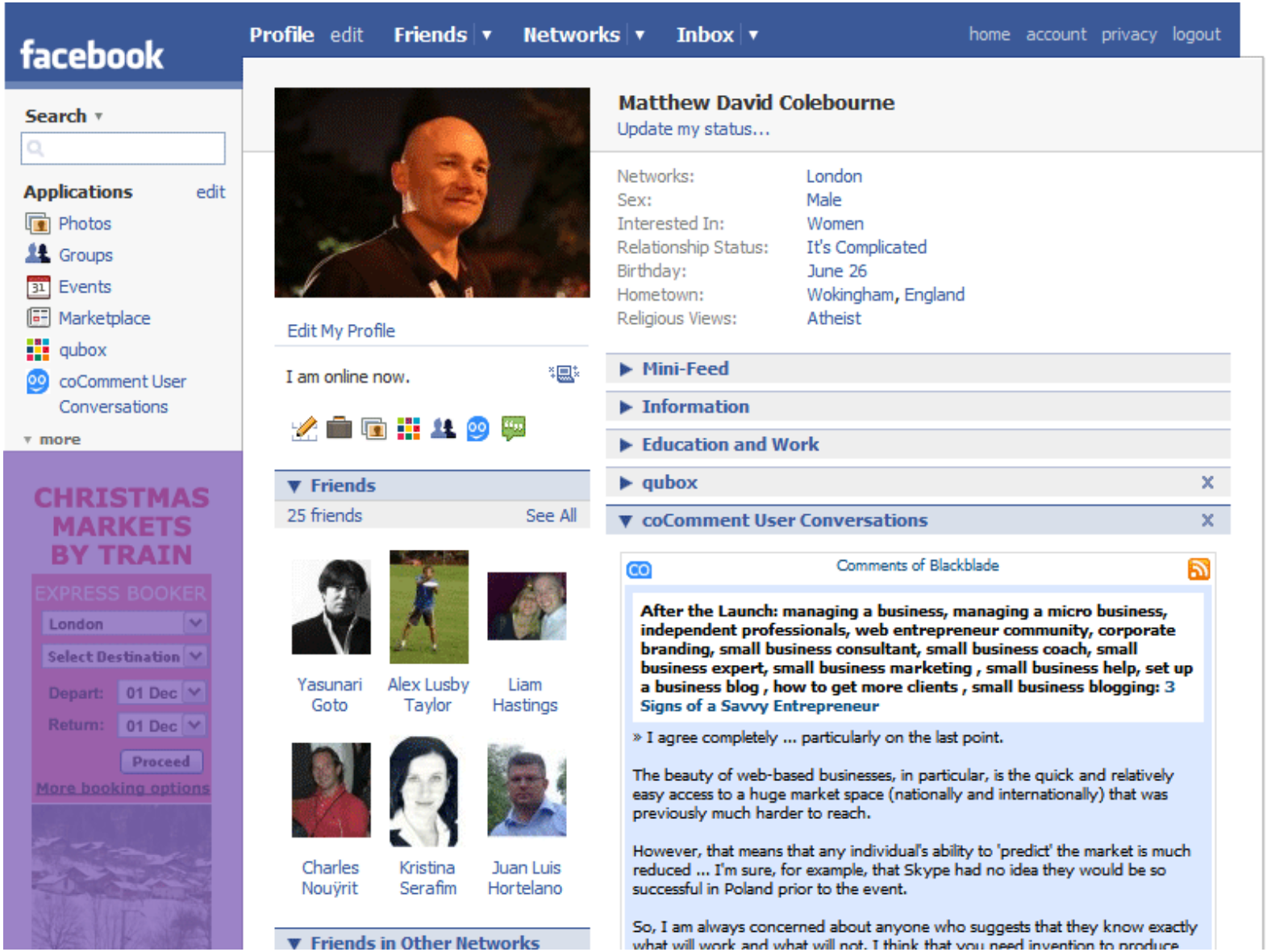


- Ads
- User
- Apps
- Peers
- Facebook

Image from `http://blog.cocomment.com/wp/wp-content/facebook.gif`

UIC

Ex: Facebook.com aggregates content from many sources.



