



Security in knowledge

# Why Haven't We Stamped Out SQL Injection and XSS Yet?

Romain Gaucher

Coverity

Session ID: ASEC-F42

Session Classification: Intermediate

# About me

- ▶ Lead security researcher at Coverity
  - ▶ Have spent a fair amount of time on automated analysis of sanitizers, framework analysis, precise remediation advices, context-aware static analysis, etc.
- ▶ Officer at WASC, contributed to several community projects (Script Mapping, Threat Classification 2, CAPEC, etc.)
- ▶ Previously at Cigital and NIST
- ▶ On the web:
  - Twitter: @rgaucher
  - Coverity SRL blog: <https://communities.coverity.com/blogs/security>

# Agenda

- ▶ This talk is not about
  - ▶ what XSS or SQL injection are
  - ▶ how to exploit XSS or SQLi
  - ▶ static analysis heuristics to find these issues
  - ▶ etc.
- ▶ This talk is about
  - ▶ what developers need to do when they write a web application
  - ▶ importance of understanding SQL and HTML contexts
  - ▶ how these contexts relates to XSS and SQLi
  - ▶ remediation advices we usually hear from security professionals

# What we studied (and what we didn't)

- ▶ We analyzed **28** Java web applications and **154** versions over time. Most were open source projects. That's **6MLOC** Java and JSP code.
- ▶ For SQL injection:
  - ▶ focus on queries embedded in applications
  - ▶ we did not analyze the code of stored procedures
- ▶ For XSS:
  - ▶ focus on server-side pages
  - ▶ most pages have JavaScript but we did not try to understand the impact of JavaScript code (i.e., after it executes)

# Analyzed projects

- ▶ These applications used several different control frameworks, such as Spring MVC, Struts, etc.
- ▶ ORMs such as JPA or Hibernate are used in two-thirds of the applications
- ▶ All projects had JSP files, but its use is quite different:
  - ▶ One project has 24 lines of JSP, another 84,000

Jira	Tatami	MemoireM2	jforum3	EcuriesDuLoup
Cosmo	Nuxeo	OpenMRS	psi-probe	Liferay
Scrumter	Jackrabbit	dotCMS	ala-portal	Jahia
TradingSystemAid	Pebble	Pyramus	WiseMapping	jforum2
Nacre	Threadfix	YouWho	JSPwiki	Roller
MasterMusicStudio	Ubanist	Connect		

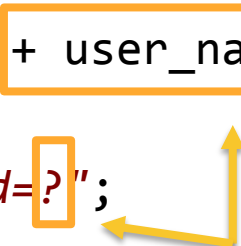
# Analysis

- ▶ This research relies heavily on the ability to statically compute contexts for HTML and SQL
  - ▶ We reported the contexts every time some dynamic data is inserted in HTML or SQL (injection site). We do not report the number of possible paths, just injection sites.
- ▶ Contexts computations
  - ▶ HTML contexts are derived from a HTML5 based parser [1], and simple CSS/JavaScript parsers
  - ▶ SQL contexts are derived from a parser that handles generically SQL, HQL, and JPQL
- ▶ Limitations
  - ▶ The contexts are computed in the Java program, we do not try to understand how JavaScript impacts the HTML contexts at runtime

# Concept of “injection site”

- ▶ The injection site is our unit for measuring the implicit or explicit string concatenations in a program
- ▶ We only care and report injection sites that are related to a sub-language we want to analyze: SQL, JPQL, HQL, HTML, JavaScript, and CSS
- ▶ Example of injection sites related to SQL:

```
String sql = "select id from users where 1=1";  
if (condition1)  
    sql += " and name='" + user_name + "'";  
if (condition2)  
    sql += " and password=?";
```



2 injection sites in one  
SQL query

# SQL Injection



Security in knowledge



# The root of all evil

- ▶ When parameterized queries are used correctly there are no real opportunities for SQL injection
  - ▶ The root of SQL injection therefore is string concatenation of a query string with tainted data
  - ▶ Bright idea: If we could eliminate the habit of developers using string concatenation for queries, we could eliminate SQL injection
- ▶ Let's examine the "common" security advice given to developers:
  - ▶ “Use an ORM”
  - ▶ “Use parameterized queries”

# "Use an ORM"

- ▶ ORMs are not typically vulnerable to SQLi
- ▶ Most projects use both ORM and non-ORM

	Project ID															
SQL	2	4	6	7	9	10	11	13	17	18	20	22	23	24	27	
JDBC API	✓		✓	✓	✓	✓	✓			✓	✓	✓	✓		✓	
Hibernate ORM						✓	✓			✓						
Hibernate Non-ORM	✓	✓				✓		✓	✓					✓	✓	
JPA ORM	✓	✓	✓					✓					✓	✓	✓	
JPA Non-ORM			✓													

# ORM versus Non-ORM

- ▶ Overall combination of ORM and non-ORM SQL access
  - ▶ Non-ORM: JDBC, Hibernate SQL / HQL, and JPA JPQL queries
  - ▶ ORM: `EntityManager.find()`, `EntityManager.persist()`, Hibernate Criteria, etc.
- ▶ ORMs are common in our dataset (67%)
  - ▶ Most use JPA, some use Hibernate exclusively
- ▶ Use of a query language (HQL, SQL, etc.) is common to all projects
  - ▶ SQL accounts for 63% of actual queries
  - ▶ HQL is second, with 31% queries
- ▶ Conclusion: string-based query construction is still very common in our dataset, even in applications using ORMs.

