

Preventing script injection vulnerabilities through software design.

BY CHRISTOPH KERN

Securing the Tangled Web

SCRIPT INJECTION VULNERABILITIES are a bane of Web application development: deceptively simple in cause and remedy, they are nevertheless surprisingly difficult to prevent in large-scale Web development.

Cross-site scripting (XSS)^{2,7,8} arises when insufficient data validation, sanitization, or escaping within a Web application allow an attacker to cause browser-side



execution of malicious JavaScript in the application's context. This injected code can then do whatever the attacker wants, using the privileges of the victim. Exploitation of XSS bugs results in complete (though not necessarily persistent) compromise of the victim's session with the vulnerable application. This article provides an overview of how XSS vulnerabilities arise and why it is so difficult to avoid them in real-world Web application software development. Software design patterns developed at Google to address the problem are then described.

A key goal of these design patterns



is to confine the potential for XSS bugs to a small fraction of an application's code base, significantly improving one's ability to reason about the absence of this class of security bugs. In several software projects within Google, this approach has resulted in a substantial reduction in the incidence of XSS vulnerabilities.

Most commonly, XSS vulnerabilities result from insufficiently validating, sanitizing, or escaping strings that are derived from an *untrusted source* and passed along to a *sink* that interprets them in a way that may result in script execution.

Common sources of untrustworthy data include HTTP request parameters, as well as user-controlled data located in persistent data stores. Strings are often concatenated with or interpolated into larger strings before assignment to a sink. The most frequently encountered sinks relevant to XSS vulnerabilities are those that interpret the assigned value as HTML markup, which includes server-side HTTP responses of MIME-type `text/html`, and the `Element.prototype.innerHTML` Document Object Model (DOM)⁸ property in browser-side JavaScript code.

Figure 1a shows a slice of vulner-

able code from a hypothetical photo-sharing application. Like many modern Web applications, much of its user-interface logic is implemented in browser-side JavaScript code, but the observations made in this article transfer readily to applications whose UI is implemented via traditional server-side HTML rendering.

In code snippet (1) in the figure, the application generates HTML markup for a notification to be shown to a user when another user invites the former to view a photo album. The generated markup is assigned to the `innerHTML` property of a DOM

A Subtle XSS Bug

The following code snippet intends to populate a DOM element with markup for a hyperlink (an HTML anchor element):

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

The anchor element's click-event handler, which is invoked by the browser when a user clicks on this UI element, is set up to call a JavaScript function with the value of category as an argument. Before interpolation into the HTML markup, the value of category is HTML-escaped using an escaping function from the JavaScript Closure Library. Furthermore, it is JavaScript-string-literal-escaped (replacing ' with \' and so forth) before interpolation into the string literal within the onclick handler's JavaScript expression. As intended, for a value of Flowers & Plants for variable category, the resulting HTML markup is:

```
<a onclick="createCategoryList('Flowers & Plants')"
    Flowers & Plants</a>
```

So where's the bug? Consider a value for category of:

```
');attackScript()//
```

Passing this value through htmlEscape results in:

```
&#39;);attackScript()//
```

because htmlEscape escapes the single quote into an HTML character reference. After this, JavaScript-string-literal escaping is a no-op, since the single quote at the beginning of the page is already HTML-escaped. As such, the resulting markup becomes:

```
<a onclick="createCategoryList('&#39;);attackScript()//')">
    &#39;);attackScript()//</a>
```

When evaluating this markup, a browser will first HTML-unescape the value of the onclick attribute before evaluation as a JavaScript expression. Hence, the JavaScript expression that is evaluated results in execution of the attacker's script:

```
createCategoryList('');attackScript()//')
```

Thus, the underlying bug is quite subtle: the programmer invoked the appropriate escaping functions, but in the wrong order.

element (a node in the hierarchical object representation of UI elements in a browser window), resulting in its evaluation and rendering.

The notification contains the album's title, chosen by the second user. A malicious user can create an album titled:

```
<script>attackScript;</script>
```

Since no escaping or validation is applied, this attacker-chosen HTML is interpolated as-is into the markup generated in code snippet (1). This markup is assigned to the innerHTML sink, and hence evaluated in the context of the victim's session, executing the attacker-chosen JavaScript code.

To fix this bug, the album's title must be *HTML-escaped* before use in markup, ensuring that it is interpret-

ed as plain text, not markup. HTML-escaping replaces HTML metacharacters such as <, >, ", ', and & with corresponding character entity references or numeric character references: <, >, ", ', and &. The result will then be parsed as a substring in a text node or attribute value and will not introduce element or attribute boundaries.

