The Case of the Promiscuous Parameters and Other Ongoing Mysteries in Web Security

Jacob West
Manager, Security Research
jacob@fortify.com
Evolution

(or is it?)
Today's Evolutionary Mysteries

I. Promiscuous Parameters
   - register_globals
   - Struts 2, Spring MVC

II. Wanton Web Data
   - Cross-Site Scripting
   - JavaScript Hijacking

III. Licentious Listeners
   - gets(), operator >>
   - readLine()

IV. Deleterious Delvers
   - addslashes()
   - <c:out/>
     .NET Validation Framework

V. Learning from past mistakes: A look at the future of web security
MYSTERY I

PROMISCUOUS PARAMETERS
Promiscuous Parameters:
When a platform or framework makes it too easy to access HTTP request parameters, it blurs the line between trusted and untrusted data and can leave itself open to attack.
PGP register_globals

- Injects request parameters as global variables
- Most infamous "feature" that led to PHP's bad security mojo
- "The" input paradigm until PHP 4.2.0
- Deprecated in PHP 5.3.0; Removed in 6.0.0
Example: `register_globals`

- Attackers can override existing server-side variables

```php
<?php
// $authorized = true for authenticated users
if (authenticated_user()) {
    $authorized = true;
}

// GET auth.php?authorized=1
if ($authorized) {
    include "/highly/sensitive/data.php";
}
?>
```
Fast Forward to 2008

• Struts 2
  – Introduces flexible POJO beans and actions
  – Automatically maps request parameters to POJO fields

• Spring MVC
  – Provides similarly "friendly" functionality for mapping request parameters in to command beans
Example: Struts 2 Action

```java
public String execute() throws Exception {
    if (isInvalid(username)) return INPUT;
    if (isInvalid(password)) return INPUT;
    authenticated = true;
    return SUCCESS;
}

private String username;
private String password;
private boolean authenticated;

public String getUsername() {
    return username;
}
public void setUsername(String username) {
    this.username = username;
}
...
```
Promiscuous Parameters

• Antipattern: Blurring trust boundaries and accepting input quietly
• Same old "convenience" over "security" tradeoff
• Old or new, this mysterious vulnerability
  – Blurs the line between trusted and untrusted
  – Makes input validation hard to apply consistently
  – Handicaps automated approaches to verifying security
• Represents a regression
  – Globally for Java
  – Specifically for Struts (from 1 to 2)
• Disappointing—frameworks should improve security, not hinder it
MYSTERY II

WANTON WEB DATA
**Wanton Web Data**: Nothing good comes from mixing code and data; the two mixing is at the root of many serious security vulnerabilities and yet we insist on repeating the mistakes of our past again and again.

The Dark Side of Ajax
Example: Buffer Overflow

```c
void trouble() {
    int a = 32;     /*integer*/
    char line[128]; /*character array*/
    gets(line);     /*read a line from stdin*/
}
```
Buffer Overflow Stack Diagram

**Before Execution**

**Scenario 1: Normal Execution**

**Scenario 2: Buffer Overflow Attack**
(Not drawn to scale)
Example: Cross-Site Scripting

```
<c:if test="${param.sayHello}">
  Hello ${param.name}!
</c:if>
```

“We never intended the code that's in there to actually be production-ready code.”

- Ryan Asleson
Reliving Past Mistakes

- Cross-site scripting looks more and more like buffer overflow

Buffer Overflow
- Allows arbitrary code execution
- Easy mistake to make in C/C++
- Exploit is hard to write
- Well known for decades

Cross-Site Scripting
- Allows arbitrary code execution
- Easy mistake to make
- Exploit is easy to write
- Well known for a decade
What About Ajax?

• Today’s rage or tomorrow’s security disaster?
• Could more JavaScript possibly be better?
• Sample of the almost 400 JavaScript CVE entries:

CVE-2007-1794: The Javascript engine in Mozilla 1.7 and earlier… can allow remote attackers to execute arbitrary code.

