The case for client-side HTML sanitization

Insane in the IFRAME

David Ross
Principal Software Security Engineer
Trustworthy Computing Security
Microsoft
Who am I?

- MSRC Engineering team member
  - Up in your vuln reports, helping to remediate

You may know me from...

- Bug bounty triage v-team lead for design-level issues
- IE XSS Filter
- Internet Explorer security
- Twitter (@randomdross)
Agenda

• HTML Sanitization – An Introduction
• The many bugs
• What’s the lesson?
• Is there a better mousetrap?
• jSanity
  • Design principles
  • Implementation
    • Security
    • Compatibility
    • Performance
  • Demo
  • Current status
• The future
• Credits / Conclusion / Q&A
HTML Sanitization – An Introduction

What is HTML sanitization?

• HTML sanitization takes markup as input, outputs “safe” markup / DOM
  • ...for some definition of “safe”
  • Distinct from encoding (eg: URLEncoding, HTMLEncoding, etc.)

• HTML sanitization is everywhere
  • Canonical example: web forum posts w/markedup
  • Modern applications with client-side business logic
    • Eg: Javascript-based Windows 8 Store apps
    • Outlook.com
HTML Sanitization – An Introduction

Why are HTML sanitization bypass bugs important?

- Worst case scenario
  - Script running from a mail message executes within the security context of the mail application
  - ...from the preview pane that appears automatically
  - Attacker could set up auto-forwarding, impersonate you, steal all your mail, etc.

Yet, HTML sanitization bugs are pervasive

- Fuzzing? Can be helpful, but difficult*...
- Threat modeling? Not so relevant...
- Smart hackers with some free time – very relevant

And the underlying platforms continue to change. All of them.

This is a hard problem.
The Many Bugs – Example #1

- Sanitizer Bypass in validator Node.js Module by @NealPoole (https://t.co/5omk5ec2UD)
  - Nesting

  **Input:** `<scrRedirecRedirect 302t 302ipt type="text/javascript">prompt(1);</scrRedirecRedirect 302t 302ipt>`
  
  **Output:** `<script type="text/javascript">prompt(1);</script>`

Observation: Removing data from markup can create XSS where it didn’t previously exist
The Many Bugs – Example #2

- CVE-2011-1252 / MS11-074
  - SharePoint / SafeHTML

Input:
```html
<div style="color:rgb(0,0,0)\n\n\&=expression(alert(1))"></div>
```

Output:
```html
<div style="color:rgb(0,0,0)\n\n\&=expression(alert(1))"></div>
```

Observations:
- Sanitizer created a delimiter (the semi-colon)
- Legacy IE CSS expression syntax required to execute script
- Context mismatch
  - Sanitizer: “expression“ is considered to be in a benign location
  - Browser: “expression“ is considered to be the RHS of a CSS property set operation
The Many Bugs – Example #3

• Codelgniter <= 2.1.1 xss_clean() sanitizer bug (CVE-2012-1915)
  • Credit: Krzysztof Kotowicz (@kkotowicz)
    • Thx @NealPoole and @adam_baldwin!

• Fake tag closure

  Input and Output:

  <img src="" onerror=alert(1)>
  <button/a="" autofocus onfocus=alert&#40;1;&#40;40;></button>

Observations:

• No content modification required to trigger the vulnerability
• Sanitizer: “>” recognized as closing the IMG or BUTTON element
  • Allows script through, believing it to be in the raw HTML context
• Browser: “>” recognized as an attribute value
  • Browser executes onerror / onfocus event handlers
The Many Bugs – Example #4

• Wordpress 3.0.3 (kses.php)
  • Credit: Mauro Gentile (@sneak)
    • Thx @superevr!

• Lower case attribute name check

  Input and Output:
  
  <a HREF="javascript:alert(0)">click me</a>

Observations:
• No content modification required to trigger the vulnerability
• Sanitizer: Only lower case “href” recognized as an attribute name
• Browser: HREF attribute recognized, javascript: URL executes on click
• Sanitizer and browser don’t agree on what constitutes an attribute name
What’s the lesson?

1. High level: Parsing / “context management” is **hard**

   Sanitizers must *exactly* emulate client-side parsing
   - An opportunity for a vulnerability any time the sanitizer and browser get out of sync
   - Sanitizer output must be safe for *all* useragents
     - If this is even possible!
What’s the lesson? (cont.)

