Ghosts of XSS Past, Present and Future.

Jim Manico

September 22, 2011

Jim Manico

- VP Security Architecture, WhiteHat Security
- Managing Partner, Infrared Security
- Web Developer, 15+ Years

- OWASP Connections Committee Chair
- OWASP Podcast Series Producer/Host
- OWASP Cheat-Sheet Series Manager

XSS: Why so serious?

- Session hijacking
- Site defacement
- Network scanning
- Undermining CSRF defenses
- Site redirection/phishing
- Load of remotely hosted scripts
- Data theft
- Keystroke logging
- Getting Stallowned

Past XSS Defensive Strategies

- 1990's style XSS prevention
  - Eliminate < , > , & , " , ' characters?
  - Eliminate all special characters?
  - Disallow user input?
  - Global filter?

- Why won't these strategies work?

XSS Defense, 1990's

Data Type	Defense
Any Data	Input Validation

#absolute-total-fail

Past XSS Defensive Strategies

- Y2K style XSS prevention
  - HTML Entity Encoding
  - Replace characters with their 'HTML Entity' equivalent
  - Example: replace the "<" character with "<"

- Why won't this strategy work?

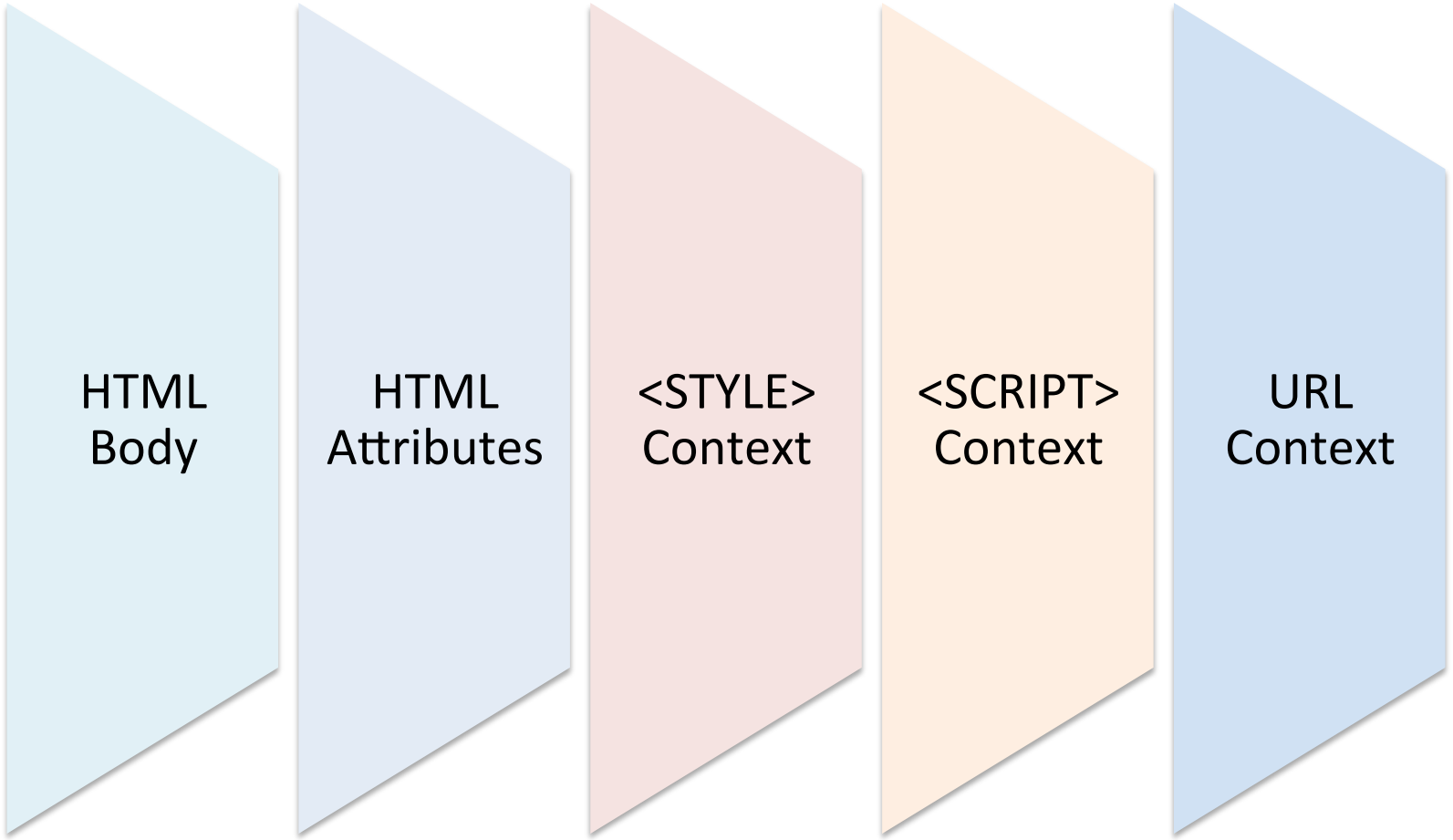
XSS Defense, 2000

Data Type	Defense
Any Data	HTML Entity Encoding

Why won't this strategy work?

Danger: Multiple Contexts

Browsers have multiple contexts that must be considered!



Past XSS Defensive Strategies

1. All untrusted data must first be canonicalized
   - Reduced to simplest form
2. All untrusted data must be validated
   - Positive Regular Expressions
   - Blacklist Validation
3. All untrusted data must be contextually encoded
   - HTML Body
   - Quoted HTML Attribute
   - Unquoted HTML Attribute
   - Untrusted URL
   - Untrusted GET parameter
   - CSS style value
   - JavaScript variable assignment

XSS Defense, 2007

Context	Defense
HTML Body	HTML Entity Encoding
HTML Attribute	HTML Attribute Encoding
JavaScript variable assignment JavaScript function parameter	JavaScript Hex Encoding
CSS Value	CSS Hex Encoding
GET Parameter	URL Encoding
Untrusted URL	HTML Attribute Encoding
Untrusted HTML	HTML Validation (Jsoup, AntiSamy)

Why won't this strategy work?