# String Concatenation

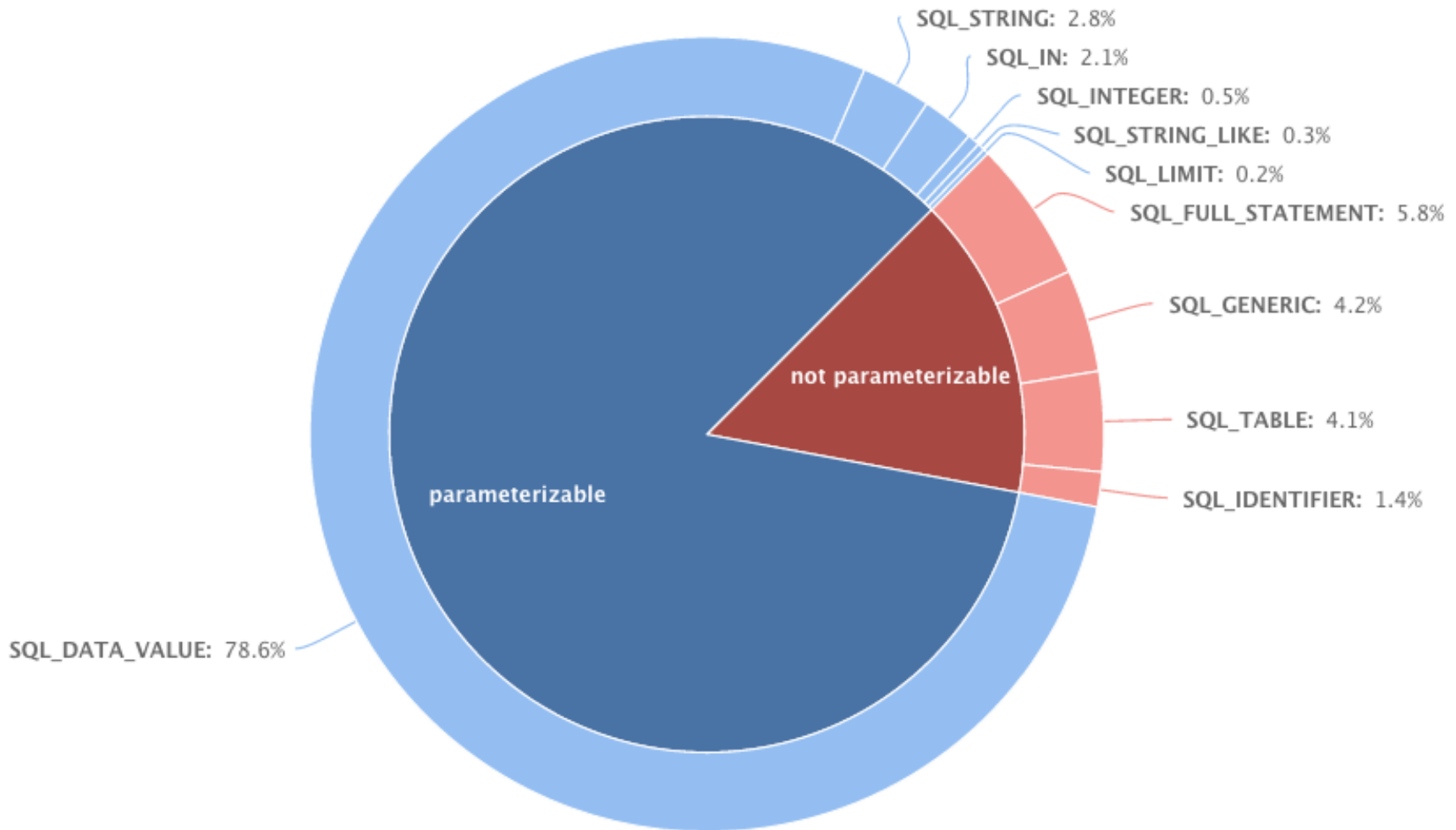
- ▶ Projects use a mixture of ORM and query languages. There is a need for query languages!
- ▶ “Use parameterized queries”

# SQL Contexts and Parameterization

- ▶ Analysis of 985 injection sites, and 545 unique queries (including JDBC, JPA, and Hibernate)
  - ▶ There is an average of 1.8 injection sites per query with a maximum of 22 injection sites in one query.
- ▶ We identified 10 different SQL contexts
  - ▶ Good: 85% of the injection sites are associated to a SQL context that can be parameterized
  - ▶ Problem: 15% of the queries cannot be parameterized

# “Use parameterized queries”

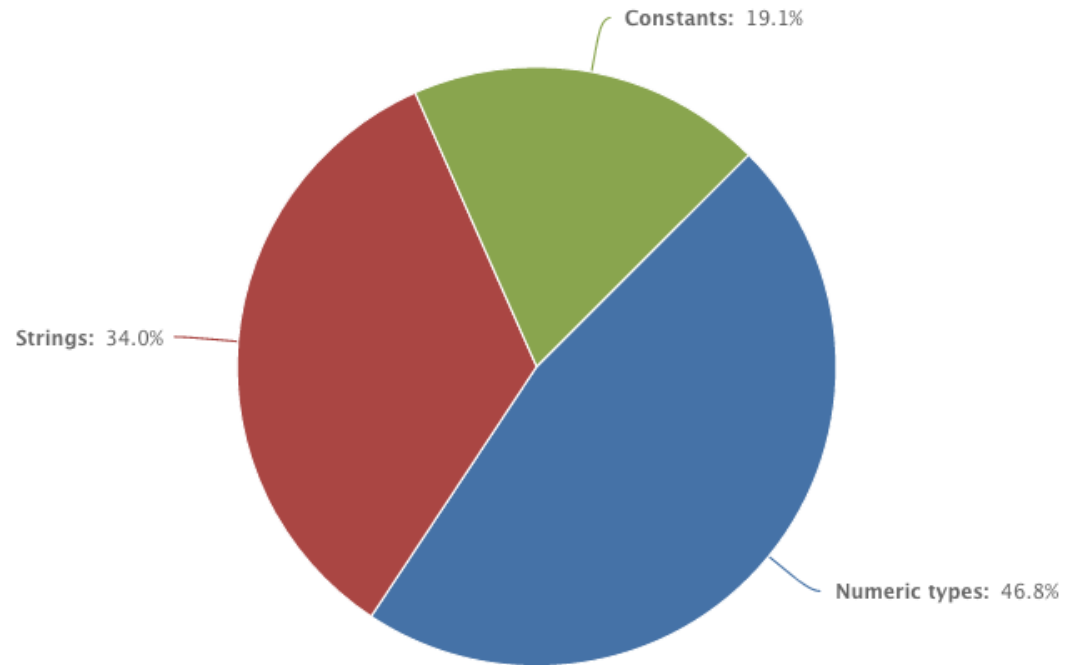
## Parameterizable vs. Not parameterizable



