SCRIPTGARD: Preventing Script Injection Attacks in Legacy Web Applications with Automatic Sanitization

Prateek Saxena
University of California, Berkeley

David Molnar and Benjamin Livshits
Microsoft Research

Abstract

The primary defense against cross site scripting attacks in web applications is the use of *sanitization*, the practice of filtering untrusted inputs. We analyze sanitizer use in a shipping web application with over 400,00 lines of code, one of the largest applications studied to date. Our analysis reveals two novel problems: *inconsistent sanitization* and *inconsistent multiple sanitization*. We formally define these problems and propose SCRIPTGARD: a system for preventing such problems automatically matching the correct sanitizer with the correct browser context. While command injection techniques are the subject of intense prior research, none of the previous approaches consider both server and browser context, none of them achieve the same degree of precision, and many other mitigation techniques require major changes to server side code. Our approach, in contrast, can be incrementally retrofitted to legacy systems. We show how to provide an aid to testers during development. Finally we sketch how SCRIPTGARD can be used as a runtime mitigation technique.

1. Introduction

Applications that render HTML on a web server for consumption by a web browser are explosively popular, but they introduce new classes of security bugs. The most common attacks are cross-site scripting (XSS) [3, 33] and cross-channel scripting (XCS) [5]. At the core of these attacks is injection of JavaScript code into a context not originally intended. These attacks lead to stolen credentials and actions performed on the user's behalf by an adversary.

To make matters worse, web applications today include not only Internet services, but also boxed products that are deployed to homes and businesses. As a result, patches to web applications can incur serious costs to fix. Just as in the case of buffer overflows with traditional desktop software, so too must a patch for these systems be pushed out to all installs all over the world.

While substantial research and commerical interst focuses on web vulnerability scanning, our work in contrast aims at retrofitting legacy web applications to make them correct before deployment. Ideally, we could create systems that resist attacks by construction. In this direction, recent work such as BLUEPRINT has proposed primitives to encode HTML output in a safe way [23]. Unfortunately, these techniques are difficult to apply to legacy web applications because they make fundamental changes to the way an application creates HTML for consumption by the web browser.

We therefore need mitigations for XSS attacks on web applications that can be incrementally retrofitted to existing code. We start by analyzing defenses in an existing Web application, a large (over 400,00 lines of code) production-grade web-based product representative of this class of applications. The application is both widely used and also must handle potentially untrusted data from multiple sources.

We focus on the use of *sanitization*: the practice of applying filters to untrusted data to preclude attacks. We discover a class of errors that are not an artifact of poorly designed sanitization routines or complete lack of validation, but rather due to subtle nesting of HTML contexts and sharing of dataflow paths in the application. Based on the empirical analysis, we propose the core problem of *automatic sanitizer placement*: given an application, automatic selection of the appropriate sanity check to apply on a dataflow path from a potentially dangerous data source to an HTML output sink. Sanitization is also needed for cases where data must be encoded to avoid causing functionality errors in a specific context, such as encoding URLs in JavaScript.

Our results show that human developers accidentally mismatch the choice of sanitizer with the browser's parsing context. This is unsurprising because correct placement of sanitizer requires a developer to have global knowledge of the program as a whole. We quantify two specific classes of errors: inconsistent sanitization and *inconsistent multiple sanitization*. Program models used in prior work are not as precise as ours, in particular not modeling the browser parsing context, rendering them unable to detect this class of errors.

The problem of automatic sanitizer placement is different from the problem of ensuring sanitizer correctness. Our results and techniques are complementary to those used for checking correctness of sanitization routines [26, 1, 15], and new proposals for encoding untrusted output in a browser-agnostic way [23]. There is a separation of concerns: given a set of sanitizers that is "correct" for specific application contexts, our work shows how to properly place those sanitizers in an application.

Solving the automatic sanitization placement problem for legacy applications is a step toward provably eliminating code injection attacks such as cross site scripting. The end goal of this line of research is a system that provides provable guarantees against code injection attacks with minimal retrofitting to legacy applications.