ESAPI CSS Encoder Pwnd

From: Abe [mailto:abek1 at sbcglobal.net]

Sent: Thursday, February 12, 2009 3:56 AM

Subject: RE: ESAPI and CSS vulnerability/problem

I got some bad news

CSS Pwnage Test Case

<div style="width: <%=temp3%>;"> Mouse over

temp3 = ESAPI.encoder().encodeForCSS("expression(alert(String.fromCharCode(88,88,88)))");

 Mouse over

Pops in at least IE6 and IE7.

lists.owasp.org/pipermail/owasp-esapi/2009-February/000405.html

Simplified DOM Based XSS Defense

1. Initial loaded page should only be static content.

2. Load JSON data via AJAX.

3. Only use the following methods to populate the DOM
   - Node.textContent
   - document.createTextNode
   - Element.setAttribute

References: http://www.educatedguesswork.org/2011/08/guest_post_adam_barth_on_three.html and Abe Kang

Dom XSS Oversimplification Danger

Element.setAttribute is one of the most dangerous JS methods

If the first element to setAttribute is any of the JavaScript event handlers or a URL context based attribute ("src", "href", "backgroundImage", "backgound", etc.) then pop.

References: http://www.educatedguesswork.org/2011/08/guest_post_adam_barth_on_three.html and Abe Kang

DOM Based XSS Defense

1. Untrusted data should only be treated as displayable text.
2. JavaScript encode and delimit untrusted data as quoted strings
3. Use document.createElement("…"), element.setAttribute("…","value"), element.appendChild(…), etc. to build dynamic interfaces.
4. Avoid use of HTML rendering methods.
5. 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
6. 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.
7. 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()).
8. When URL encoding in DOM be aware of character set issues as the character set in JavaScript DOM is not clearly defined.
9. Limit access to properties objects when using object[x] accessors
10. Don't eval() JSON to convert it to native JavaScript objects. Instead use JSON.toJSON() and JSON.parse()
11. We are just getting started. See https://www.owasp.org/index.php/DOM_based_XSS_Prevention_Cheat_Sheet

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

JSReg: Protecting JS with JS

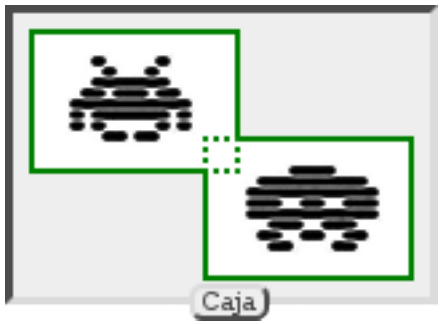
- **JavaScript re-writing**
  - Parses untrusted HTML and returns trusted HTML
  - Utilizes the browser JS engine and regular expressions
  - No third-party code

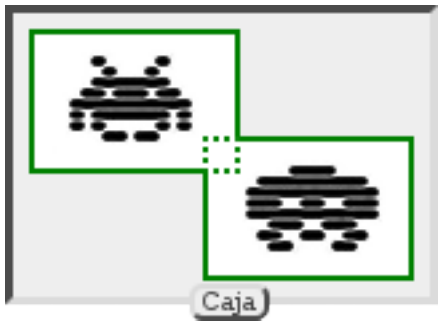
- **First layer is an iframe** used as a safe throw away box
- **The entire JavaScript objects/properties list was whitelisted** by forcing all methods to use suffix/prefix of "$"
- **Each variable assignment was then localized** using var to force local variables
- Each object was also checked to ensure 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





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 JavaScript

- **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**

#@$(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.

XSS Defense, Today

Data Type	Context	Defense
Numeric, Type safe language	Doesn't Matter	Cast to Numeric
String	HTML Body	HTML Entity Encode
String	HTML Attribute, quoted	Minimal Attribute Encoding
String	HTML Attribute, unquoted	Maximum Attribute Encoding
String	GET Parameter	URL Encoding
String	Untrusted URL	URL Validation, avoid javascript: URL's, Attribute encoding, safe URL verification
String	CSS	Strict structural validation, CSS Hex encoding, good design
HTML	HTML Body	HTML Validation (JSoup, AntiSamy, HTML Sanitizer)
Any	DOM	DOM XSS Cheat sheet
Untrusted JavaScript	Any	Sandboxing
JSON	Client parse time	JSON.parse() or json2.js

Got future?

Context Aware Auto-Escaping

- **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 by Jeff Ichnowski**
  - Fast and secure XHTML-compliant context-aware auto-encoding template language that runs on a model similar to JSP.

- **Apache Velocity Auto-Escaping by Ivan Ristic**
  - Fast and secure XHTML-compliant context-aware auto-encoding template language that runs on a model similar to JSP.

Auto Escaping Tradeoffs

- **Developers need to write highly compliant templates**
  - No "free and loose" coding like JSP
  - Requires extra time but increases quality
- **These technologies often do not support complex contexts**
  - Some are not context aware (really really bad)
  - Some choose to let developers disable auto-escaping on a case-by-case basis (really bad)
  - Some choose to encode wrong (bad)
  - Some choose to reject the template (better)

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

WhiteHat SECURITY

THANK YOU!

Gaz Heyes
Abe Kang
Mike Samuel
Jeff Ichnowski
Adam Barth
Jeff Williams
Kotowicz
many many others…

jim@owasp.org