# Parameterizable Contexts

- ▶ Out of the 833 parameterizable contexts, **94%** are parameterized
- ▶ It's still not perfect; here's the breakdown of the remaining **6%** that could be parameterized

- ▶ **Constants: 19%**
  - ▶ Constant identifiers
  - ▶ Class names
- ▶ **Strings: 34%**
  - ▶ Good ol' strings
- ▶ **Safe types: 47%**
  - ▶ Numeric, Date, etc.



# Non-Parameterizable Contexts

- ▶ **SQL\_FULL\_STATEMENT (5.8%)**: the entire SQL query is not resolvable statically (DB, SQL file, configuration files, user, etc.)
  - ▶ It's not a "real" context and most likely a design decision
  - ▶ For web requests, they should have anti-CSRF and access controls in place
- ▶ **SQL\_TABLE (4.1%)**: table name
- ▶ **SQL\_IDENTIFIER (1.4%)**: column name or a variable
- ▶ **SQL\_GENERIC (4.2%)**: keyword, etc.



# Parameterize This!

▶ Table name:

```
stmt1= c.createStatement();  
stmt1.execute("TRUNCATE TABLE `" + table + "`");
```

▶ Trigger name:

```
Statement st = conn.createStatement();  
schema = StringUtils.quoteIdentifier(schema);  
String trigger = schema + '.'  
                + StringUtils.quoteIdentifier(PREFIX + table);  
st.execute("DROP TRIGGER IF EXISTS " + trigger);
```

▶ Conclusion: developers still need to concatenate in some queries.

# Concatenation

and non-ORM SQL technologies.  
pages.

- ▶ 15% of injection sites cannot be parameterized
- ▶ There's hope, most parameterizable ones are parameterized (94%)!
- ▶ 2% of parameterizable injection sites use dynamic strings that aren't parameterized; opportunity for SQL injection

# Now What?

- ▶ String concatenation cannot easily be eliminated
  - ▶ How can its effects be mitigated?
- ▶ Let's examine some other "common" security advice:
  - ▶ "Do input validation"
  - ▶ "Escape special characters"

# “Do Input Validation”

- ▶ Input validation = “*Security through serendipity*”
- ▶ Does the domain of valid input values happens to preclude security problems for all contexts where the data is used? Not necessarily.
- ▶ The input validation of data should be dictated by the functional requirements of the application, not by the security obligations of SQL contexts.
  - ▶ “Why can’t my password contain % ?”
  - ▶ “Why can’t Miles O’Brien create a profile?”
- ▶ Conclusion: you happen to be safe sometimes, you just cannot guarantee it throughout the application.

# String Concatenation

- ▶ Different contexts have different security obligations
- ▶ Functional requirements should drive input validation
- ▶ “Sanitize special characters”

# Sanitize special characters

- ▶ What's a special character? Different dialects have different requirements.
  - ▶ Does `\` escaping work for default PostgreSQL  $\geq 9.1$ ?
  - ▶ What needs escaping in a quoted identifier?
- ▶ Escapers might be useful in a special case where the application is doing a lot of dynamic queries and parameterization would require large scale refactoring.
- ▶ Not all contexts can use an escaper. Some characters just need to be filtered out.

# Concatenation

▶ Different security obligations

▶ Should drive input validation

- ▶ Different dialects have different requirements
- ▶ Can be applied incorrectly

# SQL Conclusions

- ▶ String concatenation cannot be eradicated from SQL queries.
- ▶ Input validation may not be adequate, sanitizers can help but can also hinder.
- ▶ Security ought to provide developers helpful advice:
  - ▶ Code, code, code...
  - ▶ Specific to a technology (JDBC, Hibernate HQL, etc.)
  - ▶ And specific to a context (table name, IN clause, etc.)
- ▶ Working with developers to build this kind of documentation for your organization's specific use of technologies and query styles will help reduce SQL injection defects.



# Cross-Site Scripting



# XSS is confusing

- ▶ From a code perspective, XSS isn't a single vulnerability. It's a group of vulnerabilities that mostly involve injection of tainted data into various HTML contexts.
- ▶ There is no one way to fix all XSS vulnerabilities with one magic "cleansing function". Developers really need to understand this.
- ▶ Let's examine the "common" security advices given to developers:
  - ▶ "Use HTML escaping"
  - ▶ "Don't insert user data in <random HTML location here>"
  - ▶ "Use auto-escaping template engines"

# “Use HTML escaping”

- ▶ HTML entity escaping can be used in several locations in an HTML document
- ▶ It can be used when the web browser will be able to process the HTML entities. For example, it is correct for the value of a `<div>` or an HTML attribute.
- ▶ However, HTML escaping cannot be used in all cases: you wouldn't escape an attribute name, or the content of a `<style>`
- ▶ To better understand when we can use this HTML escaper and when we cannot, we need to talk about HTML contexts