As noted, most data flows with a potential for XSS are into sinks that interpret data as HTML markup. But other types of sinks can result in XSS bugs as well: Figure 1b shows another slice of the previously mentioned photo-sharing application, responsible for navigating the user interface after a login operation. After a fresh login, the app navigates to a preconfigured URL for the application's

main page. If the login resulted from a session time-out, however, the app navigates back to the URL the user had visited before the time-out. Using a common technique for short-term state storage in Web applications, this URL is encoded in a parameter of the current URL.

The page navigation is implemented via assignment to the window.location.href DOM property, which browsers interpret as instruction to navigate the current window to the provided URL. Unfortunately, navigating a browser to a URL of the form javascript:attackScript causes execution of the URL's body as JavaScript. In this scenario, the target URL is extracted from a parameter of the *current* URL, which is generally under attacker control (a malicious page visited by a victim can instruct the browser to navigate to an attacker-chosen URL).

Thus, this code is also vulnerable to XSS. To fix the bug, it is necessary to validate that the URL will not result in script execution when dereferenced, by ensuring that its scheme is benign—for example, https://

Why Is XSS So Difficult to Avoid?

Avoiding the introduction of XSS into nontrivial applications is a difficult problem in practice: XSS remains among the top vulnerabilities in Web applications, according to the Open Web Application Security Project (OWASP);⁴ within Google it is the most common class of Web application vulnerabilities among those reported under Google's Vulnerability Reward Program (<https://goo.gl/82zcPK>).

Traditionally, advice (including my own) on how to prevent XSS has largely focused on:

- ▶ Training developers how to treat (by sanitization, validation, and/or escaping) untrustworthy values interpolated into HTML markup.^{2,5}
- ▶ Security-reviewing and/or testing code for adherence to such guidance.

In our experience at Google, this approach certainly helps reduce the incidence of XSS, but for even moderately complex Web applications, it does not prevent introduction of XSS to a reasonably high degree of confidence. We see a combination of factors leading to this situation.

Subtle security considerations. As seen, the requirements for secure handling of an untrustworthy value depend on the context in which the value is used. The most commonly encountered context is string interpolation within the content of HTML markup elements; here, simple HTML-escaping suffices to prevent XSS bugs. Several special contexts, however, apply to various DOM elements and within certain kinds of markup, where embedded strings are interpreted as URLs, Cascading Style Sheets (CSS) expressions, or JavaScript code. To avoid XSS bugs, each of these contexts requires specific validation or escaping, or a combination of the two.^{2,5} The accompanying sidebar, “A Subtle XSS Bug,” shows this can be quite tricky to get right.

Complex, difficult-to-reason-about data flows. Recall that XSS arises from flows of untrustworthy, unvalidated/escaped data into injection-prone sinks. To assert the absence of XSS bugs in an application, a security reviewer must first find all such data sinks, and then inspect the surrounding code for context-appropriate validation and escaping of data transferred to the sink. When encountering an assignment that lacks validation and escaping, the reviewer must backward-trace this data flow until one of the following situations can be determined:

- ▶ The value is entirely under application control and hence cannot result in attacker-controlled injection.
- ▶ The value is validated, escaped, or otherwise safely constructed somewhere along the way.
- ▶ The value is in fact not correctly validated and escaped, and an XSS vulnerability is likely present.

Let’s inspect the data flow into the `innerHTML` sink in code snippet (1) in Figure 1a. For illustration purposes, code snippets and data flows that require investigation are shown in red. Since no escaping is applied to `sharedAlbum.title`, we trace its origin to the `albums` entity (4) in persistent storage, via Web front-end code (2). This is, however, not the data’s ultimate origin—the album name was previously entered by a different user (that is, originated in a different time context). Since no escaping was applied to this value anywhere along its flow from

The primary goal of this approach is to limit code that could potentially give rise to XSS vulnerabilities to a very small fraction of an application’s code base.

an ultimately untrusted source, an XSS vulnerability arises.

Similar considerations apply to the data flows in Figure 1b: no validation occurs immediately prior to the assignment to `window.location.href` in (5), so back-tracing is necessary. In code snippet (6), the code exploration branches: in the true branch, the value originates in a configuration entity in the data store (3) via the Web front end (8); this value can be assumed application-controlled and trustworthy and is safe to use without further validation. It is noteworthy that the persistent storage contains both trustworthy and untrustworthy data in different entities of the same schema—no blanket assumptions can be made about the provenance of stored data.

In the `else`-branch, the URL originates from a parameter of the *current* URL, obtained from `window.location.href`, which is an attacker-controlled source (7). Since there is no validation, this code path results in an XSS vulnerability.

Many opportunities for mistakes.

