Secure Code Generation for Web Applications

Martin Johns
ISL, Universität Passau
martin.johns@uni-passau.de
Outline

1. Past activities
2. String-based code injection
3. Secure code generation
Outline

1. Past activities
2. String-based code injection
3. Secure code generation
Me, myself, and I

Martin Johns

• Worked as a developer for quite a while
  • Wrote a considerable amount of insecure code
• Joined Prof. Posegga’s group in 2005 to work in the Secologic project
  • Joint project with SAP, Commerzbank, and Eurosec
  • Goal: Establishing the state of the art in software security
  • http://www.secologic.org
• Since 2007 further research projects
  • ScanStud, evaluation static analysis, with Siemens CERT
  • FLET, language-based security for web apps, with SAP Research
  • ORKA, dynamic access control (project management), with Fraunhofer and others
• Personal focus on web application security
WebAppSec: Mitigation

XSS/Session Hijacking

• Protection against XSS-based session hijacking attacks
• Implementation: Server-side reverse proxy
• [Johns, ESORICS 06]

Cross-site Request Forgery

• Client-side protection against CSRF attacks
• Implementation: Proxy, browser extension
• [Johns & Winter, OWASP EU 06]

Browser-based attacks on intranet resources

• Protection of intranet resources against JS malware and DNS rebinding attacks
• Implementation: Browser extension
• [Johns & Winter, DIMVA 07], [Johns, JICV 08]
WebAppSec: Detection and prevention

Cross-site Scripting detection
- Server-side detection of XSS exploits through passive HTTP monitoring
  - [Johns, Engelmann, Posegga, ACSAC 08]

Static analysis
- Evaluation of commercial static analysis tools
- Joint work with the Siemens CERT, presented at OWASP EU 08

Detection of string-based code injection
- Instruction set randomization for web applications
  - [Johns & Beyerlein, ACM SAC 07]

Language-base prevention of code injection vulnerabilities
- Topic of today’s talk
Outline

1. Past activities
2. String-based code injection
3. Secure code generation
Web application architecture

Native Code

Database

SQL

XML

Filesystem

Foreign Code

Web server

PHP

Web browser

HTML

JavaScript

CSS
String based code injection

Today’s most prevalent security bug pattern

• Affects foreign code creation

Types

• Cross-site scripting
• SQL injection
• Shell injection
• Path traversal
• XPath injection, LDAP injection, JSON injection, …
String based code injection

Dynamic code assembly

```
$pass = $_GET["password"];

$sql = "SELECT * FROM Users WHERE Passwd = '' + $pass + "'";  
```
String based code injection

```php
wget http://site.com/login?password='%20OR%20'1'=1

$pass = $_GET["password"];  
$sql = "SELECT * FROM Users WHERE Passwd = '' + $pass + '';";
```
String based code injection

wget http://site.com/login?password='%20OR%201'='1

$pass = '' OR '1'='1'';

$sql = "SELECT * FROM Users WHERE Passwd = '' OR '1'='1''"
String based code injection

The programmer’s view:

```php
$pass = $_GET[“password”];

$sql = ‘SELECT * FROM Users WHERE Passwd = ‘’ + $pass + ‘’’;
```
String based code injection

The database’s view:

```php
$pass = $_GET["password"];  
$sql = "SELECT * FROM Users WHERE Passwd = \\
\"\" + $pass + \"\";```

Code

Data

?
String based code injection

The database’s view:

```
$pass = "' OR 1=1";
```

```
$sql = "SELECT * FROM Users WHERE Passwd = ' OR '1'='1";
```

⇒ Implicit foreign code creation through string-serialization
Outline

1. Past activities

2. String-based code injection

3. Secure code generation
   • General approach
   • Datatype
   • Design and Implementation
   • Evaluation
   • Conclusion
Outline

1. Past activities

2. String-based code injection

3. Secure code generation
   • General approach
   • Datatype
   • Design and Implementation
   • Evaluation
   • Conclusion
General approach

Similarities within the bug pattern:
  • String-based foreign code assembly
  • [Unmediated interfaces to the external interpreters]

Solving the problem on the programming language level
  • Goal: Secure dynamic foreign code assembly
  • “How do we need to extend/modify the existing practice to reliably prevent the vulnerability class?”

Methodology
  • Removal of the vulnerability class’ fundamental requirements
    • Exchange the string type for code assembly
    • Abstract all direct interfaces to external interpreters
      – Offering a secure alternative is not enough
Design objectives

Do not invent a new language

• The developed concepts should be applicable for any modern programming language

Closely mimic the foreign syntax

• Respect the design decisions of the language’s inventors
• No additional training costs

Maintain the flexibility of the String type

• Dynamic assembly of foreign code is a powerful tool

→ Keep the programmers happy
Key components (I)

Programming Language

Database

External Interpreters

Web Services

Web Browser

Native code

Datatype

Abstraction Layer (Application Server)

Language integration

Foreign code

Native code

Language integration
Outline

1. Past activities
2. String-based code injection
3. Secure code generation
   • General approach
   • Datatype
   • Design and Implementation
   • Evaluation
   • Conclusion
The datatype

“Foreign Language Encapsulation Type” (FLET)

- Part of the native language
- Capable of encapsulating foreign code
- Provides strict separation between data and code

\[ \text{Code} \]

\[ \text{Data} \]

Objectives

- Code-elements should only be created explicitly
  - Instead of “take this string and treat it as code” the programmer has to specify the exact syntactic purpose of each element

\[ \text{sql} = \text{"SELECT * FROM Users WHERE Passwd = " + $\text{foo};} \]
Formal considerations (I)

Type systems for security properties

• Dominant focus: Confidentiality
• Bell-LaPadula model [Bell & LaPadula 73]
  • Multilevel security
    – Simplest case: public/secret
  • Information flow constraints
    – Simple Security Property (no read-up)
    – *-Property (no write-down)
• [Denning & Denning 77], enforcement through static program analysis
• [Volpano & Smith 96], formalizes Denning’s approach through a type system
  • public is a subtype of secret
  • Compile time enforcement through type checking
Formal considerations (II)

Biba model [Biba 77]

- Dual model to Bell-LaPadula
- Enforces integrity constraints
  - No information flows from low to high integrity
- Two axioms
  - Simple Integrity Axiom (no read-down)
  - *-Integrity Axiom (no write-up)
- Can be modeled analogous to [Volpano & Smith 96]
Applying this to our case

Our problem can be abstracted into integrity constraints

- Code elements == high integrity
- Data elements  == low integrity
- Code injection ⇒ information flow from low integrity to high integrity

Prevention of direct flows from low to high

- Indirect flows have to be possible (Example: Wiki)
- Concentration on the *-Integrity axiom
- Hence, we only require a subset of Volpano’s typing rules

Question:

- Definition of data/code element?
Identifying Language Elements

Needed: Mapping data/code to syntactical elements