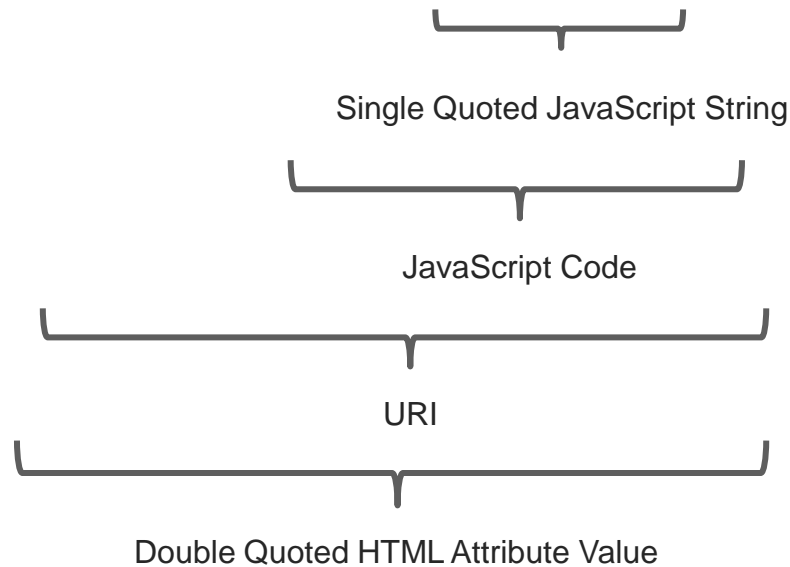
# HTML Contexts?

- ▶ No documentation really describes what HTML contexts are or gives an good list of them
- ▶ Practically, many places in an HTML page (inc. CSS, JS, etc.) have the same security obligation. These are the HTML contexts.
- ▶ Simple example, double quoted HTML attribute value

```
<div id="{inj_var}">...  
<pre class="{inj_var}">...
```
- ▶ Some of the contexts:
  - ▶ JavaScript string, HTML tag name, HTML attribute name, CSS string, CSS code, JavaScript code, HTML RCDATA, HTML PCDATA, HTML script tag body, URL, etc.

# The of nested contexts

```
<a href="javascript:hello('${inj_var}')">
```



## HTML Contexts Stack

Single quoted  
JavaScript string

JavaScript code

URI

Double quoted HTML  
attribute value

- ▶ Simplified: Our data is inserted inside a JavaScript String inside a URI inside an HTML attribute
- ▶ We'll note: HTML Attribute -> URI -> JS String

# The of nested contexts

```
<a href="javascript:hello('${inj_var}')">
```

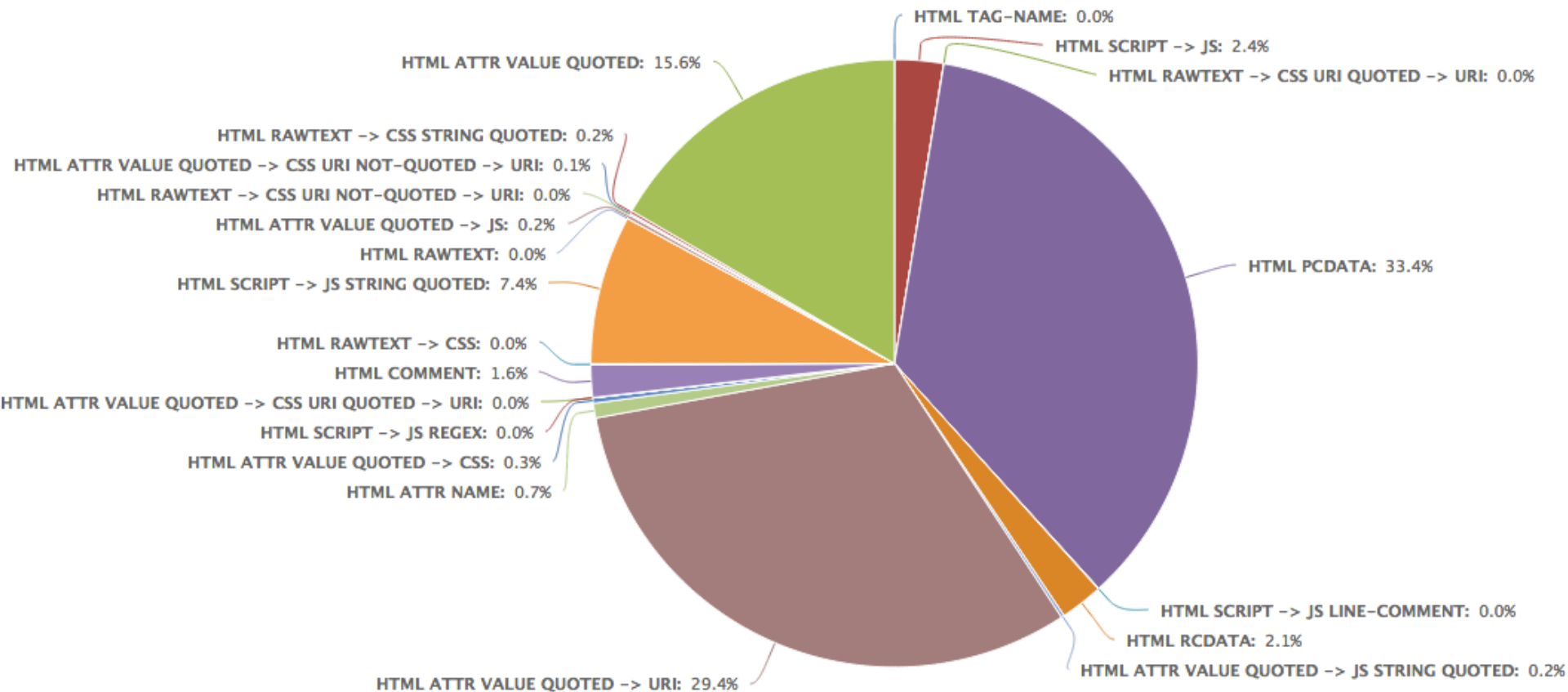
```
stack := {HTML Attribute -> URI -> JS String}
```

- ▶ The web browser will:
  - ▶ Take the content of the attribute `href` and HTML decode it
  - ▶ Analyze the URI and recognize the `javascript:` scheme
  - ▶ Take the content of the URI (i.e., after the scheme) and URL decode it
  - ▶ Since it's supposed to be JavaScript, the extracted content `hello('${inj_var}')` is sent to the JS engine for rendering
  - ▶ The JS engine will parse the program and especially the string that contains our `${inj_var}` by doing a JS string decoding
- ▶ Based on this flow we know what needs to be done to make a safe insertion for `${inj_var}` in this context

Well put grand'ma!