Figures 1a and 1b show only two small slices of a hypothetical Web application. In reality, a large, nontrivial Web application will have hundreds if not thousands of branching and merging data flows into injection-prone sinks. Each such flow can potentially result in an XSS bug if a developer makes a mistake related to validation or escaping.

Exploring all these data flows and asserting absence of XSS is a monumental task for a security reviewer, especially considering an ever-changing code base of a project under active development. Automated tools that employ heuristics to statically analyze data flows in a code base can help. In our experience at Google, however, they do not substantially increase confidence in review-based assessments, since they are necessarily incomplete in their reasoning and subject to both false positives and false negatives. Furthermore, they have similar difficulties as human reviewers with reasoning about whole-system data flows across multiple system components, using a variety of programming languages, RPC (remote procedure call) mechanisms, and so forth, and involving flows traversing multiple time contexts across data stores.

Similar limitations apply to dynamic testing approaches: it is difficult to ascertain whether test suites provide adequate coverage for whole-system data flows.

Templates to the rescue? In practice, HTML markup, and interpolation points therein, are often specified using *HTML templates*. Template systems expose domain-specific languages for rendering HTML markup. An HTML markup template induces a function from template variables into strings of HTML markup.

Figure 1c illustrates the use of an HTML markup template (9): this example renders a user profile in the photo-sharing application, including the user's name, a hyperlink to a personal blog site, as well as free-form text allowing the user to express any special interests.

Some template engines support automatic escaping, where escaping operations are automatically inserted around each interpolation point into the template. Most template engines' auto-escape facilities are noncontextual and indiscriminately apply HTML escaping operations, but do not account for special HTML contexts such as URLs, CSS, and JavaScript.

Contextually auto-escaping template engines⁶ infer the necessary validation and escaping operations required for the context of each template substitution, and therefore account for such special contexts.

Use of contextually auto-escaping template systems dramatically reduces the potential for XSS vulnerabilities: in (9), the substitution of untrustworthy values `profile.name` and `profile.blogUrl` into the resulting markup cannot result in XSS—the template system automatically infers the required HTML-escaping and URL-validation.

XSS bugs can still arise, however, in code that does not make use of templates, as in Figure 1a(1), or that involves non-HTML sinks, as in Figure 1b(5).

Furthermore, developers occasionally need to exempt certain substitutions from automatic escaping: in Figure 1c (9), escaping of `profile.aboutHtml` is explicitly suppressed because that field is assumed to contain a user-supplied message with simple, safe HTML markup (to support use of fonts, colors, and hyperlinks in the “about myself”

user-profile field).

Unfortunately, there is an XSS bug: the markup in `profile.aboutHtml` ultimately originates in a rich-text editor implemented in browser-side code, but there is no server-side enforcement preventing an attacker from injecting malicious markup using a tampered-with client. This bug could arise in practice from a misunderstanding between front-end and back-end developers regarding responsibilities for data validation and sanitization.

Reliably Preventing the Introduction of XSS Bugs

In our experience in Google's security team, code inspection and testing do not ensure, to a reasonably high degree of confidence, the absence of XSS bugs in large Web applications. Of course, both inspection and testing provide tremendous value and will typically find *some* bugs in an application (perhaps even *most* of the bugs), but it is difficult to be sure whether or not they discovered *all* the bugs (or even *almost all* of them).

The primary goal of this approach is to limit code that could potentially give rise to XSS vulnerabilities to a very small fraction of an application's code base.

A key goal of this approach is to drastically reduce the fraction of code that could potentially give rise to XSS bugs. In particular, with this approach, an application is structured such that most of its code cannot be responsible for XSS bugs. The potential for vulnerabilities is therefore confined to infrastructure code such as Web application frameworks and HTML templating engines, as well as small, self-contained application-specific utility modules.

A second, equally important goal is to provide a developer experience that does not add an unacceptable degree of friction as compared with existing developer workflows.

Key components of this approach are:

- *Inherently safe APIs.* Injection-prone Web-platform and HTML-rendering APIs are encapsulated in wrapper APIs designed to be inherently safe against XSS in the sense that no use of such APIs can result in XSS vulnerabilities.

- *Security type contracts.* Special types are defined with contracts stipu-

lating that their values are safe to use in specific contexts without further escaping and validation.

- *Coding guidelines.* Coding guidelines restrict direct use of injection-prone APIs, and ensure security review of certain security-sensitive APIs. Adherence to these guidelines can be enforced through simple static checks.