```sql
$sql = "SELECT * FROM Users WHERE Passwd = 'foobar' ";
```

General token-classes, derived from foreign grammar

- **Static-elements**
  - Names defined in the language’s grammar
  - E.g., keywords, punctuators, tag-names, ...

- **Identifier-elements**
  - Names defined on compile time
  - E.g., variable, function or table names

- **Values**
  - Values vary on run-time
  - E.g., strings, integers, attribute-values, ...

- **Not to be defined on runtime**
  $$\Rightarrow code$$

- **OK to be defined on runtime**
  $$\Rightarrow data$$
Resulting data & integrity types

Three additional native datatypes to represent foreign syntax elements

- code-token, represents foreign static- and identifier-elements
- data-token, represents foreign data-elements
- FLET, container type, representing a token stream

Two integrity classes

- $CT$ (assigned to code-tokens)
- $DT$ (assigned to data-tokens, strings, integers, floats,...)

Subtype relationship: $CT \subseteq DT$

\[
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \tau'} \quad \tau \subseteq \tau' \quad (\text{subtype})
\]

\[
\frac{\Gamma \vdash e : \tau \, \text{var}}{\Gamma \vdash e := e' : \tau} \quad (\text{assignment})
\]

\[
\frac{\Gamma \vdash e : \tau \, \text{cmd}}{\Gamma \vdash e' : \tau} \quad (\text{assignment})
\]
Resulting integrity types (II)

Claim

• By using the typing rules from [Volpano & Smith 96] that enforce the *-Axiom, CT-typed expressions cannot be defined by DT-typed values

Proof

• By induction through typing rules

⇒ As attacker controlled data enters the application typed DT, it can not end up in a CT (code) context
The FLET

FLET is modeled as a type-conserving container ⇒ Sequence of data- and code-elements (tokenstream)

(FLET) \[ \Gamma \vdash e_i : \tau_i \quad \tau_i \in \{DT, CT\} \quad i \in 1...n \]
\[ \Gamma \vdash \text{FLET}(e_1 : \tau_1, ..., e_n : \tau_n) \]

(retrieval) \[ \Gamma \vdash M : \text{FLET}(e_1 : \tau_1, ..., e_n : \tau_n) \quad \tau_i \in \{DT, CT\} \quad i, j \in 1...n \]
\[ \Gamma \vdash M.e_j : \tau_j \]

⇒ Within the native language, data and code are cleanly separated

• Final serialization step is done outside of the language’s scope
Outline

1. Past activities
2. String-based code injection
3. Secure code generation
   • General approach
   • Datatype
   • Design and Implementation
   • Evaluation
   • Conclusion
Implementation target

J2EE/HTML/JavaScript

Interesting implementation target

- XSS is one of the most pressing issues today
- Two distinct foreign syntaxes (HTML, JavaScript)
- Many non-trivial injection attack-vectors
Language Integration (I)

“How do we fill the FLET?”

• Incorporation of the foreign language elements into the native language

Three approaches

• API
• Extending the native language’s grammar
• Usage of a pre-processor
Language integration (I)

API approach

```java
SQLFlet q = new SQLQuery("SELECT").addMetaChar("*").addKeyWord("FROM").addString("Users");
```

Advantages:

- No additional means necessary (besides creating the API)
- No changes in the native language’s syntax, compiler or interpreter

Disadvantages:

- Cumbersome syntax (esp. in the case of complex languages)
- Hard to maintain code
- Programmer’s acceptance is doubtable
Language integration (II)

Extending the native language’s grammar:

```sql
SQLFlet q = SELECT * FROM Users;
```

Advantages:

• Strong mimicking of the foreign language’s syntax
• Good support for compile time checking
• Almost no learning curve

Disadvantages:

• Requires profound changes in the native language’s syntax, compiler or interpreter

→LINQ
Language integration (III)

Usage of a pre-processor:

```sql
SQLFlet q = $$ SELECT * FROM Users; $$
```

Advantages:

- Strong mimicking of the foreign language’s syntax
- Integration of the complete foreign syntax
- Almost no learning curve for developers

Disadvantages:

- Compiled code != source code
- Requires changes in the build process or the language’s compiler
- Poor support for compile-time checking of the foreign syntax
Practical realization

Source-to-source translation that translates foreign code into an Java API representation

- The pre-processor locates and parses the foreign code into tokens
- Then it generates the corresponding API calls which instantiate the token-elements and add them to the FLET container
API design

Code elements

• Completely static instantiation calls
  
  \[
  \text{FLET.addJS\_Keyword}() \Rightarrow f.addJS\_while()
  \]

Identifier elements

• Definition through static values
  
  \[
  \text{FLET.addJSIdentifier}(const \text{string}) \Rightarrow f.addJSIdentifier("document")
  \]

Data elements

• Definition through native types
  
  \[
  \text{FLET.addJS\_data}\text{string}) \Rightarrow f.addJS\_data(native\_var)
  \]
Implementing the preprocessor

Simple meta-syntact for mixing foreign and native code