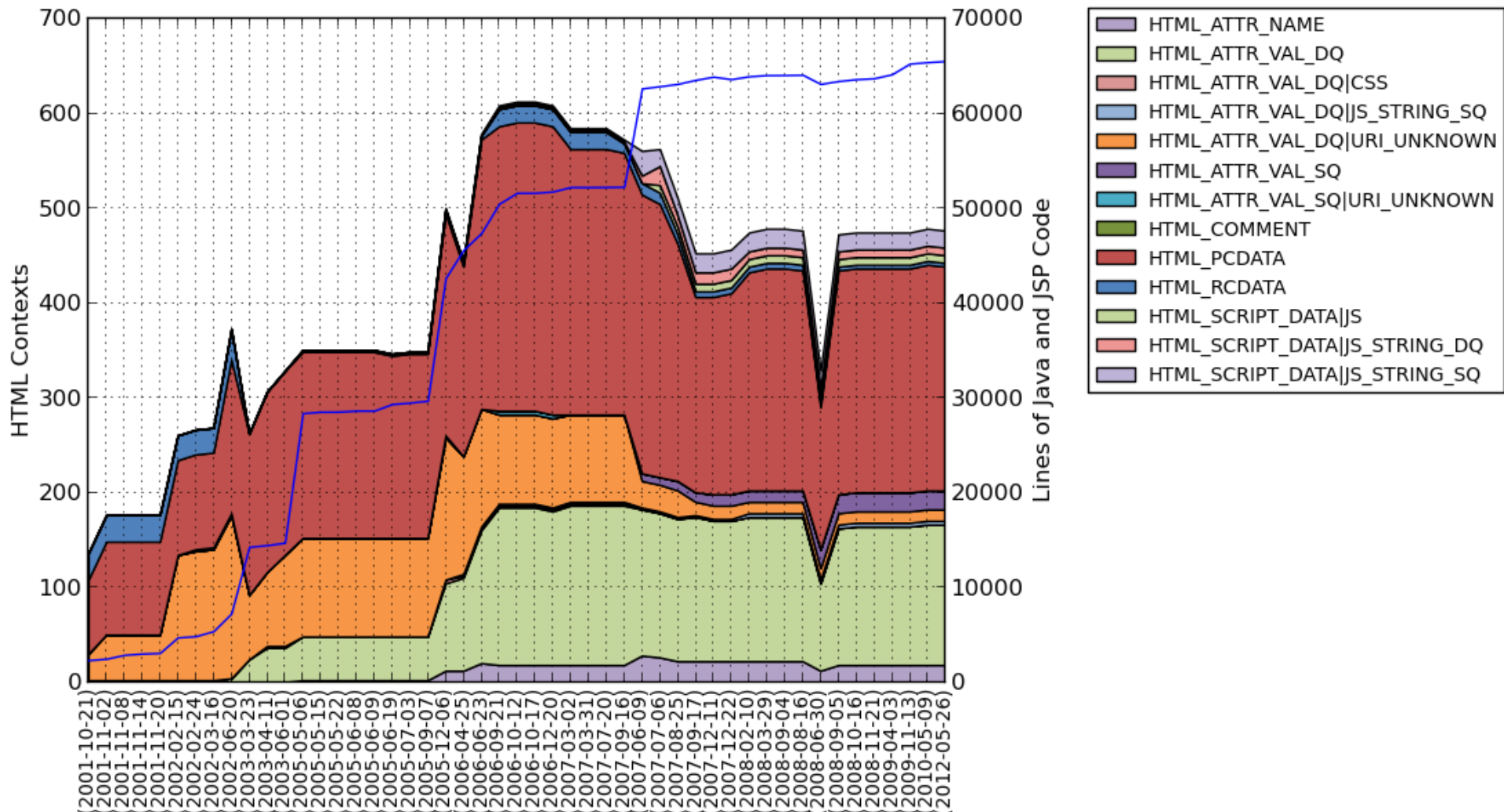


# Distribution of HTML contexts





# HTML contexts matters more now!



# Fix XSS

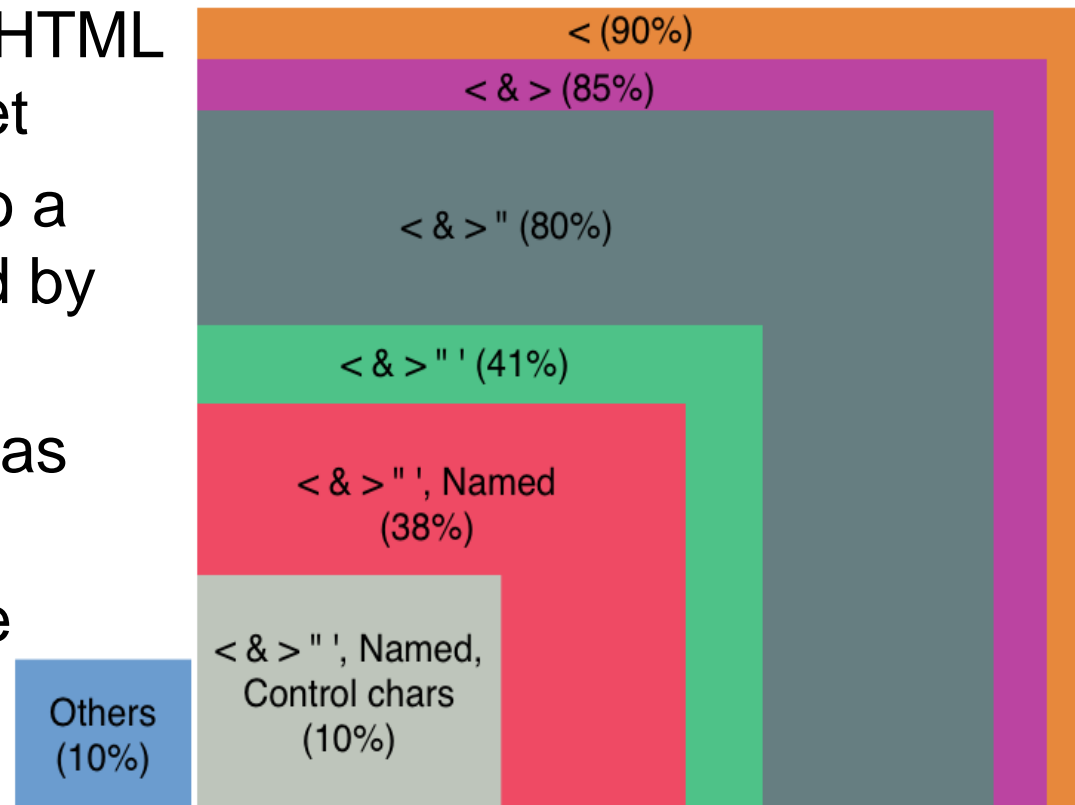
- ▶ Projects are using a lot of different HTML contexts, and it's not getting better
- ▶ “Don't insert user data in <insert HTML location here>”
- ▶ “Use auto-escaping template engines”