CVE-1999-0793: Internet Explorer allows remote attackers to read files by redirecting data to a Javascript applet.

CVE-1999-0790: A remote attacker can read from a Netscape user's cache via JS

CVE-1999-0347: Internet Explorer 4.01 allows remote attackers to read local files and spoof web pages via a "%01" character in an "about:" Javascript URL, which causes Internet Explorer to use the domain specified
Cross-Site Request Forgery (CSRF)

- Cross-Site Request Forgery
  - JavaScript submits HTTP requests on victim's behalf
  - Allows attacker to submit commands, but not inspect the response (Same Origin Policy)

- Application is vulnerable if it:
  - Relies on user’s identity (e.g. persistent or session cookies)
  - Does not have secondary authentication mechanism

- Attack against data integrity
Ajax - The Case of the Vanishing “X”

- JavaScript/JSON is gradually replacing XML in Ajax applications

XML
<book>
  <title>JavaScript, the Definitive Guide</title>
  <publisher>O'Reilly</publisher>
  <author>David Flanagan</author>
  <cover src="/images/cover_defguide.jpg" />
  <blurb>elit.</blurb>
</book>

JSON
{
  "book": {
    "title":"JavaScript, the Definitive Guide",
    "publisher":"O'Reilly",
    "author":"David Flanagan",
    "cover":"/images/cover_defguide.jpg",
    "blurb":"elit."
  }
}
New: JavaScript Hijacking - 1/2

- Builds on CSRF
- Breaks confidentiality through loophole in Single Origin Policy
- Vulnerable if:
  - Site responds to HTTP GET
  - Transmits sensitive data in JavaScript syntax
New: JavaScript Hijacking - 2/2

Malicious Server

1) “show me dancing pigs!”
2) “check this out”
3) confidential data

Browser

Mal page
{ witness code }
<script src="...">

Ajax Application

GET

JavaScript
Confusion over JavaScript Hijacking

- A system administration problem
- Not unique – just a way to parse results of CSRF
- Not interesting – just “View Source”
- JavaScript Hijacking can be prevented by:
  - Defaulting to HTTP POST
  - Using parseJSON() instead of eval()
  - Using object notation {} rather than array notation []
  - Checking for “application/json” in content-type
## How 12 Popular Frameworks Stacked Up

<table>
<thead>
<tr>
<th>Framework</th>
<th>Summary</th>
<th>Prevents JavaScript Hijacking?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>Supports JSON. Defaults to POST when no method is specified, but is easily customizable for using either POST or GET.</td>
<td>No</td>
</tr>
<tr>
<td>Script.aculo.us</td>
<td>Supports JSON. Provides additional UI controls and uses the <strong>Prototype</strong> library for generating requests.</td>
<td>No</td>
</tr>
<tr>
<td>Dojo</td>
<td>Supports JSON. Defaults to POST, but does not explicitly prevent JavaScript Hijacking.</td>
<td>No</td>
</tr>
<tr>
<td>DWR 1.1.4</td>
<td>Uses an expanded version of JSON. Does not implement any JavaScript Hijacking prevention mechanisms.</td>
<td>No</td>
</tr>
<tr>
<td>Moo.fx</td>
<td>Supports JSON. Defaults to POST, but can easily be configured to use GET.</td>
<td>No</td>
</tr>
<tr>
<td>jQuery</td>
<td>Supports JSON. Defaults to GET.</td>
<td>No</td>
</tr>
<tr>
<td>Yahoo! UI</td>
<td>Supports JSON. Responds to GET requests.</td>
<td>No</td>
</tr>
<tr>
<td>Rico</td>
<td>Does not currently support JSON, but will in the future. Supports XML as a data transfer format. Defaults to GET.</td>
<td>N/A</td>
</tr>
<tr>
<td>Microsoft Atlas</td>
<td>Supports JSON. Uses POST by default, but allows programmers to easily change POST to GET and encourages doing so for performance and caching.</td>
<td>No</td>
</tr>
<tr>
<td>MochiKit</td>
<td>Supports JSON. Defaults to GET.</td>
<td>No</td>
</tr>
<tr>
<td>xajax</td>
<td>Does not currently support JSON. Supports XML as a data transfer format.</td>
<td>N/A</td>
</tr>
<tr>
<td>GWT</td>
<td>Supports JSON. Uses POST by default; however, documentation describes how to make GET requests instead and does not mention any security ramifications.</td>
<td>No</td>
</tr>
</tbody>
</table>
Defenses Against JavaScript Hijacking

