Preventing Security Bugs Through Software Design

Christoph Kern, Google
If I had a dollar for every time someone writes an XSS...
Why so Many Bugs?

- Developer education doesn't solve the problem
  - Very large number of potentially vulnerable code sites
  - Security concerns orthogonal to primary developer focus
  - Sometimes quite subtle
- Bugs are hard to find after the fact
  - Complex, whole-system data-flows
- Low confidence in security assessment
Don't Blame the Dev, Blame the API
Preventing SQL Injection
String getAlbumsQuery = "SELECT ... WHERE " +
    " album_owner = " + session.getUserId() +
    " AND album_id = " + servletReq.getParameter("album_id");
ResultSet res = db.executeQuery(getAlbumsQuery);
Existing Best Practices

- **Prepared Statements**
  - Developers forget $\rightarrow$ potential bug
  - `dbConn.prepareStatement(`
    ```java
    "... WHERE foo = " + req.getParameter("foo")`;
    ```
  - (yes, not making this up)

- **Structural Query Builders**
  - Cumbersome for complex statements
A Simple, Safe Query API

- Desired: Query has no data-flow dependency on untrusted input
- Implied by: Query is concatenation of application-controlled strings
public class QueryBuilder {
    private StringBuilder query;

    /** ... Only call with compile-time-constant arg!!! ... */
    public QueryBuilder append(
            @CompileTimeConstant String sqlFragment) {
        ...
    }

    public String getQuery() { return query.build(); }
}
Preventing API Misuse

- Developers don't always read documentation
  - qb.append(
    "WHERE album_id = " + req.getParameter("album_id"));
- Enforce `@CompileTimeConstant` annotation via javac-integrated checker [github.com/google/error-prone, Aftandilian et al, SCAM '12]
- java/com/google/.../Queries.java:194: error: [CompileTimeConstant] Non-compile-time constant expression passed to parameter with `@CompileTimeConstant` type annotation.
  "WHERE album_id = " + req.getParameter("album_id"));
  ^
// Before
String sql = "SELECT ... FROM ...");
sql += "WHERE A.sharee = :user_id"

if (req.getParam("rating")!=null) {
    sql += " AND A.rating >= " +
    req.getParam("rating")
}

Query q = sess.createQuery(sql);
q.setParameter("user_id", ...);

// After
QueryBuilder qb = new QueryBuilder(
    "SELECT ... FROM ...");
qb.append("WHERE A.sharee = :user_id")
qb.setParameter("album_id", ...);

if (req.getParam("rating")!=null) {
    qb.append(" AND A.rating >= :rating")
    qb.setParameter("rating", ...);
}

Query q = qb.build(sess);
Practice

- Implemented inherently-safe Builder APIs for F1 [SIGMOD '12, VLDB '13] (C++, Java), Spanner [OSDI '12] (C++, Go, Java), and Hibernate.
- Refactored all existing call-sites across Google
  - Few person-quarters effort
- Removed `executeQuery(String)` methods
  - Hibernate: Errorprone checker to constrain Hibernate API use
- No more SQL injection!
Exceptional Use Cases

- E.g.: Command-line query tool
- Provide potentially-unsafe, unconstrained API
  - Subject to security review,
  - enforced using visibility whitelists [bazel.io/docs/build-encyclopedia.html#common.visibility]
  - Needed rarely (1-2% of call sites)
Preventing XSS
Ad-Hoc HTML Markup Creation

var escapedCat = goog.string.htmlEscape(category);
var jsEscapedCat = goog.string.escapeString(escapedCat);
catElem.innerHTML = '<a onclick="createCategoryList(\'' +
                jsEscapedCat + '\')">' + escapedCat + '</a>;

What if category == ");xssPayload();//"
Missing/Incorrect HTML Template Directives

{template .profilePage}

...  
<div class="name">{$profile.name}</div>
<div class="bloglink">
    <a href="{$profile.blogUrl}">
    ...
</a>

<div class="about">
    {$profile.aboutHtml |noAutoescape}

    ...
</div>

{/template}
Missing/Incorrect HTML Template Directives

{template .profilePage}
...
<div class="name">{$profile.name}</div>
<div class="bloglink">
  <a href="{$profile.blogUrl |sanitizeUrl}">...
  <div class="about">
    {$profile.aboutHtml |noAutoescape}
  </div>
  ...
</div>
{/template}
Complex, Whole-System Dataflows