```
String name = request.getParameter("name");
HTMLFlet h = new HTMLFlet();
h.addOpeningTag_b();
h.addText("Hallo ").addText(name);
h.addClosingTag_b();
```

Translated to native Java (using the FLET API)

```
String name = request.getParameter("name");
HTMLFlet h = new HTMLFlet();
h.addOpeningTag_b();
h.addText("Hallo ").addText(name);
h.addClosingTag_b();
```

Furthermore

- Meta-syntax for simple FLET operations, such as concatenation
- API calls for splitting, searching, and iterating FLETs
Abstraction Layer: Position

Three possible positions:

- Integral part of the native language
- As a standalone intermediate entity
- Part of the external entity
Abstraction Layer: Position (II)

Decision for the prototype:

• Implementation within the native language’s runtime environment

Reasons:

• Integration on the server side
• Convenient interface to the FLET
• No deployment problems
The actual communication is still character-based

• The FLET has to be securely serialized

Strategies

• Translation into non-executable representation
  • If the foreign language provides such a representation
    – HTML: Entities (&…;)
    – URLs: Percent-encoding (%..)
    – JavaScript: various, e.g., String.fromCharCode()
  • Current code context is significant
    – Applicable representation might depend on this context
    – Reliable context information through FLET available

• Comparison of parse trees
  • [Su & Wasserman 06]
  • Replacement of data-values with dummy-values
Implementation

Project with SAP Research

J2EE filter

• Intercepts outbound communication
• Provides a FLET-based interface
• The legacy string-based interface is rerouted to log
• Uses only standard J2EE techniques
• Easy deployment
Outline

1. Past activities

2. String-based code injection

3. Secure code generation
   • General approach
   • Datatype
   • Design and Implementation
   • Evaluation
   • Conclusion
Evaluation (I)

Protection Evaluation

• Servlet that blindly echos user-provided data back into various HTML/JavaScript contexts
• Tested against documented XSS attack techniques