• Prevent CSRF
  – Decline malicious requests by requiring unique token
  … and remember
  – Default to POST not enough
    (Developers add GET so that result can be cached)
  – Check for a known HTTP header not enough
    (Flash CSRF vulnerability)

• Prevent execution of JavaScript
  – while(1);, /* ... */ , etc
  … and remember
  – calling parseJSON() rather than eval() does not help
Wanton Web Data

• Antipattern: Mixing code and data fluidly
• Mixing code and data is never a good idea
• Old or new, this mysterious vulnerability
  – Makes it hard for programmers to code securely because the convention is based on insecure practices
• Represents a regression
  – XML is designed to represent and transport data
• Disappointing—frameworks should improve security, not hinder it
MYSTERY III

LICENSTIOUS LISTENERS
Licentious Listeners:
Programs must accept input, but if they accept too much it can squander resources and bring the system to its knees.
Rosetta Stone

• A single story translated across languages
  – Egyptian Hieroglyphics
  – Egyptian Demotic
  – Classical Greek

• A single vulnerability translated across languages
  – C
  – C++
  – Java
  – .NET and beyond
A Vulnerability Rosetta Stone

• Oldest trick in the book:
  
  ```c
  char buf[128];
  gets(buf);
  ```

• At runtime:
  
  ```
  warning: this program uses gets(), which is unsafe.
  ```

  (one of 4 attacks used by the Morris Worm)
A Vulnerability Rosetta Stone

• C++:

```cpp
char buf[128];
cin >> buf;
```

• At runtime:

(silence)
A Vulnerability Rosetta Stone

• Now Java:

```java
String str;
str = bufferedReader.readLine();
```

• How much data is read? How much data do you have?
A Vulnerability Rosetta Stone

• Improvement? C++:

```cpp
string str;
cin >> str;
```

• How much data is read? How much data do you have?
Licentious Listeners

• Antipattern: Reading unbounded input
• Just because you can't execute code doesn't mean you don't win
• Old or new, this mysterious vulnerability
  – Is destined to be repeated again and again
  – Favors ease of use over controlled behavior and security
• Is an unfortunate holdover that hasn't been fixed since gets()
• Disappointing—we've known the right way to read input for decades
MYSTERY IV

DELETERIOUS DELVERS
Deleterious Delvers: Input validation based on blacklisting can lead to a false sense of security, which is sometimes more dangerous than no security at all.
PHP magic_quotes_gpc

• **Runs** `addslashes()` **on values from GET, POST, and COOKIE**

• Escapes the following SQL meta characters: single quotes ('), double quotes ("), null bytes (NULL), and backslashes (\) using a backslash (single quote for ' if `magic_quotes_sybase` is true)

• Fails to prevent SQL injection in some cases; does nothing for vulnerabilities that rely on other metacharacters (e.g. XSS, etc)

• Deprecated in PHP 5.3.0 and removed in PHP 6.0.0
Example: `magic_quotes_gpc`

- Queries that expect un-quoted non-strings are trivially vulnerable

```php
// "0; DELETE FROM usr";
$id = $_POST['userId'];
mysql_query("SELECT * FROM usr WHERE id={$id}" );
```

- Queries that use `LIKE` are vulnerable to attacks using `%` and `_`

```php
$sub = $_POST['subject'];
mysql_query("SELECT * FROM msg WHERE subject \ LIKE '{$_sub}%':");
```
Fast Forward