- Ads
- User
- Apps
- Peers
- Facebook

Image from `http://blog.cocomment.com/wp/wp-content/facebook.gif`

UIC

Ex: Facebook.com aggregates content from many sources.

- Ads
- User
- Apps
- Peers
- Facebook

Image from `http://blog.cocomment.com/wp/wp-content/facebook.gif`

UIC

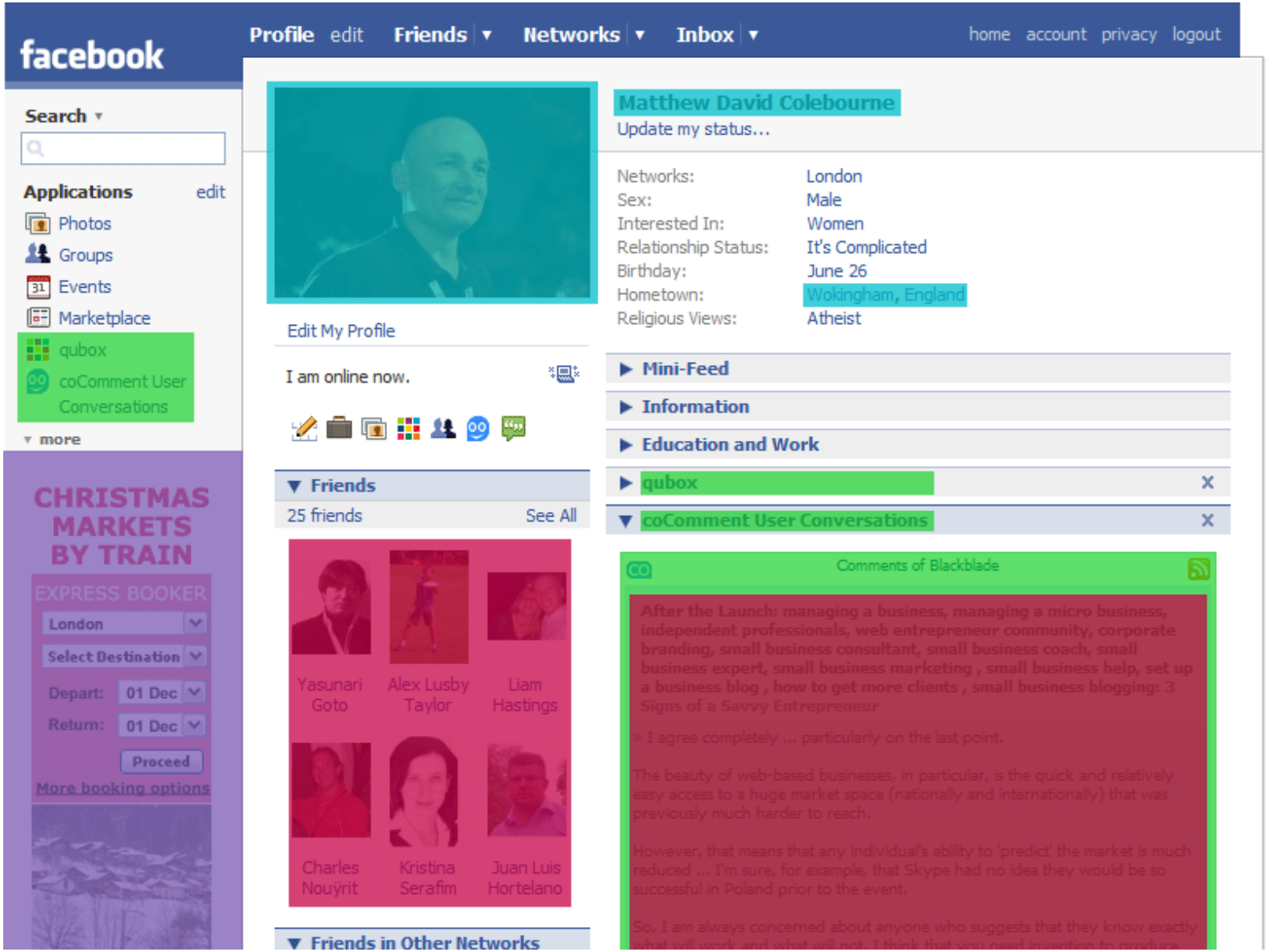
Ex: Facebook.com aggregates content from many sources.

