The Past, Present and Future of XSS Defense

Jim Manico

2011 OWASP Brussels

Jim Manico

- Managing Partner, Infrared Security
- Web Developer, 15+ Years

- OWASP Connections Committee Chair
- OWASP ESAPI Project Manager
- OWASP Podcast Series Producer/Host

- Kauai/Hawaii Resident with wife Tracey

XSS Defense, Past Exploitable Defenses

- **Input Validation Alone**
  - Sometimes applications needs to support

    < ' " &

    … and other "dangerous" characters
  - Can be very difficult
    - File upload input
    - HTML inputs

- **HTML Entity Encoding Alone**
  - Works well for untrusted data placed in HTML "normal" contexts
  - Does not stop XSS in unquoted HTML attribute and other contexts

XSS Defense Today: Quite Challenging

1. **All untrusted data must first be canonicalized**

   Reduced to simplest form

2. **All untrusted data must be validated**

   Positive Regular Expression Rule

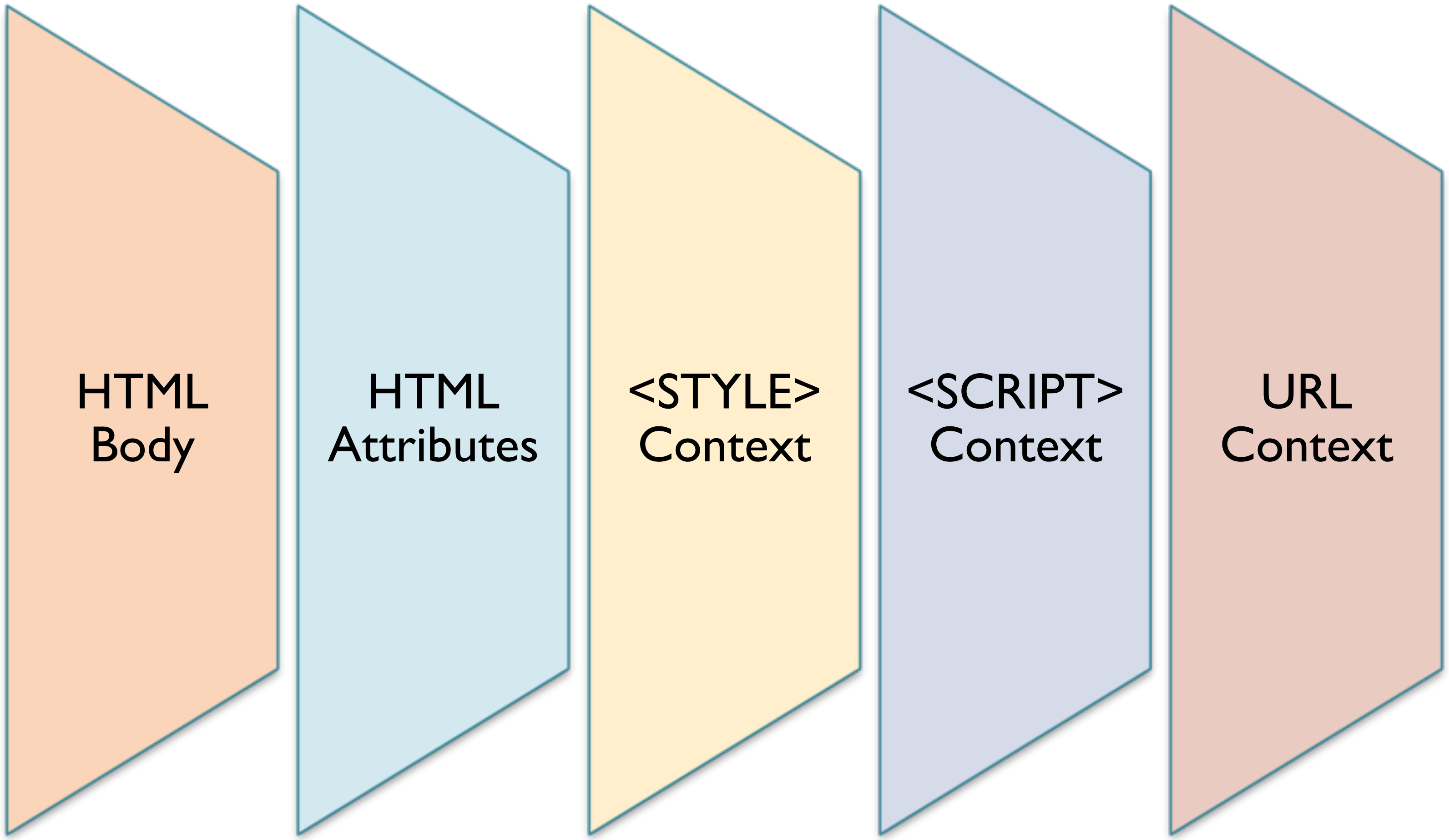
   Blacklist Validation

3. **Untrusted data must be contextually sanitized/encoded**

   - HTML Body
   - HTML Attribute
   - URI Resource Locator
   - Style Tag
   - Event handler
   - Within Script tag

Danger: Multiple Contexts

Browsers have multiple contexts that must be considered!