2. Parsing is the **difficult** part of sanitization

   • The “business logic” is easy
     • Eg: What tags, attributes, CSS, etc. are considered acceptable

   • Logically, the sanitizer was built to define this business logic
     • The sanitizer’s value is not derived from being yet another HTML parser!

   • All the bugs identified previously are parsing / “context management” related
Is there a better mousetrap?

- Why not use the browser’s own parsing!
  - ...the browser can never be “wrong”
    - No opportunity for a parsing mismatch

- Implication: Sanitization is client-side
  - So is jQuery!
Introducing... jSanity

• An experiment in creating a better kind of sanitizer API
• Security advantages
  • No independent parsing / context handling
    • Thus fundamentally invulnerable to the most prevalent and most difficult to address sanitization vulnerabilities
    • No regular expressions!
• Pervasive allow-list strategy (no block-listing)
• Doesn’t trigger mutation-based XSS (mXSS)
  • HTML → DOM only, never HTML → DOM → HTML → DOM
• Configurable to provide the right level of sanitization for a given scenario
  • Eg: Allow data-foo attributes, otherwise block data-*
• Simple
  • Straightforward, explicitly defined ruleset for “business logic”
    • Isolated within the code, not strewn across a large codebase
    • So much tricky sanitization / parsing logic is unnecessary!
jSanity Design principles

• Secure by Default
  • Against injection of script
  • Against **everything else** (configurable)
    • CSS-based overlay attacks
    • IFRAME or FORM injection
    • External content download (privacy for mail scenarios)
    • Namespace squatting
    • Unclosed tags / attributes, etc.

• Compatible
  • Cross-browser
  • IE9+, at minimum

• Performant
  • Competitive with other techniques
jSanity Implementation

• Basic facts
  • jQuery plugin
  • ~1K LoC, currently

• Sanitization process
  1. Use `document.implementation.createHTMLDocument` to parse input into a DOM

<HTML><BODY>
  <P></P>
  <SCRIPT></SCRIPT>
  <SPAN></SPAN>
</BODY></HTML>
jSanity Implementation (cont.)

• Sanitization process (cont.)
  2. Use `document.createTreeWalker` to walk the DOM
jSanity Implementation (cont.)

• Sanitization process (cont.)
  3. Remove elements / attributes / etc. not explicitly allowed*

*Old (less-performant) approach: Build yet another DOM by copying safe elements / attributes / etc. to a new DOM during tree walk
jSanity Implementation (cont.)

• Sanitization process (cont.)
  4. Finalized DOM is attached into the page DOM under a specified element
jSanity Implementation (cont.)

- Sanitization process (cont.)
  5. Resolve jQuery Promise to notify hosting page
jSanity Security

- `document.implementation.createHTMLDocument`
  - **Must never run script**
  - Must never pull external content
  - Just parse and build a DOM, that’s it!
  - Implementation bugs in this API should be straightforward to fix

- Additional security properties
  - Allow-lists elements, attributes, CSS, etc.
    - Namespace support (eg: SVG)
  - Prefixes NAME / ID attributes to avoid squatting
    - CLASS attributes must be optionally handled by a callback
  - Constrains display area of output to mitigate overlay attacks
jSanity Security (cont.)

• Additional security properties (cont.)
  • Validates link URLs to avoid script injection, etc.
    • Callbacks allow for custom validation / translation
    • Callers can specify script that runs onClick
      • Eg: “Do you really want to navigate to this link?”
  • Allows for regulation of external content
  • Conservative ruleset by default
  • Validates supported / secure environment
    • Browser version, docmode, etc.
jSanity Security (cont.)

• Bugs found in pentest (Credit: Gareth Heyes)
  • CLASS attributes can contain multiple class names
    • ...so simple prefixing doesn’t work as it does with NAME / ID
    • INPUT: `<a name="x" class="x y">test</a>`
    • OUTPUT: `<a name="jSanity_targetDiv_x_y" class="jSanity_targetDiv_x y">test</a>`
  • DoS conditions
    • `<math>` bomb
      • `<math>` element doesn’t contain a style property
    • `<div data="A">`
      • Causes `elementNode.setAttribute(name, value)` to explode
      • Fixed with a try/catch
    • DOM squatting preventing node removal
      • Eg: `foo<form><input name="parentNode">`
      • Confused node removal code, fortunately failed secure
  • Various other implementation bugs
jSanity Compatibility