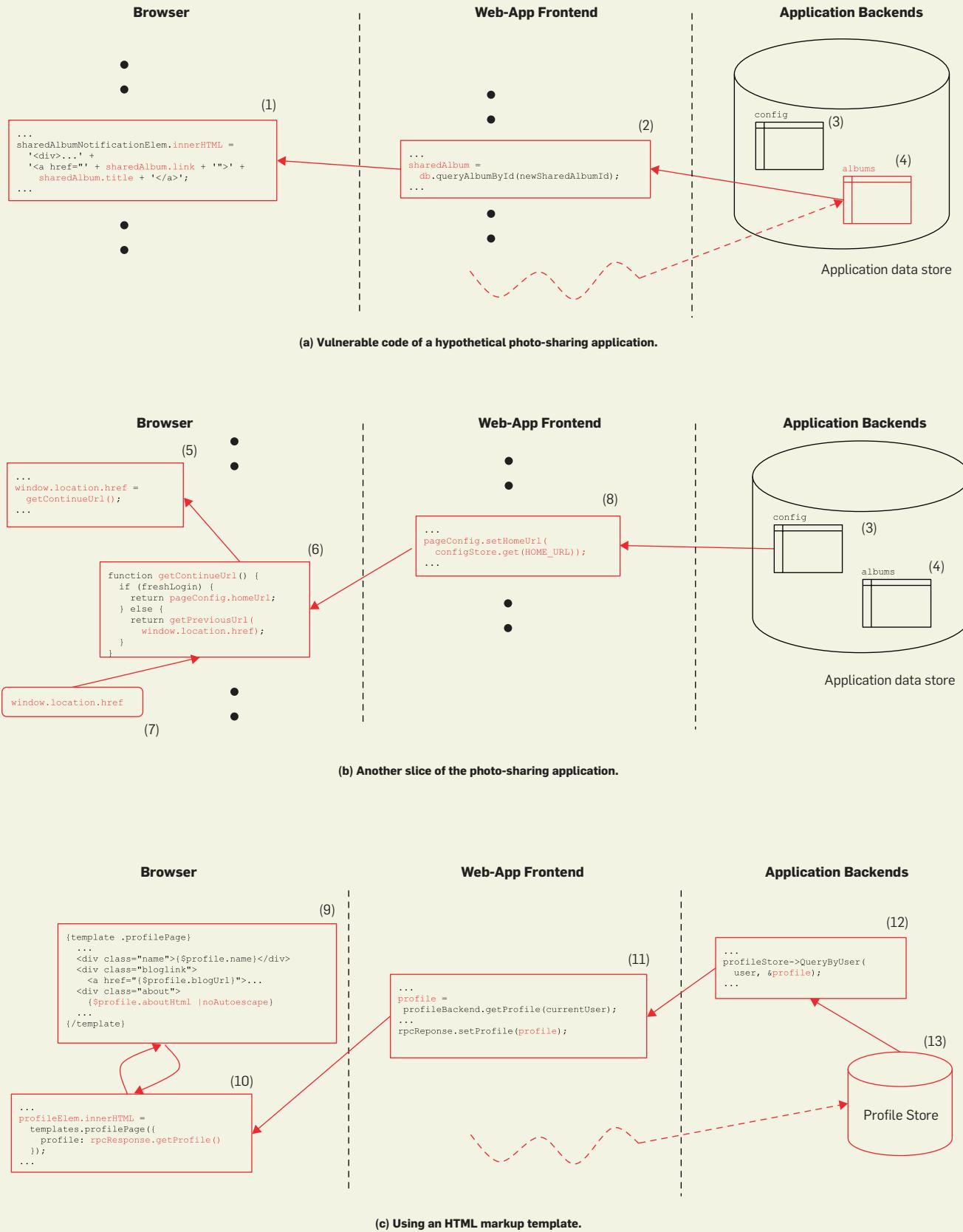
Inherently safe APIs. Our goal is to provide inherently safe wrapper APIs for injection-prone browser-side Web platform API sinks, as well as for server- and client-side HTML markup rendering.

For some APIs, this is straightforward. For example, the vulnerable assignment in Figure 1b (5) can be replaced with the use of an inherently safe wrapper API, provided by the JavaScript Closure Library, as shown in Figure 2b (5'). The wrapper API validates at runtime that the supplied URL represents either a scheme-less URL or one with a known benign scheme.

Using the safe wrapper API ensures this code will not result in an XSS vulnerability, regardless of the provenance of the assigned URL. Crucially, none of the code in (5') nor its fan-in in (6-8) needs to be inspected for XSS bugs. This benefit comes at the very small cost of a runtime validation that is technically unnecessary if (and only if) the first branch is taken—the URL obtained from the configuration store is validated even though it is actually a trustworthy value.

In some special scenarios, the runtime validation imposed by an inherently safe API may be too strict. Such cases are accommodated via variants of inherently safe APIs that accept types with a security contract appropriate for the desired use context. Based on their contract, such values are exempt from runtime validation. This approach is discussed in more detail in the next section.

