Preventing Security Bugs Through Software Design

Christoph Kern, Google
xtof@google.com
The Same Bugs, Over and Over Again...

- SQL-injection, XSS, XSRF, etc -- OWASP Top 10
- Root cause
  - Inherently bug-prone APIs
  - Developers are humans and make mistakes
  - APIs often widely used
- Many potential bugs
  → Some actual bugs
- Inherently incomplete bug-finding approaches (testing, static analysis)
  → Once introduced, bugs are difficult to eliminate
Don't Blame the Developer, Blame the API
Inherently Safe APIs

- API design prevents introduction of security bugs in application code
- Approx. as convenient to use as original, vuln-prone API
- Soo... Is this practical?
Preventing SQL Injection
SQL Injection

```java
String getAlbumsQuery = "SELECT ... WHERE " +
    " album_owner = " + session.getUserId() +
    " AND album_id = " + servletReq.getParameter("album_id");
ResultSet res = db.executeQuery(getAlbumsQuery);
```
Existing Best Practices

- "Use Prepared Statements"
  - Developers forget → potential bug
  - `dbConn.prepareStatement("... WHERE foo = " + req.getParameter("foo");`
  - (yes, not making this up)

- "Use Structural Query Builder APIs"
  - Cumbersome for complex statements
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();; }
}
Static Check of API Contract

qb.append(
    "WHERE album_id = " + req.getParameter("album_id"));

--> 
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"));
   ^

[github.com/google/error-prone, Aftandilian et al, SCAM '12]
// Unsafe API
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", ...);

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

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

Query q = qb.build(sess);
In 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 (*) bug-prone `executeQuery(String)` methods ⇒ SQL Injection doesn't even compile
- Straightforward implementation
(*) Exceptional Use Cases

- E.g.: Command-line query tool
- Provide potentially-unsafe, unconstrained API
  - "Back door" for `executeQuery(String)`
  - 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
void showProfile(el, profile) {
    // ...
    profHtml += "<a href='" + htmlEscape(profile.homePage) + '">");
    // ...
    profHtml += "<div class='about'> + profile.aboutHtml + "</div>");
    // ...
    el.innerHTML = profHtml;
}

void showProfile(el, profileElement, rpcResponse) {
    // ...
    profileElelement = rpcResponse.getProfile();
    // ...
    profHtml += "<a href='" + htmlEscape(profile.homePage) + '");
    // ...
    profHtml += "<div class='about'> + profile.aboutHtml + "</div>");
    // ...
    el.innerHTML = profHtml;
}

message ProfileProto {
    optional string name = 1;
    optional string home_page = 2;
    optional string about_html = 3;
}
Safe HTML Rendering: Strict Contextual Autoescaping

```
{template .profilePage autoescape="strict"}
...
<div class="name">{$profile.name}</div>
<div class="homepage">
  <a href="{$profile.homePage}">...
  <div class="about">
    {$profile.aboutHtml}
  </div>
  ...
</div>
{/template}
```
Safe HTML Rendering: Strict Contextual Autoescaping

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

... 
<div class="name">{$profile.name |escapeHtml} </div>
<div class="homepage">
  <a href="{$profile.homePage |sanitizeUrl|escapeHtml}">...
  <div class="about">
    {$profile.aboutHtml |escapeHtml} 
    ...
  </div>

{/template}
```
Types to Designate Safe Content

- Context-specific types
  - SafeHtml
  - SafeUrl
  - ...
- Security type contracts
  - "Safe to use (wrt XSS) in corresponding HTML context"
  - Contract ensured by types' public API (builders/factory-functions)
  - "Unchecked Conversions" – mandatory security review
Coding Rules

- Use strict template for all HTML markup
- Bug-prone DOM-APIs (.innerHTML, location.href, etc) strictly forbidden in application code
- Enforced by compile-time check (Closure JS Conformance)
- Errors reference safe alternatives
  - .innerHTML -> strict template; goog.dom.safe.setInnerHTML(Element, SafeHtml)
  - location.href -> goog.dom.safe.setLocationHref(Location, string|SafeUrl)
  - etc
Safe-Coding-Conformant Application Code

Browser

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

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

Web-App Frontend

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

```
message ProfileProto {
  optional string name = 1;
  optional string home_page = 2;
  optional SafeHtmlProto about_html = 3;
}
```

Application Backends

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

```
...return UncheckedConversions::SafeHtml(sanitized);
```

Profile Store

```
Profile
Profile
Profile
HtmlSanitizer
```
Practical Application

- Strict contextual escaping in Closure Templates, AngularJS, et al.
- Adopted in several flagship Google applications
- Drastic reduction in bugs
  - One case: ~30 XSS in 2011, ~0 (*) since Sep 2013
- More background: [Kern, CACM 9/’14]
Design Patterns
Inherently Safe APIs: Confining/Eliminating "Bug Potential"

- Inherently-safe API: By design, calling code can't have (certain types of) bugs
  - Potential for bugs (of specific class) confined in API's implementation
  - Potential for bugs eliminated from application code

- In practice: Reduction in actual bugs
API Design Principle: No Unsupported Assumptions

- Values of basic types (esp String):
  - (conservatively) assumed attacker-controlled
    - Unconditionally apply run-time escaping/validation
    - In practice, almost always functionally correct
- Dual of static/dynamic taint tracking
  - "Strings are evil, unless proven otherwise"
  - vs. "Strings are harmless unless tainted"
Types & Type Contracts

- Type Contract "tele-ports" promise about a value from source to sink
  - Irrespective of complexity of intervening (whole-system) data flow
- Enable localized reasoning about whole-program correctness
- Modulo type integrity
  - Assumes reasonably rigorous type encapsulation
Usability & Practical Applicability

- Similarity to familiar APIs and coding patterns
  - Some refactoring OK, esp if automatable
- Lightweight approval process for exceptions
- Errors and findings: Compile time is best time
  - Before changelist is even sent for review
  - Clear-cut "deviation from safe coding practice" error vs. ambiguous "potential bug" finding
  - Familiar presentation (type error, straightforward static check)
Design for Reviewability

- "What percentage of source code do I have to read/analyze/understand to establish absence of a class of bugs?"
  - > yy% + large project → you'll almost certainly have some bugs
  - < 0.xx% → probably in good shape

- Inherently Safe APIs & confined bug potential
  ⇒ Drastically reduced review burden
  ⇒ Comprehensive reviews become practical
  ⇒ High-confidence assessments
Open Source

- Closure SafeHtml types & DOM wrappers
- Closure Templates Strict Contextual Escaping
- Closure Compiler Conformance
- AngularJS Strict Contextual Escaping
- @CompileTimeConstant checker (part of Error Prone)
- Coming soon: Just released: Java Safe HTML types
Questions?