• Cross-platform
• Based on javascript
  • A javascript-based sanitizer is easily auditable (vs. a “black box”)
  • HTML updated frequently enough that sanitizers are often under pressure to add support for new elements, attributes, etc.
    • Simple and extensible for customers
      • HTML elements, attributes, etc. may be...
        • Allowed by default
        • Allowed on an opt-in basis
        • Allowed via custom callbacks
• Drawback: Requires browser primitives
  • No support prior to IE9
jSanity Performance

• Async (vs. synchronous toStaticHTML API)
  • Leverages jQuery promises / deferreds
  • setTimeout
    • Uses NobleJS setTimeout polyfill where setTimeout isn’t available
    • Work is performed with minimal blocking of other script on the page
• Sorry, no multithreading
  • Web Workers can’t touch the DOM
• Competition
  • Server-side sanitizers
    • Zero client-side perf impact
    • Results of jSanity sanitization may be pre-computed and cached
  • .toStaticHTML() API
jSanity Performance - Benchmarks

• Sanitization of ~1MB markup from the front page of 10 major sites

• Visual output
  • Dot (.) == 10ms heartbeat
  • X == ~5 skipped heartbeats
  • # == “All Done”

• Where we started
  • XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX#...........
  • ☹ Slow synchronous return, lots of UI non-responsiveness

• Where we wound up after going async, and perf improvements
  • XXXX..XX.XX.....XX.XX..#.........
  • ☑ Immediate synchronous return, opportunity for script on the page to execute alongside sanitization
  • “All Done” (including rendering) in ~650ms on a standard desktop w/IE10
jSanity Performance – Benchmarks (cont.)

• The competition
  • .toStaticHTML
    • No DOM insertion
      • XXX..#
      • Fast ~140ms to “All Done”
      • At least jSanity still wins on synchronous response 😊

• With DOM insertion
  • XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    • Long freeze
      • Due to STYLE elements passing through
      • Liberal sanitization allows CSS overlay
jSanity Performance – Benchmarks (cont.)

• Conclusions
  • Raw toStaticHTML is tough to beat
    • ...but we win on synchronous response time
    • jSanity is fast enough that rendering speed dictates overall sanitizer performance
      • We will edge closer to toStaticHTML’s rendering time as more safe CSS is allowed through sanitization
      • Still, jSanity will never be as liberal as toStaticHTML
        • More secure
        • Now also demonstrated to aid performance
  • Pervasive use of Deferred w/setImmediate is proven to maintain a responsive UI
Options precedence / inheritance rules:
(Options specified on target element) > (options specified on sanitize() call) > (default options)
jSanity Status

• Feature complete
• Certainly not “done”
  • Needs real-world testing
• Challenges
  • We need a better solution for <STYLE> elements
    • Today we call back into host to handle <STYLE> elements
    • May not be possible to solve this today without custom parsing logic
The future

- Investigating how to make jSanity broadly available
- But… If you can’t wait, feel free to build your own!
  - Remember: Building your own sanitizer can be harder than it looks!

- Framework integration
- Transparently override default sanitization in platforms that are based on web technology
The future

- Where do these new technologies fit into the picture...

  - Shadow DOM
    - Isolation within a single DOM
    - Something jSanity can leverage in the future

  - Seamless frames + SANDBOX + SRCDOC
    - Solves some of the biggest problems with IFRAME-based isolation
      - CSS bleed-through
      - Non-rectangular shape
    - Potential disadvantages
      - Limited configurability
      - Limited browser support (today)
      - Agility in the event of a bypass, globally or due to an application artifact
Credits

• jSanity is simply an API that leverages client-side sanitization techniques pioneered by a few amazing individuals
  • Mario Heiderich (@0x6D6172696F)
    • JSAgents / IceShield
  • Gareth Heyes (@garethheyes)
    • JSLR

• Special thanks to...
  • Ben Livshits
  • Loris D’Antoni
    • Used jSanity in demonstrating FAST domain specific language for tree manipulation

• Other shout-outs
  • Google Caja HTML sanitizer
    • Client side, parses HTML
  • Stefano Di Paola, Eduardo ‘Sirdarckcat‘ Vela N.
Conclusion

• HTML sanitizers as they exist today have recurring, unnecessary bypass vulnerabilities due to parsing / “context management”

• It is possible to build a client-side sanitizer offering a fundamental security advantage relative to server-side approaches
  • Based on browser primitives that exist today
  • Acceptable compatibility and performance characteristics

• Not-so-bold prediction
  • In five years, the majority of top tier applications and frameworks will support client-side sanitization based on browser primitives
I just presented on HTML sanitization at OWASP AppSec EU 2013. AMA!

Slides will be up on Slideshare.net shortly...