Strictly contextually auto-escaping template engines. Designing an inherently safe API for HTML rendering is more challenging. The goal is to devise APIs that guarantee that at each substitution point of data into a particular context within trusted HTML markup, data is appropriately validated, sanitized, and/or escaped, unless it can be demonstrated that a specific data item is safe to use in that context based on

Figure 1. XSS vulnerabilities in a hypothetical Web application.

its provenance or prior validation, sanitization, or escaping.

These inherently safe APIs are created by strengthening the concept of contextually auto-escaping template engines⁶ into SCAETEs (*strictly* contextually auto-escaping template engines). Essentially, a SCAETE places two additional constraints on template code:

- ▶ Directives that disable or modify the automatically inferred contextual escaping and validation are not permitted.
- ▶ A template may use only sub-templates that recursively adhere to the same constraint.

Security type contracts. In the form just described, SCAETEs do not account for scenarios where template parameters are intended to be used without validation or escaping, such as `aboutHtml` in Figure 1c—the SCAETE unconditionally validates and escapes all template parameters, and disallows directives to disable the auto-escaping mechanism.

Such use cases are accommodated through types whose contracts stipulate their values are safe to use in corresponding HTML contexts, such as “inner HTML,” hyperlink URLs, executable resource URLs, and so forth. Type contracts are informal: a value satisfies a given type contract if it is known that it has been validated, sanitized, escaped, or constructed in a way that guarantees its use in the type’s target context will not result in attacker-controlled script execution. Whether or not this is indeed the case is established by expert reasoning about code that creates values of such types, based on expert knowledge of the relevant behaviors of the Web platform.⁸ As will be seen, such security-sensitive code is encapsulated in a small number of special-purpose libraries; application code uses those libraries but is itself not relied upon to correctly create instances of such types and hence does not need to be security-reviewed.

The following are examples of types and type contracts in use:

- ▶ `SafeHtml`. A value of type `SafeHtml`, converted to string, will not result in attacker-controlled script execution when used as HTML markup.
- ▶ `SafeUrl`. Values of this type will not result in attacker-controlled script execution when dereferenced as hyperlink URLs.

▶ `TrustedResourceUrl`. Values of this type are safe to use as the URL of an executable or “control” resource, such as the `src` attribute of a `<script>` element, or the source of a CSS style sheet. Essentially, this type promises that its value is the URL of a resource that is itself trustworthy.

These types are implemented as simple wrapper objects containing the underlying string value. Type membership in general cannot be established by runtime predicate, and it is the responsibility of the types’ security-reviewed factory methods and builders to guarantee the type contract of any instance they produce. Type membership can be based on processing (for example, validation or sanitization), construction, and provenance, or a combination thereof.

SCAETEs use security contracts to designate exemption from automatic escaping: a substituted value is not subject to runtime escaping if the value is of a type whose contract supports its safe use in the substitution’s HTML context.

Templates processed by a SCAETE give rise to functions that guarantee to emit HTML markup that will not result in XSS, assuming template parameters adhere to their security contracts, if applicable. Indeed, the result of applying a SCAETE-induced template function itself satisfies the `SafeHtml` type contract.

Figure 2c shows the application of SCAETE and security type contracts to the code slice of Figure 1c. Strict contextual escaping of the template in (9) disallows use of the `noAutoescape` directive. Simply removing it, however, would enable the automatic escaping of this value, which is in this case undesired. Instead, we change the `aboutHtml` field of the profile object to have `SafeHtml` type, which is exempt from automatic escaping. The use of this type is threaded through the system (indicated by the color green), across RPCs all the way to the value’s origin in back-end code (12’).

Unchecked conversions. Of course, eventually we need to create the required value of type `SafeHtml`. In the example, the corresponding field in persistent storage contains HTML markup that may be maliciously supplied by an attacker. Passing this untrusted markup through an HTML

sanitizer to remove any markup that may result in script execution renders it safe to use in HTML context and thus produces a value that satisfies the `SafeHtml` type contract.

To actually create values of these types, *unchecked conversion* factory methods are provided that consume an arbitrary string and return an instance of a given wrapper type (for example, `SafeHtml` or `SafeUrl`) without applying any runtime sanitization or escaping.