Danger: DOM Based XSS

- 1. Untrusted data should only be treated as displayable text.
- 2. Always JavaScript encode and delimit untrusted data as quoted strings
- 3. Use document.createElement("…"), element.setAttribute("…","value"), element.appendChild(…), etc. to build dynamic interfaces. Avoid use of HTML rendering methods.
- 4. Understand the dataflow of untrusted data through your JavaScript code. If you do have to use the methods above remember to HTML and then JavaScript encode the untrusted data
- 5. Make sure that any untrusted data passed to eval() methods is delimited with string delimiters and enclosed within a closure or JavaScript encoded to N-levels based on usage, and wrapped in a custom function.
- 6. Limit the usage of dynamic untrusted data to right side operations. And be aware of data which may be passed to the application which look like code (eg. location, eval()). 7. When URL encoding in DOM be aware of character set issues as the character set in JavaScript DOM is not clearly defined.
- 8. Limit access to properties objects when using object[x] accessors
- 9. Don't eval() JSON to convert it to native JavaScript objects. Instead use JSON.toJSON() and JSON.parse()

(1) Auto-Escaping Template Technologies

- **XHP from Facebook**
  - Makes PHP understand XML document fragments similar to what E4X does for ECMAScript
- **Context-Sensitive Auto-Sanitization (CSAS) from Google**
  - Runs during the compilation stage of the Google Closure Templates to add proper sanitization and runtime checks to ensure the correct sanitization.
- **Java XML Templates (JXT) from OWASP**
  - Fast and secure XHTML-compliant context-aware auto-encoding template language that runs on a model similar to JSP.

Copyright 2010 - AppSec Training LLC

Context-aware Auto-escaping Tradeoffs

- **Developers need to write highly compliant templates**
  - No "free and loose" coding like JSP
  - Requires extra time, but increased quality

- **These technologies often do not support complex contexts**
  - Some choose to let developers disable auto-escaping on a case-by-case basis (really bad decision)
  - Some choose to encode wrong (bad decision)
  - Some choose to reject the template (better decision)

(2) Javascript Sandboxing

- **Capabilities JavaScript (CAJA) from Google**
  - Applies an advanced security concept, capabilities, to define a version of JavaScript that can be safer than the sandbox
- **JSReg by Gareth Heyes**
  - Javascript sandbox which converts code using regular expressions
  - The goal is to produce safe Javascript from a untrusted source
- **ECMAScript 5**
  - Object.seal(obj)
    Object.isSealed(obj)
  - Sealing an object prevents other code from deleting, or changing the descriptors of, any of the object's properties

Copyright 2010 - AppSec Training LLC

JSReg: Protecting JavaScript with JavaScript

- **JavaScript re-writing**
  - Parses untrusted HTML and returns trusted HTML
  - Utilizes the browser JS engine and regular expressions
  - No third-party code

- **First layer is an iframe** used as a safe throw away box
- **The entire JavaScript objects/properties list was whitelisted** by forcing all methods to use suffix/prefix of "**$**"
- **Each variable assignment was then localized** using var to force local variables
- Each object was also checked to ensure it didn't contain a window reference

Google CAJA: Subset of JavaScript

- Caja sanitizes JavaScript into *Cajoled* JavaScript

- Caja uses multiple sanitization techniques
  - Caja uses STATIC ANALYSIS when it can
  - Caja modifies JavaScript to include additional run-time checks for additional defense

Copyright 2010 - AppSec Training LLC

CAJA workflow

- The web app loads the Caja runtime library, which is written in JavaScript
- All un-trusted scripts must be provided as Caja source code, to be statically verified and cajoled by the Caja sanitizer
- The sanitizer's output is either included directly in the containing web page or loaded by the Caja runtime engine

CAJA Compliant Applications

- **A Caja-compliant JavaScript program is** one which
    - is statically accepted by the Caja sanitizer
    - does not provoke Caja-induced failures when run cajoled

- **Such a program should have the same semantics whether run *cajoled* or not**

Copyright 2010 - AppSec Training LLC

#@$(This

- **Most of Caja's complexity is needed to defend against JavaScript's rules regarding the binding of "this".**
- **JavaScript's rules for binding "this" depends** on whether a function is invoked
  - by construction
  - by method call
  - by function call
  - or by reflection
- If a function written to be called in one way is instead called in another way, its **"this" might be rebound to a different object** or even to the global environment.

Copyright 2010 - AppSec Training LLC

(3) Browser Protections

- **Content Security Policy**
  - JavaScript policy standard

- **Reflective Defense XSS in Chrome**
- **IE 8 Cross-Site Scripting Filter**
  - Blacklist browser-based URL filters
  - Early versions of IE8's browser-based filter actually made XSS possible on sites that did not even have XSS vulns due to errors in MS's filter

Copyright 2010 - AppSec Training LLC

Awesomeness: Content Security Policy

- **Externalize all JavaScript within web pages**
  - No inline script tag
  - No inline JavaScript for onclick or other handling events
  - Push all JavaScript to formal .js files using event binding

- **Define the policy for your site** and whitelist the allowed domains where the externalized JavaScript is located

- **Add the X-Content-Security-Policy response header** to instruct the browser that CSP is in use

- **Will take 3-5 years** for wide adoption and support

THANK YOU!

jim@owasp.org

2011 OWASP Brussels