Secure Design: A Better Bug Repellent

Christoph Kern, IEEE SecDev '17
Security Defects
Implementation Bugs

- Shallow & Localized
- Patchable
- Straightforward & Testable
Design Flaws

Deep & Diffuse

Complex & Costly

Subtle

Image credit: Vaniato/Shutterstock.com
Implementation Bugs

- Shallow & Localized
- Patchable
- Straightforward & Testable
- Ubiquitous & Recurrent

Design Flaws

- Deep & Diffuse
- Complex & Costly
- Subtle
- Remediable
Assurance
Essence of a Bug: Violated Precondition

- Potentially-vulnerable API or language primitive
- Precondition: Predicate on program state at call-site
  - `void *memcpy(void *dest, const void *src, size_t n)`
    requires `valid_buf(dest, n) \land valid_buf(src, n) \land \neg overlaps(dest, src, n)`
  - `*p`
    requires `valid_buf(p, sizeof(*p))`
  - `sql_query(db: DBConn, q: string) returns ResultSet`
    requires `trusted_fx_sql(q)`
  - `Element.setInnerHTML(s: string)`
    requires `safe_fx_html(s)`
Absence of Evidence ≠ Evidence of Absence
Demonstrating Absence of Bugs

To show:

"For all call sites, for all reachable program states, precondition holds at call site"

```javascript
var byline = 'by <i>' + escape_html(user_nick) + '</i>);
assert safe_fx_html(byline); // precondition
headerElem.innerHTML = byline;

var byline = 'by <a href=' + user_profile_url + '>' + user_nick + '</a>);
assert safe_fx_html(byline); // precondition
headerElem.innerHTML = byline;
```
function renderPost(p) {
  ...
  byEl.innerHTML = 'by <a href=...>'
    + p.by + '</a>'
  }

function onUpdate(posts) {
  ...
  renderPost(post);
}

function onXhrResp(rpc) {
  ...
  onUpdate(rpc.resp().posts());
}

Abc buildAbc(Xyz xyz) {
  ...
}

Xyz getXyz(...) {
  ...
  abcBackend.getXyz(rpc, p)
}

Abc buildAbc(Xyz xyz) {
  ...
}

Status storeXyz(const Xyz& xyz) {
  ...
  db->write(...)
}

func putXyz(...) err {
  ...
  err=abcRe.putXyz(rpc, p)
}

Abc buildAbc(Xyz xyz) {
  ...
}
Non-Scaleable Process

- Large & complex relevant program/system slices
- Complex, non-automatable reasoning

\[ \text{safe}_\text{fx}_\text{html}(s) \equiv "s, parsed and evaluated as HTML markup, only has safe (side) effects" \]

- Exact meaning of \textit{safe}?  
- Undecidable post-conditions

- Moving target: \textbf{The bugs just keep on coming!}
Scaleable Implementation Security
Design Goals

**Local Reasoning**: Preconditions established by surrounding code

**Scalable & Mandatory Expert Review**

- Avoid need for expert reasoning about preconditions all throughout application code
- Confine security-relevant program slice to expert-owned/reviewed source
Inherently-safe (Precondition-free) APIs

- Public (wrapper) API without (security) preconditions

  ```javascript
  goog.dom.safe.setLocationHref = function(loc, url) {
    loc.href = isSafeSchemeOrRelativeUrl(url) ? url : 'about:invalid';
  }
  ```

- Disallow use of "raw" API via static checks / lint / presubmit-hook
Types to "Teleport" Assertions

- Type contract captures asserted predicate on value

  \[ \forall v: v \text{ instanceof SafeHtml} \Rightarrow \text{safeFxHtml}(v\text{.toString}()) \]

- Type contract ensured by public builders/c'tors

  SafeHtml safeHtml = new SafeHtmlBuilder("div").escapeAndAppendContent(s).build()

  SafeHtml safeHtml = htmlSanitizer.sanitize(untrustedHtml)