We propose an automatic sanitization technique that prevents errors arising from mismatches of sanitizers and browser parsing contexts. We implement the technique in SCRIPTGARD and apply it to a large application with over 400,00 lines of code. Unlike existing template-based HTML writing systems, such as ASP .NET's web and HTML controls, SCRIPTGARD performs *context-*

sensitive sanitization. SCRIPTGARD allows developers to create custom nesting of HTML contexts, which web applications often do, without sacrificing the consistency of the sanitization process. SCRIPTGARD employs several simple, but novel features that make it more amenable to practical deployment and wider defense: (a) we sketch, but have not yet implemented, a design that shifts the performance intensive analysis to a pre-deployment analysis server, leaving only a low-overhead runtime path detector at runtime; (b) during analysis, SCRIPTGARD employs *positive taint-tracking*, which contrasts with traditional taint-tracking because it is conservative (hence does not miss identifying sources of untrusted data) and can provide defense against cross channel-scripting attacks [5].

1.1 Contributions

This paper makes the following contributions:

- **Motivation and analysis.** In Section 2, we demonstrate that manual sanitizer placement is error-prone. We identify two classes of errors that are common: *inconsistent sanitization* and *inconsistent multiple sanitization*. In other words, sanitizer placement varies depending on the execution *path*, not just the sink to which the data flows. In contrast state-of-the-art templating systems such as ASP.NET consider only the sink, leading to inconsistent practices. We also discover that certain combinations of sanitizers used in practice can result in unintended errors.
- **Testing for security.** We propose a testing strategy for finding sanitization flows that involves automatic server-side instrumentation combined with high-fidelity web browser instrumentation. Our ability to do both server- and client-side provenance tracking allows us to reason about potential sanitization flaws with considerably better precision compared to techniques that focus exclusively on the server [1, 20, 24].
- **Runtime auto-sanitization.** We propose, but have not yet evaluated, a low-overhead runtime monitoring and repair solution that automatically performs the correct sanitization strategy. Our approach either adds sanitizers or removes existing ones. We sketch how a training phase can reduce the runtime overhead by caching path specific information used to pick the correct sanitizers.
- **Evaluation.** In Section 5 we evaluate both the testing approach as well as runtime auto-sanitization on a range of large-scale widely-used web applications. Our benchmark is an application with over 400,00 lines of code. We performed our security testing on a set of 53 large web pages derived from 7 applications built on top of our test platform. Each page contains 350–900 DOM nodes. We found inconsistent sanitization on 1,207 paths SCRIPTGARD analyzed (4.7% of the total paths) and observed an additional 285 instances of inconsistent multiple sanitization. Our runtime mitigation technique ensures that none of these inconsistent uses of sanitizers will result in problems at runtime. We provide provable guarantees subject to caveats described in Section 6 on the correctness of sanitizers and the soundness of our runtime instrumentation.

1.2 Paper Organization

The rest of the paper is organized as follows. Section 2 gives motivation for the specification inference techniques SCRIPTGARD uses. Section 3 provides a formalization for our work. Section 4 talks about SCRIPTGARD analysis and runtime recovery. Section 5 describes our experimental evaluation. Section 6 highlights topics for discussion. Finally, Sections 7 and 8 describe related work and conclude.

2. Overview

This section is organized as follows. Section 2.1 introduces a motivating example capturing features of real-world code we analyzed. Section 2.2 emphasizes some of our observations about the structure of real-world, legacy applications embodied in the running example. Section 2.3 summarizes key challenges that exist in building a solution. Section 2.4 presents the common sanitization errors addressed by SCRIPTGARD. Finally, we provide an overview of SCRIPTGARD's solution.

2.1 Motivating Example

Figure 2 shows a fragment of ASP/C# code which is difficult to sanitize correctly. Before we delve into why this is so, we first explain how this code is representative of characteristics seen in real-world code.

Example 1 ASP .NET platform provides boiler-plate templates for HTML output, called "controls", which are classes that handle safe rendering of of HTML content. The platform provides a set of built-in sanitizers that it either automatically applies on output variables or allows web applications to override using custom sanitizers. However, we have observed that several large applications find the expressiveness provided by the built-in controls insufficient and extensively define custom controls. The running example is typical of such real-world applications— it defines its own custom controls, `AnchorLink` and `DynamicLink`, to render user-specified links. The first custom control enables rendering links directly as HTML whereas the second control allows dynamically inserting it in the web page via JavaScript. The code invokes the `Write` method of the built-in `HtmlWriter` class to incrementally write string outputs to the HTML stream.