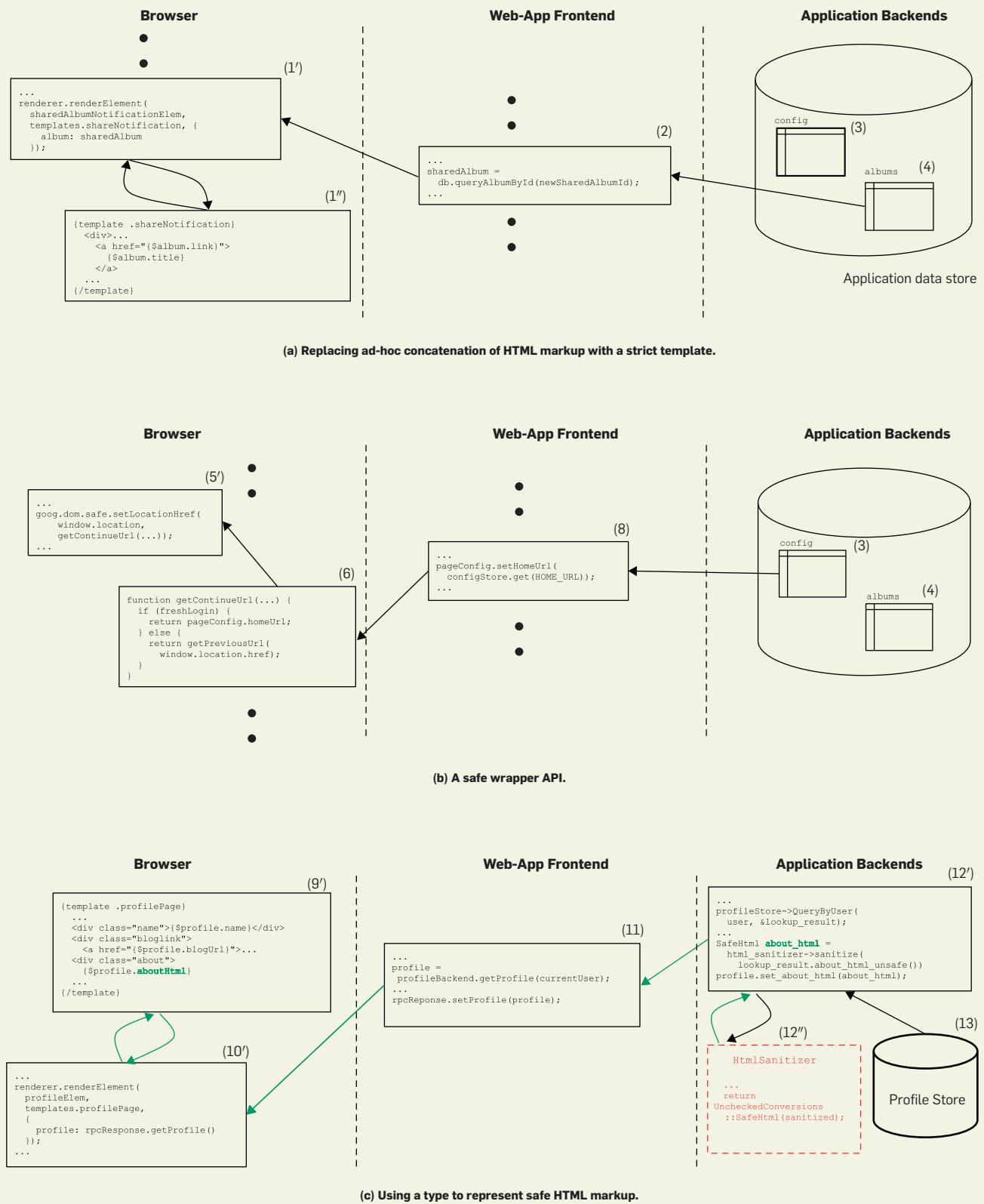
Every use of such unchecked conversions must be carefully security reviewed to ensure that in all possible program states, strings passed to the conversion satisfy the resulting type’s contract, based on context-specific processing or construction. As such, unchecked conversions should be used as rarely as possible, and only in scenarios where their use is readily reasoned about for security-review purposes.

For example, in Figure 2c, the unchecked conversion is encapsulated in a library (12”) along with the HTML sanitizer implementation on whose correctness its use depends, permitting security review and testing in isolation.

Coding guidelines. For this approach to be effective, it must ensure developers never write application code that directly calls potentially injection-prone sinks, and that they instead use the corresponding safe wrapper API. Furthermore, it must ensure uses of unchecked conversions are designed with reviewability in mind, and are in fact security reviewed. Both constraints represent coding guidelines with which all of an application’s code base must comply.

In our experience, automated enforcement of coding guidelines is necessary even in moderate-size projects—otherwise, violations are bound to creep in over time.

At Google we use the open source error-prone static checker¹ (<https://goo.gl/SQXCvw>), which is integrated into Google’s Java tool chain, and a feature of Google’s open source Closure Compiler (<https://goo.gl/UyMVzp>) to whitelist uses of specific methods and properties in JavaScript. Errors arising from use of a “banned” API include references to documentation for the corresponding safe API, advising developers on how to address

Figure 2. Preventing XSS through use of inherently safe APIs.

the error. The review requirement for uses of unchecked conversions is enforced via a package-visibility mechanism provided by Google's distributed build system.³

If tool-chain-integrated checks were not available, coding guidelines could be enforced through simpler lint-like tools.