- Ads
- User
- Apps
- Peers
- Facebook

Image from `http://blog.cocomment.com/wp/wp-content/facebook.gif`

UIC

Ex: Facebook.com aggregates content from many sources.

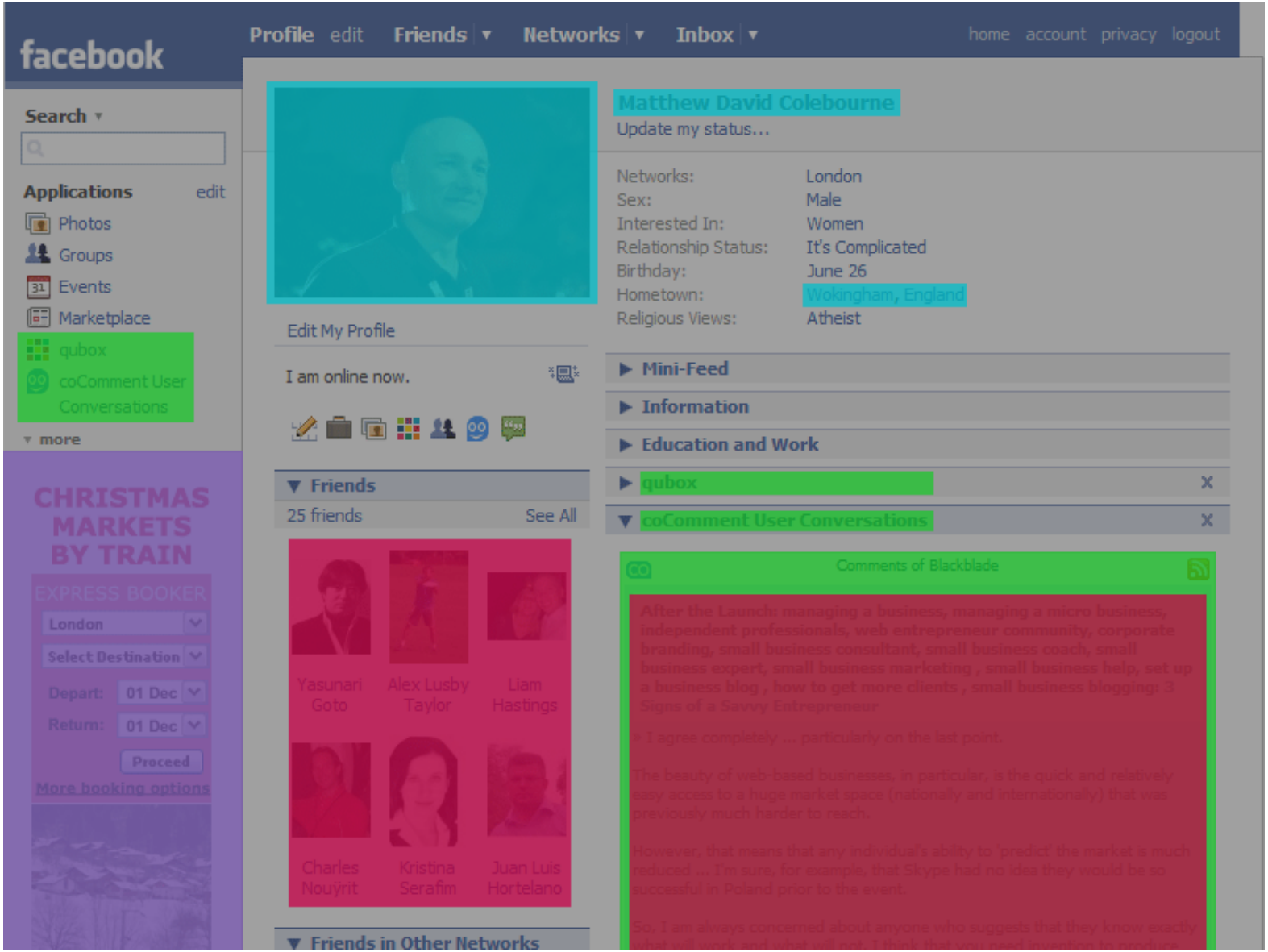


- Ads
- User
- Apps
- Peers
- Facebook

Image from `http://blog.cocomment.com/wp/wp-content/facebook.gif`

UIC

Ex: Facebook.com aggregates content from many sources.