# Focusing on the HTML escaping...

- ▶ HTML escaping is important and can be correctly used in 50% of the injection sites
  - ▶ In addition, it makes safe 70% of them when implemented correctly (but makes the data incorrect)
- ▶ However, let's look at the different implementations of HTML escapers

# Dive into HTML escapers...

- ▶ HTML escaping should be simple
- ▶ We found **76** different HTML escapers in our dataset
- ▶ Most of them belong to a framework/library used by the application
- ▶ In average, a project has **5.7** HTML escapers
- ▶ All HTML escapers are not equivalent...



# Fix XSS

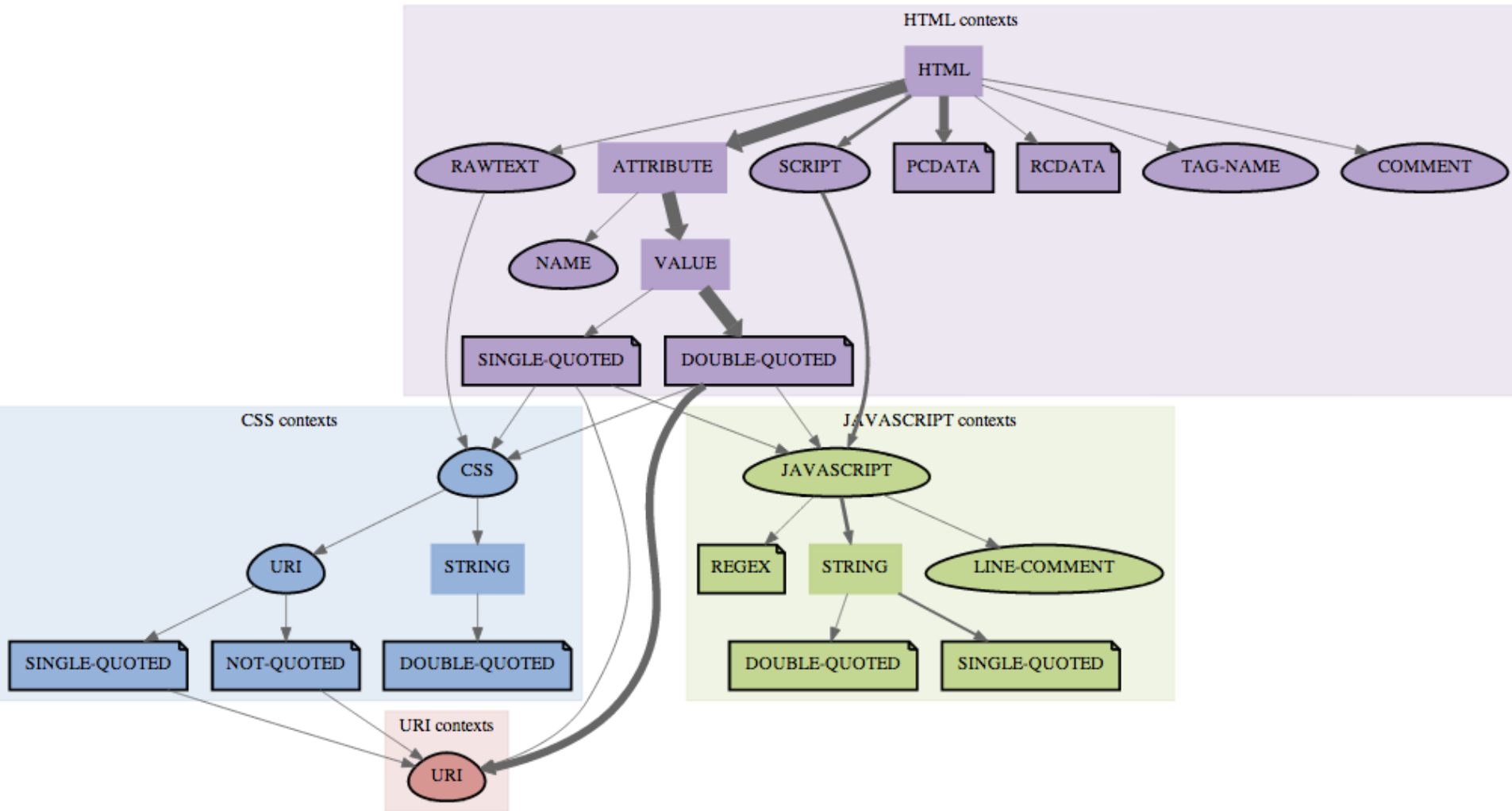
- ▶ Projects are using a lot of different HTML contexts, and it's not getting better
- ▶ HTML escaping seems to mean a lot of different things... Only 41% of the implementations seems sufficient
- ▶ “Don't insert user data in <insert HTML location here>”
- ▶ “Use auto-escaping template engines”