```java
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws IOException {
    String bad = req.getParameter("data");
    [...] 
    HTMLFlet h $==$ <h3>Protection test</h3> $$ 
    h $+\$ Text: $data(bad)$ <br /> $$ 
    h $+\$ Link: <a href="data(bad)">link</a> <br /> $$ 
    h $+\$ Script: <script>document.write(data(bad));</script><br /> $$ 
    [...] 
    FletPrinter.write(resp, h); // Writing the FLET content 
    resp.getWriter().println(bad); // Testing if the legacy interface 
    // is correctly disabled 
}
```
Evaluation (II)

JSPWiki

• Mature J2EE wiki engine
• ~ 70,000 LoC in 365 java/jsp-files
• Good evaluation target
  • Non-trivial dynamic HTML generation
  • No database backend

Porting to FLET

• 103 files had to be adapted
• ~ 1 person-week

Result

• Chosen version (2.4.103) had several XSS issues
• After porting, these issues were resolved
Evaluation (III)

Performance Evaluation

• Test machine: Windows XP running Apache Tomcat
• Measuring tool: HP Loadrunner simulating various, increasing numbers of simultaneous users

Result

• Average observed runtime overhead: up to 25%
Outline

1. Past activities
2. String-based code injection
3. Secure code generation
   • General approach
   • Datatype
   • Design and Implementation
   • Evaluation
   • Conclusion
Conclusion

Our approach

• reliably prevents string-based code injection,
• can be used for existing languages/frameworks/servers,
• allows integration of the complete foreign syntax,
• preserves (most of) the string-type conventions,
• and is applicable for all foreign language types
  • Query, mark-up, general purpose, hybrid, ...
• Furthermore, given a grammar and a token/type mapping, preprocessor and API could (in theory) be generated automatically
The end

Thanks for your attention

martin.johns@uni-passau.de
Related work

Preprocessor-based integration of foreign code
  • SQLJ, Embedded SQL

API-based integration
  • DOM, SQLDom

Direct integration
  • LINQ, EAX

Dynamic taint tracking
  • [Nguyen-Tuong et al. 05], [Pietraszek & Berghe 05], [Xu et al. 06]

During development
  • Static taint tracking [Huang et al. 04], [Livshits & Lam 05], [Jovanovic et al. 06]
  • Static ISR [Boyd & Keromytis 04], Prepared Statements
Taint analysis

Dynamic tainting

• Establishing on runtime if untrusted data ends up in security sensitive places
• To be effective, it has to be implemented on the character level (a.k.a. precise tainting)
  • This requires a low-level alteration of the String datatype
  • Hard to do as every single function that handles strings has to be instrumented
• Can’t be implemented on a source code level
  • Source code of the language’s interpreter is therefore required
• Relies on sanitation functions that remove the taint-flag
  • String-based code assembly remains
  • Would not have prevented the Samy-worm
FLET and LINQ

LINQ

• Foreign code integration on a semantical level
• Excellent
• Very good support for static checking of foreign code’s correctness
• Complete foreign syntax coverage is hard to achieve
• Focus on data-centric foreign code (SQL, XML)
  • General purpose/hybrid languages?

FLET

• Foreign code integration on a purely syntactical level
• Covers the full foreign syntax
• Flexible
• Can be implemented without changing the native compiler
• Uniform foreign code assembly method regardless of the actual foreign language
Extension of the formal model

Problem:

• By implementing the FLET through an API we allow a flow from strings to identifier-tokens
• This is not typeable in our current system

Solution: Three integrity types

• $DT$: Code-tokens
• $IT$: Identifier-tokens, constant strings
• $DT$: Date-tokens, all other native value

Subtyping relationship

\[ CT \subseteq IT \subseteq DT \]
Initial motivation

C

Programming Language

Memory Management

Java

Programming Language

Abstraction Layer (Virtual Machine)

Memory Management
Initial motivation

PHP, Java, etc.

Programming Language

Database

External Interpreters

Web Services

Web Browser

SQL

e.g., bash

XML

HTML / JS

?
Observation

• Desired data/code separation is actually only one-way
  • Stack traces, error-messages, etc.
• No definition of code-elements through data-values
⇒ Constraints on information flow

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Integrity level</th>
</tr>
</thead>
<tbody>
<tr>
<td>DT</td>
<td>Data</td>
<td>low</td>
</tr>
<tr>
<td>CT</td>
<td>Code</td>
<td>high</td>
</tr>
<tr>
<td>FLET</td>
<td>Record-type</td>
<td>((\tau_1, \ldots, \tau_n), \tau_i \in {high, low})</td>
</tr>
</tbody>
</table>
Motivation

PHP, Java, ASP, etc.

Programming Language

Database

External Interpreters

Web Services

Web Browser

Abstraction Layer (Application Server)

Programmer Language

Database

External Interpreters

Web Services

Web Browser
String based code injection

The native compiler/interpreter’s view:

```php
$pass = $_GET["password"]; $sql = "SELECT * FROM Users WHERE Passwd = '" . $pass . "'";
```

String String String
Software Security

Lessons learned from C security

When it comes to code-based security issues, there are three general approaches:

• **Mitigation**
  • Limit the adversaries abilities even when a vulnerability exists
  • Stack Guards, DEP, ASLR, etc.

• **Detection**
  • In the source code, e.g., static analysis
  • During execution, e.g., taint tracking

• **Prevention**
  • Removal of the root cause
  • CCured, Java vs. C
WebAppSec: Prevention

Topic of today’s talk...