- Ads
- User
- Apps
- Peers
- Facebook

Image from `http://blog.cocomment.com/wp/wp-content/facebook.gif`

UIC

Web application authors endeavor to confine code and data into protection domains.

Important for web app to enforce these constraints:

- Limit capabilities within a protection domain
- Limit inter-domain data flows

Cross-site scripting is a general class of attack to breach domain protection measures.

Abstract view of cross-site scripting (XSS):

1. Attack code input to web application

   Type 0 DOM level zero reflection
   Type 1 Request reflection
   Type 2 Stored/persistent

2. Web app does not sufficiently filter or sanitize input
3. Attack succeeds

   → domain protections breached
   → trust violated

Detection and suppression of malicious web content are challenging tasks for a web app.

- Standard protection measures (i.e., *same-origin policy*) too crude to be useful
  - Web apps must "roll own" fine-grained security policy enforcement mechanism
- Content parsing performed inconsistently across browsers
  - Identification of potentially harmful script code is hard
  - No robust way to distinguish active "code" from passive "data" [Hansen, 2008]

Web apps are better positioned to define policy rules (rather than enforce them).

- App developers have better knowledge of:
  - origins of all emitted content
  - capabilities that are (in)appropriate for outsourced content
  - (un)desirable interactions between protection domains

A key insight is made by Jim, Swamy and Hicks in BEEP.

Observation ([Jim et al., 2007])

To safely embed unknown, untrusted content:

- *Web applications should define policy-based constraints.*
- *Web browsers should enforce these policies.*

Note: We already rely on browser to enforce same-origin policy.

Content restrictions are well justified.

In summary,

- modern web applications integrate content from variety of sources
- level of trust varies by content source and use context
- web apps well suited to define protection domain policies
- browsers best suited to enforce policies

Outline

UIC

Quote

"In talking with the browser companies there seems to be more and more interest in content restrictions.

. . . The obvious answer [is] use an iframe to isolate it. That, unfortunately, has all sorts of user experience issues.

. . . So the best alternative is to create something that tells the browser, 'If you trust me, trust me to tell you to not trust me.' "

Robert Hansen (a.k.a. "RSnake")
Author, XSS Cheat Sheet
`http://ha.ckers.org/blog/20070811/content-restrictions-a-call-for-input/`

How might content restrictions work?

1. Web app breaks document down into logical regions:
   - contents of HTML element
     e.g., `<div>...</div>`
   - value of HTML element attribute
     e.g., `href="..."`
2. Web app declares policy-based constraints per region:
   - Inline
     e.g., `<div policy="...">...</div>`
   - Remote
     e.g., HTTP header targeting region
3. Browser associates policies with regions
4. Browser composes constraints for nested regions
   - Most restrictive constraint applies
5. Browser enforces composite constraints

Policy-based constraints are weak without robust policy targeting.

Sometimes, intended $\neq$ actual policy enforcement region

- Spurious close tags
- Implied "omitted" close tags (i.e., malformed HTML)

Policy-based constraints are weak without robust policy targeting.

Sometimes, intended $\neq$ actual policy enforcement region

- Spurious close tags

  Intended `<div policy="...">``</div><script...>``</div>`

- Implied "omitted" close tags (i.e., malformed HTML)

Policy-based constraints are weak without robust policy targeting.

Sometimes, intended $\neq$ actual policy enforcement region

- Spurious close tags

  Intended `<div policy="...">``</div><script...>``</div>`

  Actual `<div policy="...">␣</div><script...></div>`

- Implied "omitted" close tags (i.e., malformed HTML)

Policy-based constraints are weak without robust policy targeting.

Sometimes, intended $\neq$ actual policy enforcement region

- Spurious close tags

  Intended `<div policy="...">``</div><script...>``</div>`

  Actual `<div policy="...">␣</div><script...></div>`

- Implied "omitted" close tags (i.e., malformed HTML)

  Intended `<table><div policy="...">`
  `</table><script...>``</div></table>`

Policy-based constraints are weak without robust policy targeting.