# “Don’t insert data into <?>”

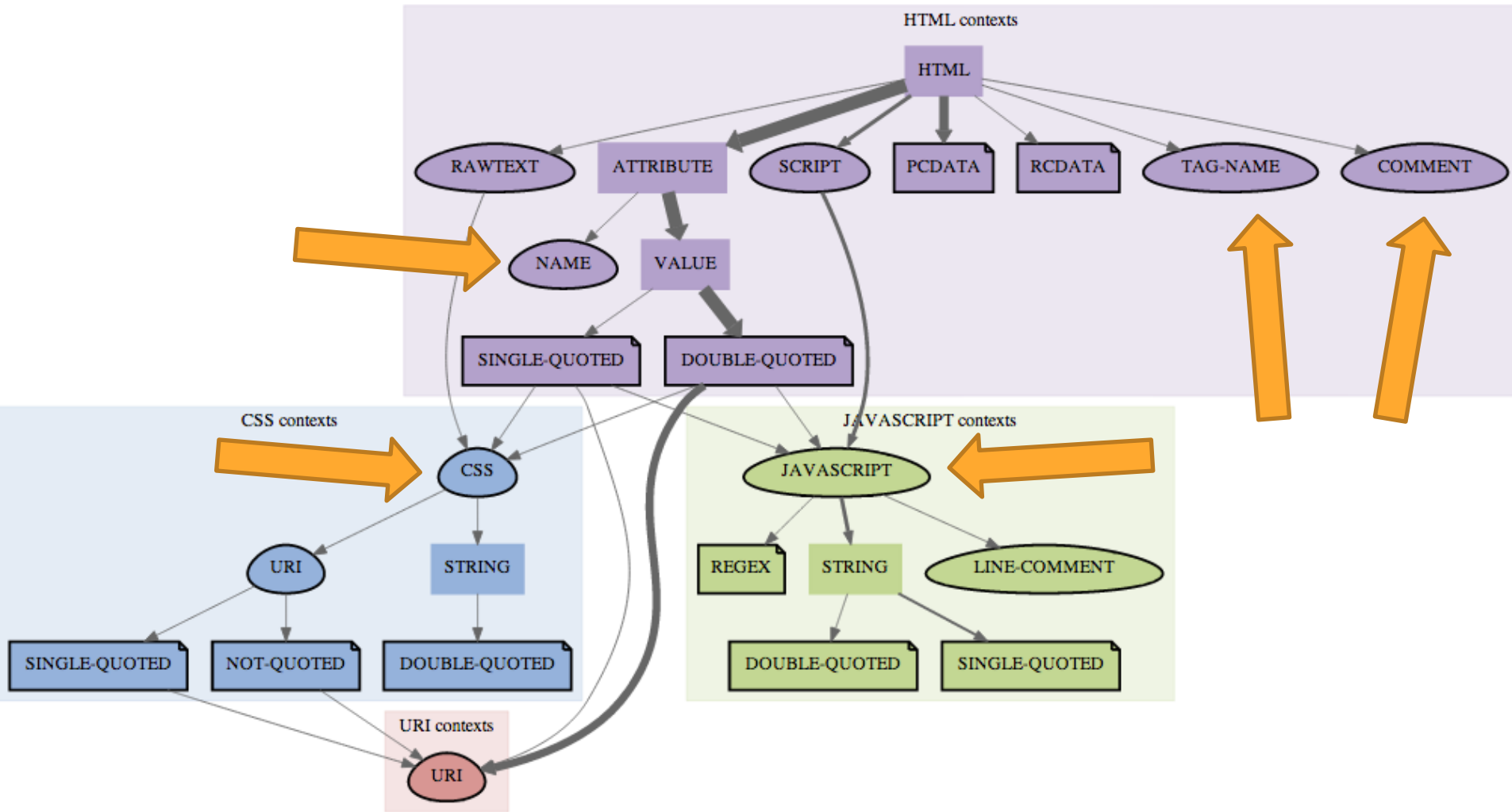
- ▶ We found that many security guidances just say “don’t do this”
- ▶ OWASP XSS cheat sheet [2] has the following rule:

Never put data HERE	“Context” name
<code>&lt;script&gt;...HERE...&lt;/script&gt;</code>	Directly in script
<code>&lt;!--...HERE...--&gt;</code>	HTML comment
<code>&lt;div ...HERE...=test /&gt;</code>	Attribute name
<code>&lt;HERE... href="/test" /&gt;</code>	Tag name
<code>&lt;style&gt;...HERE...&lt;/style&gt;</code>	Directly in CSS

# Observed HTML contexts and nesting



# Observed HTML contexts and nesting





- 
- ▶ Developers need to add data into many HTML contexts
  - ▶ “Use auto-escaping template engines”

# Auto-escaping template libraries

- ▶ A good trend with template engines is to provide auto-escaping.
- ▶ Auto-escaping means that the engine will take the dynamic data and escape it without any directive from the developer.
- ▶ Such reasoning is great and clearly makes a typical web application more secure.
- ▶ However, do they understand the contexts and provide the required escaping/filtering to be safe for XSS?