In the photo-sharing example, such checks would raise errors for the assignments to `innerHTML` and `location.href` in figures 1a–1c and would advise the developer to use a corresponding inherently safe API instead. For assignments to `innerHTML`, this typically means replacing ad hoc concatenation of HTML markup with a strict template, rendered directly into the DOM element by the template system's runtime, as shown in Figure 2a.

Putting It All Together

Revisiting the code slices of the example applications after they have been brought into adherence with the coding guideline shows (figures 2a–2c) that uses of injection-prone data sinks have been replaced with corresponding inherently safe APIs in (1'), (5'), (9') and (10'). Now, none of these code snippets can result in an XSS bug, and neither they nor their fan-in need to be inspected during a security review.

The only piece of code left requiring security code review (aside from infrastructure code such as the implementation of the SCAETE, its runtime, and API wrapper libraries) is the `HtmlSanitizer` package (12"), and specifically the package-local fan-in into the unchecked conversion to `SafeHtml`. Correctness of this conversion relies solely on the correctness of the HTML sanitizer, and this package can be security reviewed and tested in isolation. If a library is shared across multiple applications, its review cost is amortized among users.

Of course, there are limitations to the guarantees this approach can provide: first, the security reviewer may miss bugs in the security-relevant portion of the code (template systems, markup sanitizers, and so forth); second, application code may use constructs such as reflection that violate encapsulation of the types we rely on; finally, some classes of XSS bugs (in practice, relatively rare) cannot be ad-



Using such APIs prevents XSS bugs and largely relieves developers from thinking about and explicitly specifying escaping and data validation.



dressed by generally applied contextual data validation and escaping as ensured by our design patterns, and these need to be addressed at other layers in Web application frameworks or in the design of individual Web applications.⁷

Developer impact. Comparing the vulnerable code slices in figures 1a–1c with their safe counterparts in figures 2a–2c shows our approach does not impose significant changes in developer workflow, nor major changes to code. For example, in Figure 2b (5'), we simply use a safe wrapper instead of the "raw" Web-platform API; otherwise, this code and its fan-in remain unchanged.

The coding guidelines do require developers to use safe APIs to generate HTML markup, such as the strict template in Figure 2a (1'). In return, however, using such APIs prevents XSS bugs and largely relieves developers from thinking about and explicitly specifying escaping and data validation.

Only in Figure 2c is a more significant change to the application required: the type of the `aboutHtml` field changes from `String` to `SafeHtml`, and use of this type is threaded through RPCs from back end to front end. Even here, the required changes are relatively confined: a change in the field's type and the addition of a call to the `HtmlSanitizer` library in back end code (12').

Such scenarios tend to be rare in typical Web applications; in the vast majority of uses the automatic runtime validation and escaping is functionally correct: most values of data flows into user-interface markup, both application-controlled and user-input-derived, tend to represent plain text, regular `http/https` URLs, and other values that validate and/or escape cleanly.

Practical Application

This design pattern has been applied in several open source and proprietary Web application frameworks and template engines in use at Google: support for strict contextual auto-escaping has been added to Closure Templates (<https://goo.gl/Y4G9LK>), AngularJS (<https://goo.gl/RvQvXb>), as well as a Google-proprietary templating system. Security engineers and infrastructure developers at Google have also implemented libraries of types such as `SafeHtml` and `SafeUrl`, and

added inherently safe APIs to the Google Web Toolkit (<https://goo.gl/dGk5G8>), the JavaScript Closure Library (<https://goo.gl/7nbXCg>), and various Google-proprietary libraries and frameworks.

Decrease in incidence of XSS bugs.

It is challenging to derive precise statistics regarding the impact of any particular approach to bug prevention: our design patterns prevent XSS bugs from being introduced in the first place, but we do not know how many bugs would have been introduced without their use.

We can, however, make observations based on bug counts in existing large projects that adopted our approach over time. Such observations can be considered anecdotal only, since bug counts are likely influenced by many variables such as code size and complexity and security-related developer education. Nevertheless, the observations suggest our approach significantly contributes to notable reductions in XSS vulnerabilities.

Several development teams of flagship Google applications have adopted these design patterns and coding guidelines. They have established static enforcement that all HTML markup is produced by strictly contextually auto-escaped templates, and they have disallowed direct use of certain injection-prone Web-platform APIs such as `innerHTML`.

