Cross Site Scripting (XSS) and PHP Security

Anthony Ferrara

NYPHP and OWASP Security Series

June 30, 2011

What Is Cross Site Scripting?

Injecting Scripts

Into

Otherwise Benign and Trusted

Browser Rendered Content

Example XSS

```
$html = '<script>alert("hi");</script>';
echo "<b>$html</b>";
```

```
<b><script>alert("hi");</script></b>
```

Two Types Of XSS

- Transient XSS
  - Passes Data Through Request Data
  - Affects Only The Tainted Request(s)
- Persistent XSS
  - Stores Data On The Server
  - Affects All Requesters (visitors)

Before We Talk About XSS, We Need To Talk About: Filter In, Escape Out

What Is “Filter In”?

- Can have 2 meanings based on context
  - Removing or stripping unsafe/invalid content
  - Rejecting unsafe/invalid content
- All input should be filtered
  - Would you accept “2f” as an age?
- What is “input”?
  - Anything that is not hard coded into your code
  - This includes your database…

Wait, Does That Mean I Need To Filter Everything Twice?

Yes.

What Is “Escape Out”?

- Escaping means to make content “safe” for a context
- All output **must** be escaped
  - How it should be escaped depends upon context
    - SQL requires different techniques than HTML
  - It should be escaped as close to output as possible
- What is “Output”?
  - Anything that leaves the memory of the program
    - SQL, Files, HTML, REST, Headers, XML, JSON, etc

Wait, Does That Mean I Need To Escape Everything Multiple Times?

Yes.

What Are The Parts Of HTML?

- Nodes: <a> ← (the “a”)
- Values: foo ← (the “foo”)
- Attribute Names: <c d=“e”/> ← (the “d”)
- Attribute Values: <f g=“h” /> ← (the “h”)
- CSS Identifiers: .foo {} ← (the “.foo”)
- CSS Literals: .foo {color:”bar”} ← (the ”bar”)
- JS Code: alert(‘g’) ← (the “alert”)
- JS Literals: alert(‘h’) ← (the “h”)
- HTML Comments: <!– bar → ← (the “bar”)

Let’s Talk About Escaping First

Never Allow Unfiltered User Input :

- Node Names
  - <**foo** />
- Attribute Names
  - <a **bar**=“” />
- HTML Comments
  - <!– **Foo** →
- CSS Identifiers
  - .baz{foo}
- JS Code
  - **biz**();

You Cannot Escape Content For Those HTML Components!

Values

- <foo>bar</foo>
- Need To Escape The Following Characters:
  - & -> &
  - “ -> "
  - < -> &lgt;
  - > -> >
  - ‘ -> '
- Prevents Injection of New Tags

Attribute Values

- Always quote the attribute value
  - <foo bar=“baz” />
- Need To Escape The Following Characters:
  - & -> &
  - “ -> "
  - < -> &lgt;
  - > -> >
  - ‘ -> '

JS Literals

- Always quote string literals
- Always cast numeric literals
- Need To Escape The Following Characters:
  - All Non-Alpha Numeric Characters
    - Use \xNN format
- Be Aware That Not All Literals Can Be Escaped
  - setInterval('foo') ← "Foo" should never be unfiltered

Tools Available In PHP For Escaping

htmlspecialchars()

- Useful for escaping Values and Attribute Values
- Should always pass “ENT_QUOTES” flag
- Should always set the character set

```
htmlspecialchars($input, ENT_QUOTES, “UTF-8”)
```

preg_replace_callback()

- Useful for escaping JS literals

```
preg_replace_callback(
        '/[^a-z0-9]/i',
        function ($match) {
                $chr = dechex(ord($match[0]));
                return '\\x'.
                        str_pad($dechex, 2, '0', STR_PAD_LEFT);
        },
        $data
);
```

OWASP's ESAPI

- Useful for all HTML escaping needs
- Has multiple methods for escaping
- https://www.owasp.org/index.php/ESAPI

```
$encoder->encodeForHTML($data);
$encoder->encodeForHTMLAttribute($data);
$encoder->encodeForJavaScript($data);
```

Smarty

- Templating Engine for PHP
- Does not escape anything by default!
  - Cannot be told to do so
- Must explicitly use special syntax to escape

{$var|html}

Twig

- Templating Engine for PHP
  - Similar to Smarty, but cleaner and more powerful
- Does Intelligent Escaping Automatically
- Can be turned off as needed

{{ var }}

Let’s Talk Filtering

Filtering Guidelines

- Always Favor White-listing over Black-listing
  - Filtering against valid values is more robust
- Always do it for all input
  - Including Content From The Database!
  - Allows changes to the filter to propagate automatically
- Identify Improper Input and Notify The User
  - Gives User a Chance To Fix The Issue
  - Also gives immediate feedback to an attacker

What Can You Safely Filter?

- All User Supplied Data
- Any part of HTML, if filtered properly, can be supplied by user input

- Be Careful When Filtering Sensitive Elements:
  - URLs
  - JavaScript Code
  - HTML Content

Filtering HTML

- Check For Improper Tag Structure
  - <a>
- Check For “Bad” Tags:
  - style, script, comments, etc
- Check For “Special” Attributes
  - href, src, js events, style, etc
  - Make sure they are valid, and not JS (or remove them entirely)

Tools Available In PHP For Filtering

strip_tags()

- Removes all tags except those explicitly allowed
- Removes all attributes
  - Not effective if you need links, etc
- Removes everything that is wrapped by < >
  - May break user's intent

strip_tags($data, '<b><u><i>');

HTMLPurifier

- Library to sanitize HTML
- Very Smart
  - Cleans up document structure
  - Allows safe attributes
  - Highly configurable
- http://htmlpurifier.org/

```
$purifier->purify($data);
```

Don’t Roll Your Own!

HTML Sanitization Is Not A Trivial Problem To Solve

The XssBadWebApp

- Designed To Be A “Real World” Application
- Several Known XSS Vulnerabilties
  - No known non-xss vulnerabilities
- Designed For Educational Use Only
- Released under the BSD License
- Available At GitHub
  - github.com/ircmaxell/XssBadWebApp

Demonstration Time!

Quick Review

- There Is No “Magic” Solution
- Always Filter Input
  - Even When “Input” Comes From The Database
- Always Escape Output
  - Escaping Is Context Dependent
- Several Tools Are Available
  - Use Them!

Questions?

Comments?

Snide Remarks?

Anthony Ferrara

Anthony.Ferrara@nbcuni.com
ircmaxell@php.net

blog.ircmaxell.com
@ircmaxell