# Auto-escaping template libraries

Engine	Contextual Autoescaping	HTML Autoescaping	On By Default	Manual Escaping	Escapes
GCT	Yes	N/A	Yes	N/A	Yes
.NET Razor	No	Yes	Yes	Yes	Yes
Ruby on Rails	No	Yes	Yes	Yes	No
HAML	No	Yes	Yes	Yes	No
NHAML	No	Yes	Yes	Yes	No
Facelets	No	Yes	Yes	Yes	N/A
Mustache	No	Yes	Yes	Yes	No
Twig	No	Yes	No	Yes	Yes
Smarty	No	Yes	No	Yes	Yes
Spring	No	No	N/A	Yes	N/A
.NET WebForms	No	No	N/A	No	N/A
JSP	No	No	N/A	Yes	N/A

- ▶ Some don't even escape ' properly
- ▶ Only GCT provides a fairly good context aware auto-escaping, others essentially perform HTML escaping all the time

TML contexts

- ▶ Sometimes a false sense of security

# Sad truth about XSS

- ▶ I'm not saying that we have been all wrong until now. We just need to come up with a more complete solution for XSS:
  - ▶ Sadly, there is no one silver bullet for XSS
  - ▶ We cannot rely on one escaping function or filter
  - ▶ We cannot rely **yet** only on most auto-escaping template engines
- ▶ For the time being, we need:
  - ▶ Libraries that allow developer to escape the data properly for many contexts, and filter the data when an escaper cannot be used
  - ▶ Give actionable guidance to the developers

# Escapers should be available

- ▶ List of required escapers based on our dataset:
  - ▶ HTML escaping (^85%)
  - ▶ URI encoding (^30%)
  - ▶ JavaScript string escaping (^13%)
  - ▶ CSS string escaping (^0.16%)
  - ▶ JavaScript regex escaping (^0.01%)
- ▶ Libraries like OWASP ESAPI [3] or Coverity Security Library [4] already have these escapers; promote them!

# Filters should be available

- ▶ HTML contexts that require a filter based on our dataset:
  - ▶ HTML Attribute -> URI (^30%)
  - ▶ HTML Attribute -> JS code (0.2%)
  - ▶ HTML Attribute -> CSS (0.3%)
  - ▶ Etc.
- ▶ How to handle URIs?
  - ▶ Filter the scheme to make sure you allow it (http, https, etc.)
  - ▶ Make sure the authority makes sense for your application
  - ▶ URI encode each element of the path
  - ▶ URI encode query string parameter names and values
- ▶ ESAPI has a `isValidURL` which seems very restrictive, and `CSL.asURL` rewrites the URL to make it safe (but you need to manually encode the params & paths)

# XSS Conclusions

- ▶ XSS is complex because it can happen in many different locations in the HTML pages (HTML contexts).
- ▶ Input validation may not be adequate
- ▶ Sanitizers can help but it's very difficult to find a complete library for XSS
- ▶ OWASP XSS cheat sheet is a good start, but I believe we should do more:
  - ▶ We started blogging [5] about these kind of issues and will continue doing it as we improve our technologies or just come across “interesting stuff”
  - ▶ We will also continue to improve CSL [4] by adding filters, etc.
  - ▶ We need to be more serious about raising awareness of HTML contexts



Developers need  
some <3



# What's up with developers?!

- ▶ It's getting better. More and more developers have a basic understanding of security issues like XSS and SQLi.
  - ▶ For SQL, developers parameterized 94% of the time when possible
  - ▶ For HTML, we see many places where HTML escaping is used
- ▶ But it's just too complex and convoluted for them to understand everything
  - ▶ Security obligations of each context
  - ▶ Nested contexts and ordering of escapers
- ▶ There are gaps in the way security information is presented online; StackOverflow is a scary place. However, if you don't give them advice, who will?

# Ways forward

- ▶ Focus developer communication on the code point of view, as opposed to the attacker's point of view:
  - ▶ Attacks and threats are the "why" we need to fix
  - ▶ Developers need the "how" and "where" to fix
- ▶ Provide helpful advice:
  - ▶ Actionable (i.e. code, code, and more code)
  - ▶ To a specific technology (JSTL, Hibernate HQL, jQuery, etc.)
  - ▶ In a specific context or contexts (IN clause, URI, CSS string, etc.)
- ▶ What does helpful advice look like?
  - ▶ Not this: "Parameterized all SQL statements."
  - ▶ How about this?

# Helpful Advice

## How to Parameterize Data Values

Example: "SELECT \* FROM table WHERE name = '" + userName + "'"

JDBC

```
String paramQuery = "SELECT * FROM table WHERE name = ?";  
PreparedStatement prepStmt = connection.prepareStatement(paramQuery);  
prepStmt.setString(1, userName);
```

Hibernate  
Native  
Query

```
String paramQuery = "SELECT * FROM table WHERE name = :username";  
SQLQuery query = session.createSQLQuery(paramQuery);  
query.setParameter("username", userName);
```

JPA Native  
Query

```
String paramQuery = "SELECT * FROM table WHERE name = :username";  
Query query = entityManager.createNativeQuery(paramQuery);  
query.setParameter("username", userName);
```

# Helpful Advice

## How to add data in a CSS string within a CSS block using CSL.

Provide several examples that show the common usages for different HTML contexts, and technologies your developers are using

Expression  
Language

```
<style>
a[href *= "${cov:cssStringEscape(param.foobar)}"] {
    background-color: pink!important;
}
</style>
```

Java  
or  
JSP Scriptlet

```
<style>
<%
    String parm = request.getParameter("foobar");
    String cssEscaped = Escape.cssString(parm);
%>
a[href *= "<%= cssEscaped %>"] {
    background-color: pink!important;
}
</style>
```

# Conclusions

- ▶ XSS and SQLi can be nuanced and complex. XSS is actually becoming more and more complex over time as the number of contexts and nesting increases.
- ▶ Current advices we saw are either:
  - ▶ Generic remediation
  - ▶ Precisely how to perform the attack
- ▶ Guidance needs to be both more specific and technical, yet also simpler and shorter. The only way to get that is to take tailored, detailed advice and crop it down for the development team's needs.
- ▶ Remember: your developers need some <3

# Questions?



Security in knowledge

# References

- ▶ [1] [HTML 5 tokenizer specification](#)
- ▶ [2] [OWASP XSS Cheat Sheet](#)
- ▶ [3] [OWASP ESAPI Java](#)
- ▶ [4] [Coverity Security Library Java](#)
- ▶ [5] [Coverity Security Research Blog](#)
- ▶ [6] [OWASP SQLi Prevention Cheat Sheet](#)