• Sun JSTL
  – `<c:out>` tag applies XML encoding by default

• Microsoft .NET Framework
  – Validation framework uses blacklisting on incoming request parameters and some output tags
Example: Context is King

• XML encoding is sufficient in some contexts:

```html
<html>
Your address is <c:out value="${sender}"/>
</html>
```

• … and insufficient in others:

```script
alert(Your address is <c:out value="${sender}"/>);
</script>
Deleterious Delvers

- Antipattern: Blacklisting against specific attacks
- Symptom of "too good to be true" syndrome; input validation is hard
- Old or new, this mysterious vulnerability
  - Leads to a false sense of confidence when it comes to security
  - Blurs the line between trusted and untrusted
  - Makes it less likely that good input validation mechanisms will be built
- Disappointing—we've bemoaned overly broad black lists for years
LEARNING FROM PAST MISTAKES
Conclusions

• The same mistakes manifest themselves again and again
• Fundamental principals hold true (in light of competition)
  – Preserve trust boundaries
  – Do not mix code and data
  – Bound operations that read input
  – Use whitelists and indirect selection for input validation
• Increasingly we depend on frameworks to get security right
• Framework architects must choose security over ease of use
Success is foreseeing failure

– Henry Petroski
An Experiment

• Fortify documented JavaScript Hijacking in 2007

• We immediately contacted all of the frameworks we discovered were vulnerable and:
  – Shared our full research into the vulnerability
  – Provided their developers guidance and patches for remediation
  – Constructively debated details about the vulnerability and fix
We will fix it

- “…Thanks for the heads up on this…”
- “…Thanks for the paper. We are looking at the issue, and we’re starting to formulate some solutions that mesh well with what you’re suggesting…”
Case Study: JavaScript Hijacking (2/4)

- This is not a client-side framework problem
  - “…Entirely dependent on the server to do the right thing”
  - “Why the hell should there be security documentation in client frameworks?”
  - “I added comment stripping support so that people would shut up, not because it’s useful in theory or practice…”
Case Study: JavaScript Hijacking (3/4)

• You are recommending bad practices
  – “But by recommending bad practices, and by failing to strongly recommend good practices, you are making things worse…”
Case Study: JavaScript Hijacking (4/4)

- 4 mentioned in documentation:
  - dojotoolkit.org/2007/04/02/note-javascript-hijacking
  - getahead.org/dwr/security/script-tag-protection
  - www.prototypejs.org/learn/json
  - developer.yahoo.com/security/
A Look to the Future of Web Security

• Better web standards
• Frameworks offer a chance to build security in
  – New platforms and frameworks making progress
• Better automated analysis
  – Software-layer defenses
• Process, process, process
Better Web Standards

• Cookies – Broken
  – Need *working* HTTP only cookies

• Browsers – Broken
  – Need to distinguish between scripts from different domains
  – Need to write make it easier to distinguish code and data

• Collin Jackson, Stanford
  (http://www.collinjackson.com/)
Frameworks Offer a Chance to Build Security In

• Provide secure defaults
• Built-in security features
  – Prevent CSRF / JavaScript Hijacking by default
  – Basis for input validation framework
• Opportunity for better validation
  – Better separation between display and controller
  – Better definition for browser/server interaction
New Platforms and Frameworks Making Progress

• Flash?
  – Documented security best-practices
  – Hard to mistake data and code
  – Well-defined system for cross-domain communication

• PHP 6?
  – Removes several (already deprecated) insecure features
    • register_globals
    • magic_quotes_gpc
    • ...
Better Automated Analysis

• Investigate customization

• Map tool against security standards; best scenario is cyclic:
  – The tool reinforces coding guidelines
  – Coding guidelines are written with automated checking in mind