Sometimes, intended $\neq$ actual policy enforcement region

- Spurious close tags

  Intended `<div policy="...">``</div><script...>``</div>`

  Actual `<div policy="...">␣</div><script...></div>`

- Implied "omitted" close tags (i.e., malformed HTML)

  Intended `<table><div policy="...">`
  `</table><script...>``</div></table>`

  Actual `<table><div policy="...">`
  `␣</table><script...></div></table>`

Content restrictions require isolation of hypertext.

Observation

- *Effective content restriction requires accurate targeting of policy-based constraints to web content regions.*
- *To accurately target, policy declarations must robustly convey the targeted region's precise textual extent to the policy enforcement mechanism.*
  - *We term this need* ***hypertext isolation.***

Outline

UIC

Hypertext isolation mechanisms compromise on several properties.

Ideal hypertext isolation mechanism...

1. increases utility of content restrictions
2. degrades well in today's browsers
3. has no regression from existing methods
4. maximizes usability

Six proposed techniques for hypertext isolation were analyzed. Each fell short in one or more of these areas.

Evaluated techniques fell into six categories.

1. Document separation
   - `<iframe src="..."></iframe>`
2. Request separation
   - `<div src="..."></div>`
3. Response partitioning
   - MIME Multipart/Related (MHTML)
4. Element content encoding
   - `<div src="data:..."></div>`
5. Tag matching
   - `<div tag="`*unique*`">...</div tag="`*unique*`">`
6. Character range encoding
   - `<?isolate src="data:...">`

Transitioning to a new feature requires legacy browser support.

Observed failure modes in non-supported browsers:

- **poor** No content rendered
- **graceful** At *least* trusted document regions are rendered
- **safe** At *most* trusted document regions are rendered
- **best** Trusted fallback content rendered in place of untrusted content

Ideally, hypertext isolation should not make matters worse.

Some evaluated techniques had drawbacks over existing methods.

- Additional rendering delays
  e.g., appending untrusted content
- Additional, intensive HTTP request operations
  e.g., `<div src="...">`
- Unreadable (to humans) hypertext
  e.g., base64 encoding

Hypertext isolation should enable usable content restrictions.

Some interesting applications benefit from the ability to:

1. isolate any type of document region
   - HTML element contents
   - element attribute values
   - JavaScript tokens
2. use same syntax in all contexts (i.e., context-free)
   - Retrofitting existing web apps

Although necessary, hypertext isolation is hard to get right.

In summary,

- Hypertext isolation is required for policy-based capability restriction of web content
- Many outstanding proposals provide isolation
- Isolation techniques differ on key design compromises

Outline

UIC

Conclusion

To conclude,

- It is vitally important to the security of web apps that hypertext isolation be standardized and universally supported.
- Careful compromises should be made to obtain a sound framework for web content restrictions.
- This will help web apps continue to evolve while minding security.

References

Hansen, R. (2008).
XSS cheat sheet.
http://ha.ckers.org/xss.html.
Retrieved on May 22, 2008.

Jim, T., Swamy, N., and Hicks, M. (2007).
Defeating script injection attacks with browser-enforced embedded policies.
In *16th International World Wide Web Conference*, Banff, AB, Canada.

Q & A

Thanks for your attention!

Questions?

HTML standards community takes action!

Ian Hickson, editor for Web Hypertext Application Technology Working Group (WHATWG), yesterday added to HTML 5 proposed standard:

1. `sandbox` attribute of `<iframe>` element
   - Allows specification of (default-deny) policies:
     - `allow-same-origin`
     - `allow-forms`
     - `allow-scripts`
   - `<iframe>` can not open modal dialogs or alerts
   - all plugins disabled within `<iframe>`
   - navigation restrictions
2. `seamless` attribute of `<iframe>` element
   - Layout of `<iframe>` flows seamlessly into surrounding document (similar to `<div>`)
   - CSS style rules cascade into `<iframe>`

www.whatwg.org/specs/web-apps/current-work/#sandbox