- APIs may rely on type contract

  goog.dom.safe.setInnerHtml = function(el, html) {
      el.innerHTML = SafeHtml.unwrap(html);
  }

  new SafeHtmlBuilder("div").appendContent(safeHtml)
Global Properties

from

Local Reasoning + Types
Application @Google
Preventing SQL Injection Vulnerabilities

- **TrustedSqlString type and builders**
  
  \[
  \text{TrustedSqlString} \equiv \text{compile-time-constants and concatenations thereof}
  \]

- **Compile-time-constant expressions constraint**
  - Go, C++: Natively expressible
  - Java: custom static check (based on Error-prone framework [1])
    
    ```java
    public TrustedSqlStringBuilder append(@CompileTimeConstant final String sql)
    ```

- **Query APIs (Spanner [2], F1 [3]) require TrustedSqlString**
Compile-Time Security

- API ensures:
  SQL queries have no data-flow dependency on untrusted input
- Encodes best practice ("always use bind parameters")
- Potential vulnerability → compilation error

```java
trSqlBuilder.append("WHERE thing_id = " + thingId));
```

```
java/com/google/.../Queries.java:194: error: [CompileTimeConstant] Non-compile-time constant expression passed to parameter with @CompileTimeConstant type annotation.
  "WHERE thing_id = " + thingId);
  ^
```
Developer Ergonomics

// Ad-hoc, unsafe & vulnerable code

String sql = "SELECT ... FROM ...";
sql += "WHERE A.sharee = @user_id";

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

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

// Safe QueryBuilder

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 = db.newQuery(qb);
There are always exceptions...

- Command-line utilities, admin UIs, ...
- Accommodating Exceptions:
  - "Unchecked conversion" from String to TrustedSqlString
  - Subject to security review
Preventing XSS Vulnerabilities [4]

- **Types:** SafeHtml, SafeUrl, TrustedResourceUrl, ...

- **Builders & Factories:**
  
  ```java
  SafeHtml linkHtml = new SafeHtmlBuilder("a")
  .setTarget(TargetValue.BLANK)
  .setHref(SafeUrls.sanitize(profileUrl))
  .escapeAndAppendContent(profileLinkText)
  .build()
  ```

- **Strictly contextually** Auto-escaping HTML Template Systems: Closure (aka Soy), Angular, Polymer (Resin), GWT, and proprietary frameworks

- **Type-safe DOM API Wrappers** (`goog.dom.safe.setInnerHTML`, `setLocationHref`, ...)

- **Static check (JS Conformance)** disallows XSS-prone DOM APIs (`el.innerHTML = v`)
Eradicating (*) XSS

- Adopted by flagship Google projects (GMail, G+, Identity Frontends, ...), and underlying frameworks
- Significant reduction in bugs (10s → ~0)
- Reasonable effort
  - Legacy code: Significant one time refactoring effort
  - New code: Straightforward/seamless

(*) almost...
Experience & Observations

- Scalable process
  - Team of ~5 security engineers/developers/maintainers
  - ~1 security engineer (weekly rotation) supporting ...
  - ... usage across entire Google codebase (100s if not 1000s devs)
- "Design for Reviewability"
- Developer ergonomics
- Toolchain integration
  - Type checker + few custom static checks
  - Single source repo [5]
  - Large-scale refactoring [6,7]
- Mandatory security reviews ("unchecked conversions")
  - Bazel BUILD rule visibility
Potential(*) for Security Bugs is a Security Design Flaw

(*)Widespread, throughout application code
Appendix
Open Source

- Closure SafeHtml types & DOM wrappers
- Closure Compiler Conformance
- Closure Templates Strict Contextual Escaping
- AngularJS Strict Contextual Escaping
- Polymer Strict Contextual Escaping (aka Resin)
- Safe HTML in Google Web Toolkit (GWT)
- @CompileTimeConstant checker (part of Error Prone)
- Java Safe HTML types
- Bazel build system (supports rule visibility to constrain usage)
References