• Software more amenable to automated analysis
  – Developers are encouraged to code in verifiable ways
  – Building blocks follow the same advice (frameworks)
  – Security must trump usability in some case

• Software-layer protections
Security in the Development Lifecycle
More Questions than Answers

- Same activities for all software project?
- How to get budget / internal support?
- Which vulnerabilities do I have to fix?
- What about outsourcing?
- How to handle open source?
- Who does the work?
Objective: Describe What Works

BSIMM

• Building Security In Maturity Model

• Real data from real initiatives
More Questions than Answers

• Same activities for all software project?
• How to get budget / internal support?
• Which vulnerabilities do I have to fix?
• What about outsourcing?
• How to handle open source?
• Who does the work?
Fortify's Most Recent Contribution

- Brian Chess (Fortify); Gary McGraw and Sammy Migues (Cigital)
- Build a maturity model from actual data gathered from 9 large-scale software security initiatives
- Create software security framework
- Nine in-person executive interviews
- Build bullet lists (one per practice)
- Bucketize the lists to identify activities
- Create levels
  - Objectives → Activities
  - 110 activities supported by real data
  - Three levels of “maturity”
Real World Data

- **Age**
  - Avg 5.25 yrs
  - Newest 2.5
  - Oldest 10

- **SSG Size**
  - Avg 41
  - Smallest 12
  - Largest 100
  - Median 35

- **Satellite size**
  - Avg 49
  - Smallest 0
  - Largest 300
  - Median 20

- **Dev size**
  - Avg 7750
  - Smallest 450
  - Largest 30,000
  - Median 5000
Software Security Framework

<table>
<thead>
<tr>
<th>Governance</th>
<th>Intelligence</th>
<th>SSDL Touchpoints</th>
<th>Deployment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strategy and Metrics</td>
<td>Attack Models</td>
<td>Architecture Analysis</td>
<td>Penetration Testing</td>
</tr>
<tr>
<td>Compliance and Policy</td>
<td>Security Features and Design</td>
<td>Code Review</td>
<td>Software Environment</td>
</tr>
<tr>
<td>Training</td>
<td>Standards and Requirements</td>
<td>Security Testing</td>
<td>Configuration Management and Vulnerability Management</td>
</tr>
</tbody>
</table>


Common Ground

• Everyone has a software security group (SSG)
• SSG is roughly 1% size of dev team
• Ten activities that ALL do

1. Evangelist role
2. Policy
3. Awareness training
4. History in training
5. Security features
6. Static analysis
7. SSG does ARA
8. Black box tools
9. External pen testing
10. Good network security
Ten Surprising Things

1. Bad metrics hurt
2. Secure-by default frameworks
3. WAF Myth
4. QA can’t do security
5. Evangelize over audit
6. ARA is hard
7. Practitioners don’t talk attacks
8. Training is advanced
9. Decline of Pen Testing
10. Fuzz testing
Building Security In Maturity Model (BSIMM)

- Release March 10th (or sooner)
- Top-down presentation through GOALS and OBJECTIVES
- 110 activities with examples
- Three levels of maturity
- Creative commons

- How to use the model
- Where do you stand?
- What should you do next?
Report Card
Learn More

• Nine Things Everybody Does

• Top 10 Surprises

• A Software Security Framework

• Coming March 10th
• http://www.bsa-mm.com
PARTING THOUGHTS
<Your Code>
The Buck Stops With Your Code

• Security problems everywhere you look
  – Languages, libraries, frameworks, etc.

• Right answer
  – Better languages, libraries, frameworks, etc.

• Realistic answer
  – Build secure programs out of insecure pieces
Summary

• Security is part of programming
• Mistakes happen. Plan for them.
• Learn to identify old "bad ideas" reincarnated
  – Blurring trust boundaries
  – Mixing code and data
  – Unbounded operations that read input
  – Blacklists for input validation
• Framework designers play an important role in the solution
  – Don't let them forget it
• In the end, you're the one responsible for your software
  – Push for better frameworks and browser standards
  – Use automation and process to mitigate risk today