Browser

```
{template .profilePage}
...
<div class="name">{$profile.name}</div>
<div class="bloglink">
  <a href="{$profile.blogUrl}">
    ...
  </a>
<div class="about">
  {$profile.aboutHtml |noAutoescape }
  ...
{/template}
```

```
profileElem.innerHTML =
  templates.profilePage(
    profile: rpcResponse.getProfile()
  );
```

Web-App Frontend

```
... profile =
  profileBackend.getProfile(
    currentUser);
...
rpcResponse.setProfile( profile);```

Application Backends

```
... profileStore->QueryByUser(
    user, &profile);
...```

Profile Store

(1)
Strictly Contextually Autoescaping Template Systems

- Template system infers correct context-sensitive sanitization/escaping [Samuel et al, CCS '13]
- No escaping directives/modifiers (the strict part)
- Recursive
Strict Contextual Template

```html
{template .profilePage autoescape="strict"}

   ...
   <div class="name">{$profile.name}</div>
   <div class="bloglink">
      <a href="{$profile.blogUrl}"...>
   <div class="about">
      {$profile.aboutHtml}
   </div>
   ...
{/template}
```
Strict Contextual Template

{template .profilePage autoescape="strict"}  
...  
<div class="name">${profile.name |escapeHtml}</div>  
<div class="bloglink">  
  <a href="${profile.blogUrl |sanitizeUrl|escapeHtml}">...</a>  
</div>  
<div class="about">  
  ${profile.aboutHtml |escapeHtml}  
  ...  
</div>  
{/template}
Types to Designate Safe Content

- Simple wrappers for string
- Context-specific type contracts
  - SafeHtml
  - SafeUrl
  - TrustedResourceUrl
  - SafeStyle
  - SafeStyleSheet
  - SafeScript
- Similar types in Google Web Toolkit, ca 2009.
Creating Safe-Content-Typed Values

- Inherently-Safe Builders/Producers
  - Structural builders
  - Strict template evaluation

- Unchecked Conversions
  - Subject to security review (BUILD-visibility)
  - Guidelines on appropriate use -- reviewability & local reasoning
Disallow Injection-Prone Sinks

- `.innerHTML`, server-side responses, etc.
- **Static enforcement**
  - Javascript conformance pass in Closure Compiler
  - Errorprone
  - Reviewed white-lists
Putting it all Together

Browser

```html
{template .profilePage autoescape="strict"}
...
<div class="name">{$profile.name}</div>
<div class="bloglink">
  <a href="{$profile.blogUrl}">
    ...
  </a>
</div>
</template>
```

```javascript
renderer.renderElement(profileElem, templates.profilePage, {
  profile: rpcResponse.getProfile()
});
```

Web-App Frontend

```javascript
profile = profileBackend.getProfile(currentUser);
rpcResponse.setProfile(profile);
```

Application Backends

```javascript
profileStore->QueryByUser(user, &lookup_result);
SafeHtml about_html = html_sanitizer->sanitize(lookup_result.about_html_unsafe());
profile.set_about_html(about_html);
```

Profile Store

```javascript
HtmlSanitizer::SafeHtml(sanitized);
```
Practical Application

- Strict contextual escaping in Closure Templates et al.
- Adopted in several flagship Google applications
- Drastic reduction in bugs
  - One case: ~30 XSS in 2011, None (*) since Sep 2013
- More background: [Kern, CACM 9/’14]
Caveats/Limitations

- **Type system**
  - Reflection, casts, loose visibility
  - But: Idiomatic usage patterns matter!

- **No formal guarantees**
  - Correctness properties ultimately based on human reasoning
  - But: By design, local reasoning, and drastically reduced scope
  - But: In practice, most bugs found in application code

- **Pathological uses:** Control-flow dep. effectively implies Data-flow dep.
  - But: Threat model -- Non-malicious programmer
Lessons Learnt
It's OK to change code!
Strings are Bad
Unless Proven Otherwise
Types
Simple Static Checks
Don't Track "Taint", Make or Track "Safe"
Simple, Safe, Familiar-ish APIs (>98%)
Review-Gated Unsafe API (<2%)
Build on Existing Tooling
Benefits
(Potentially) Vulnerable Code never even Written/Checked-in
Confines Bug Potential into Very Small Portion of Codebase
Drastic Reduction in Bugs Observed
Drastic Reduction in Review Burden
Increased Confidence in Correctness
It's all about API Design
Questions?