Robuste und Praktikable Ansätze zur Verhinderung von Sicherheitsdefekten

Christoph Kern, Google

Weit verbreitete Sicherheitslücken

- SQL-injection, XSS, XSRF, etc -- OWASP Top 10
- Grundproblem:
  - APIs/Frameworks erlauben/ermöglichen Enführung von Sicherheitsdefekten
  - Weitläufig verwendete APIs
  - Fehler sind menschlich und daher unvermeidlich
- Einmal eingeführte Sicherheitslücken umfassend zu eliminieren -- in der Praxis schwierig

Ansatz

Vermeidung von Sicherheitslücken ist
Verantwortung des API-Designers,
nicht des Anwendungsentwicklers

SQL Injection

SQL Injection

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

Sicheres API

```
public class QueryBuilder {
  private StringBuilder query;

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

  public String getQuery() { return query.build(); }
}
```

Statischer Check des API-Kontrakts

```
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]

APIs im Vergleich

```
// Vorher
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", ...);
```

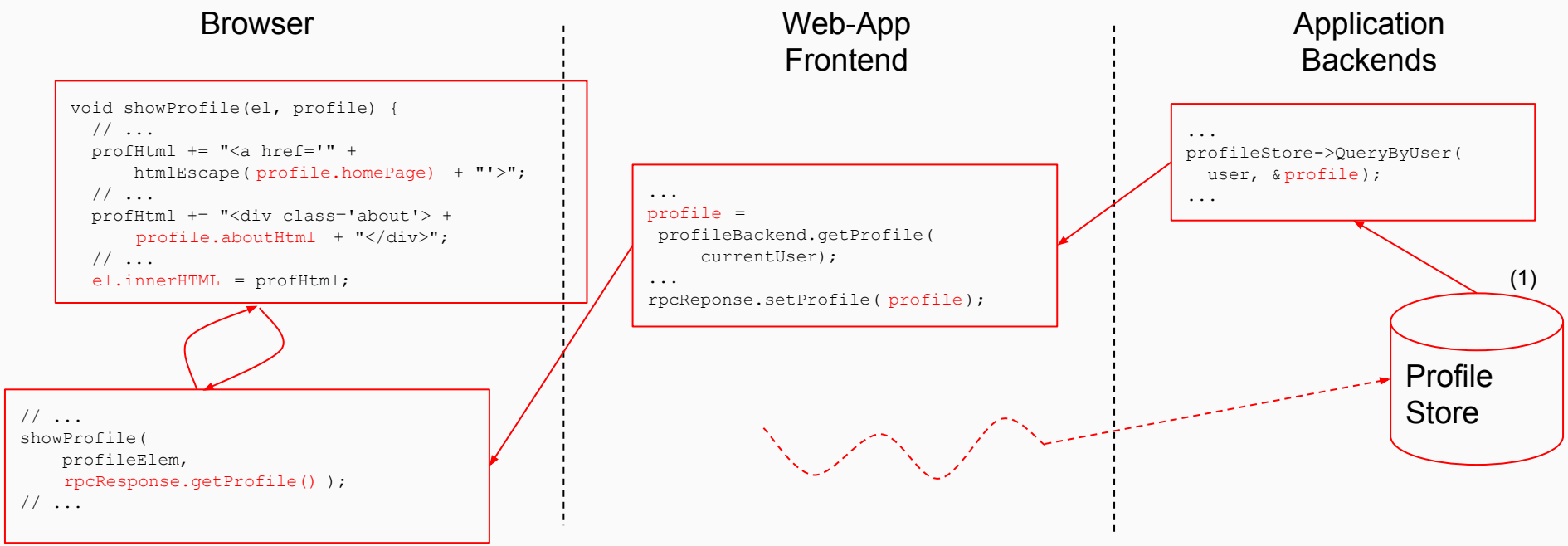
```
// Nachher
QueryBuilder qb = new QueryBuilder(
    "SELECT ... FROM ...");
qb.append("WHERE A.sharee = :user_id");
qb.setParameter("album_id", ...);

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

Query q = qb.build(sess);
```

Praxis

- Sichere QueryBuilder APIs für F1 [SIGMOD '12, VLDB '13] (C++, Java), Spanner [OSDI '12] (C++, Go, Java), and Hibernate
- Google-weite Anpassung der Quellencodebasis
  - Aufwand: 2 Entwickler für 2-3 Quartale
- Fehleranfällige `executeQuery(String)` Methoden aus API entfernt(*)
- SQL-Injection im Prinzip unmöglich -- potentielle Sicherheitslücke führt zu Kompilierungsfehler
- (*) In bestimmten Modulen erlaubt, vorbeh. Sicherheitsanalyse

XSS

Typische Webapplikation



Striktes Kontext-Sensitives HTML Template

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

Striktes Kontext-Sensitives HTML Template