One of the largest and most complex of these applications, using more than 1,000 HTML templates in the Closure Templates language, migrated to strict auto-escaping in early 2013. Throughout 2012 (before migration), 31 XSS bugs were filed in Google's bug tracker against this application. Post-migration, only four XSS bugs were filed in the year to mid-2014, and none at all in the first half of 2014. For another large application (also using more than 1,000 templates) whose migration is still in progress, there was a reduction from 21 to nine XSS bugs during the same time period.

Even without full compliance with the coding guidelines, some benefits can be realized: as the fraction of compliant code increases, the fraction of code that could be responsible for vulnerabilities shrinks, and confidence in the absence of bugs increases. While there is little reason not to write

new code entirely in adherence to the guidelines, we can choose not to refactor certain existing code if the cost of refactoring exceeds benefits and if we already have confidence in that code's security through other means (for example, intensive review and testing).

Conclusion

Software design can be used to isolate the potential for XSS vulnerabilities into a very small portion of an application's code base. This makes it practical to intensively security-review and test just those portions of the code, resulting in a high degree of confidence that a Web application as a whole is not vulnerable to XSS bugs. Our approach is practically applicable to large, complex, real-world Web applications, and it has resulted in significant reduction of XSS bugs in several development projects.

This approach to what is fundamentally a difficult problem involving whole-system data flows incorporates two key principles:

- Based on the observation that in typical Web apps, it is functionally correct to conservatively runtime-escape and -validate the vast majority of data flowing into injection-prone sinks, we choose to treat all string-typed values as potentially untrustworthy and subject to runtime validation and escaping, regardless of their provenance. This design choice altogether obviates the need for whole-program reasoning about the vast majority of whole-system data flows in a typical Web application.

- Only in scenarios where default, runtime validation and escaping is functionally incorrect, we employ type contracts to convey that certain values are already safe to use in a given context. This use of types permits compositional reasoning about whole-system data flows and allows security experts to review security-critical code in isolation, based on package-local reasoning.

Our coding guidelines impose certain constraints on application code (though they typically require only limited changes to existing code). In contrast, many existing approaches to the prevention and detection of XSS aim to be applicable to existing, unmodified code. This requirement makes

the problem much more difficult, and generally requires the use of complex whole-program static and/or dynamic data-flow analysis techniques. For an overview of existing work in this area, see Mike Samuel et al.⁶ Relaxing this requirement negates the need for special-purpose tools and technologies (such as runtime taint tracking or whole-program static analysis), allowing us to rely solely on the combination of software design, coding guidelines enforceable by very simple static checks, existing language-native type systems, and a small enhancement to existing contextually auto-escaping template systems. Thus, our approach can be used in applications written in a variety of programming languages, without placing special requirements on tool chains, build systems, or runtime environments. □

Related articles on queue.acm.org

Fault Injection in Production

John Allspaw

<http://queue.acm.org/detail.cfm?id=2353017>

High Performance Web Sites

Steve Souders

<http://queue.acm.org/detail.cfm?id=1466450>

Vicious XSS

George Neville-Neil

<http://queue.acm.org/detail.cfm?id=1113330>

References

1. Aftandilian, E., Sauciuc, R., Priya, S. and Krishnan, S. Building useful program analysis tools using an extensible Java compiler. *International Working Conference on Source Code Analysis and Manipulation* (2012), 14–23.
2. Daswani, N., Kern, C. and Kesavan, A. *Foundations of Security: What Every Programmer Needs to Know*. Apress, 2007.
3. Morgenhaler, J.D., Gridnev, M., Sauciuc, R. and Bhansali, S. Searching for build debt: Experiences managing technical debt at Google. *Third International Workshop on Managing Technical Debt* (2012), 1–6.
4. OWASP. Top 10 List, 2013; https://www.owasp.org/index.php/Top_10_2013-Top_10.
5. OWASP. XSS (cross site scripting) prevention cheat sheet, 2014; [https://www.owasp.org/index.php/XSS_\(Cross_Site_Scripting\)_Prevention_Cheat_Sheet](https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet).
6. Samuel, M., Saxena, P. and Song, D. Context-sensitive auto-sanitization in Web templating languages using type qualifiers. *Proceedings of the 18th ACM Conference on Computer and Communications Security* (2011), 587–600.
7. Su, Z. and Wasserman, G. The essence of command injection attacks in Web applications. In *Proceedings of POPL* (2006); <http://dl.acm.org/citation.cfm?id=1111070>
8. Zalewski, M. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2012.

Christoph Kern (xtof@google.com) is an information security engineer at Google. His primary focus is on designing APIs and frameworks that make it easy for developers to write secure software and eliminate or reduce the risk of accidentally introducing security bugs.

Copyright held by author.