```
{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}
  …
{/template}
```

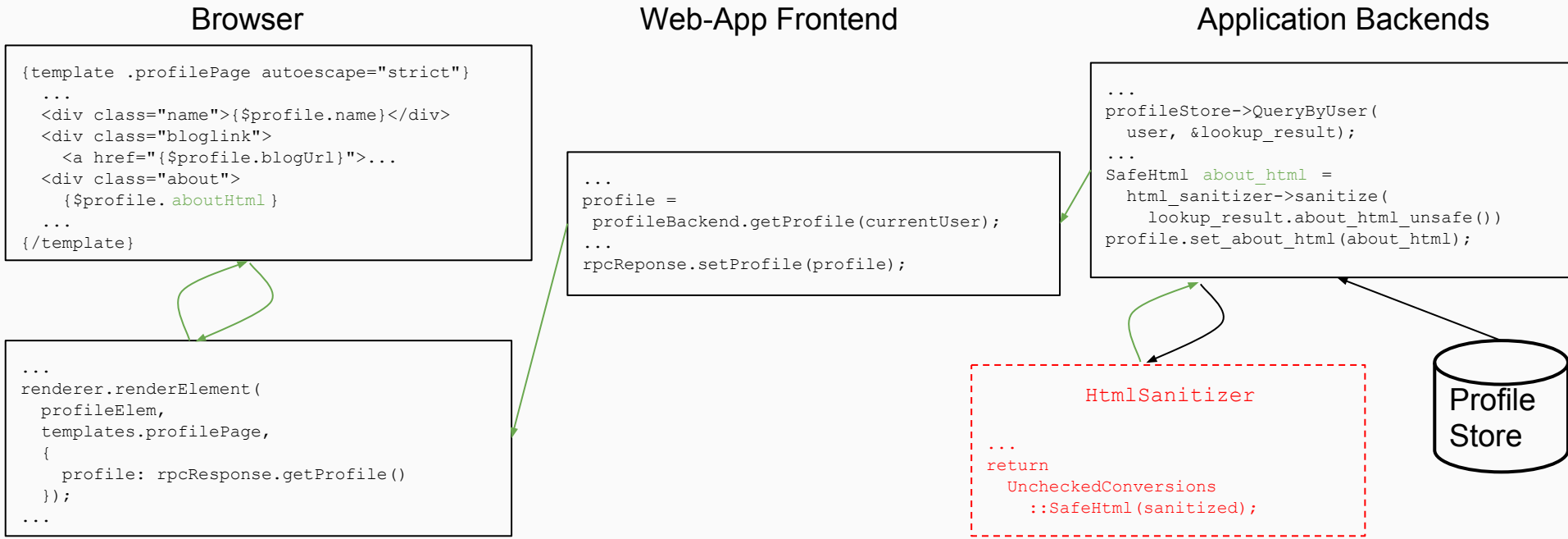
Kontext-Spezifische Datentypen

- Kontext-spezifische Datentypen
  - SafeHtml
  - SafeUrl
  - ...
- Kontrakt
  - Datentyp kann im entsprechenden Kontext ohne XSS-Risiko verwendet werden
  - Kontrakt durch Factory-Functions aufrecht erhalten
  - "Unchecked Conversions" -- obligatorische Sicherheitsüberprüfung

Strikte Sicherheitsrichtlinie

- Fehleranfällige DOM-APIs (.innerHTML, location.href, etc) in Anwendungsquellcode strikt verboten
- Statischer Check zur Kompilierungszeit (Closure JS Conformance)
- Verweis auf sichere Alternativen
  - .innerHTML -> Striktes Template; goog.dom.safe.setInnerHtml(Element, SafeHtml)
  - location.href -> goog.dom.safe.setLocationHref(Location, string|SafeUrl)
  - etc

Richtlinien-Konformer Code



Praxis

- Strikte kontext-sensitive Validierung implementiert in Closure Templates, AngularJS, etc.
- Anwendung in komplexen Google web apps (GMail, G+, etc)
- Drastische Reduzierung der Sicherheitslücken
  - **~30 XSS** in 2012, **Keine** (in Anwendungs-Quellencode) seit Sept. 2013
- [Kern, CACM 9/'14]

Einschränkungen

- Umgehung der Richtlinien mittels kreativer Programmierkonstrukte
  - Reflection, casts, etc
- Keine Formale Methoden
  - Informelle Datentypen-Kontrakte
  - Informelle Analyse
  - Wichtig: **Modul-lokale** Analyse

Zusammenfassung

APIs &
Strikte Entwicklungsrichtlinien ...

… verhindern spezifische Sicherheitslücken "by Design"

Datentypen & Kontrakte

(Unkomplizierte) Statische Checks

Notwendigkeit manueller Code-Analyse auf spezifische Module eingeschränkt

API-Design FTW

Fragen?