In this code, there are four data flow paths from the source of input (the function `AnchorLink.SetAttribRender`), which accepts a user link to emit, to the same HTML output operations defined in function `BaseTagControl.RenderControl`. The four paths result from the two branch conditions in the shown fragment. Each of the four paths is a series of operations that constructs a distinct HTML "template" by concatenating strings with data that must be sanitized. In essence, execution of each path instantiates a different HTML template with data to be sanitized. Figure 1 shows the four different templates instantiated with data to be sanitized by the execution of the four different paths in the running examples. ■

Each application template places data in certain HTML contexts. An example HTML context is the "URL Path Attribute" context, which means that the web browser is expecting a string that the browser will treat as a URL path. Each context requires a different *sanitizer*, which is a function from strings to strings that transforms the data into a string safe for embedding in the associated context. For instance, the `EcmaScriptStringEncode` sanitizer is function that makes a string safe for usage in the JavaScript string literal context. Here safe may refer both to functionality and to security; sanitization prevents errors from strings that are not valid for the parsing context. The implementation of the sanitizer, for the context of our abstraction, is not relevant— indeed, many different implementations can achieve the safety property.

This code is representative of applications that are subject to static analysis and security audits — it is careful to restrict the context in which data is allowed to be embedded. For instance, it rigorously appends quotes to each attribute value, as recommended by security guidelines [28]. We have eliminated the actual sanitization checks in this example to illustrate the challenges in applying them correctly. If sanitization checks were missing in the code to begin with, several static/dynamic analysis tools (such as CAT .NET [7]) can already identify these deficiencies. As a result, we

HTML output	Nesting of HTML contexts
`<script type="text/javascript"> document.write(' <a href="javascript: onNav(\u0027 TOENCODE\u0027);"></a>'); </script>`	JavaScript String Literal, Html Attribute, JavaScript String Literal
`<a href="javascript: onNav('TOENCODE');"></a>`	Html Attribute, JavaScript String Literal
`<a href="TOENCODE"></a>`	Html Attribute
`<script type="text/javascript"> document.write( ' <a href="TOENCODE"></a>'); </script>`	JavaScript String Literal, Html Attribute

Figure 1. The different HTML templates output by executing different paths in the running example. TOENCODE denotes the holes in the output templates, which are filled by data that must be sanitized.

ignore the class of errors where sanitization is completely missing in this work.

2.2 Observations

This example illustrates some features of real-world code that motivate the need for our analysis and automatic repair.

Nested contexts in output templates. In any output template, data may be simultaneously placed in multiple contexts. For instance, the first output template demonstrated in Figure 1 illustrates the issue. The string to be encoded is placed in three nested contexts: a JavaScript string, HTML attribute, and another JavaScript string in that order. To understand this nesting of contexts, consider the behavior of the web browser when parsing this shown fragment of HTML code. The browser first recognizes a `script` code block and feeds th string to be encoded to the JavaScript parser which creates an AST corresponding to `document.write` function. Therefore, the string is first processed by the JavaScript string lexer/parser and hence we say, it is placed in a JavaScript string context. When the `document.write` statement executes, a DOM node for an anchor tag is created dynamically. This invokes the HTML attribute parser in turn of the `"javascript : ..."` string. This step places the data in the HTML attribute context. Finally, the JavaScript parser is invoked again when the JavaScript URI is processed by the browser.

At each stage of the parsing, the sanitization must prevent the string from breaking out of the current context or introducing a dangerous sub-context. Each context may allow a different set of attack vectors — for instance, a " does not break out of a single-quoted JavaScript string, but does prematurely end the double-quoted attribute value. Furthermore, the problem becomes more challenging because of the fact that the browser automatically transforms the string when transitioning from one context to the other — for instance, the string is automatically Unicode decoded when the AST is generated from the JavaScript string context. We conclude that the applied sanitization must deal with multiple nested contexts, as well as, with the transcoding effect introduced by the browser on context transitions. Failure to sanitize the input in an order consistent with the order of nesting of contexts can result in broken functionality or unexpected behavior.

String templates. String operations (such as concatenation and format string text substitution) are used to construct HTML output templates. However, string operations are also used to implement a variety of other semantic operations that are not related to output template construction. In legacy applications, templates are constructed piecewise, often spanning operations in more than one function. Identifying operations that implement the semantic task of creating output templates requires program analysis. We observe that even for a string operation that is ascertained to be a template constructing operation, automatically identifying the context it introduces in the template is challenging. For instance, prepending the string `"<a href="` to data conceptually places it in an HTML attribute context during template construction. However, this semantic interpretation is not explicit in the code representation and is difficult to derive without analysis.

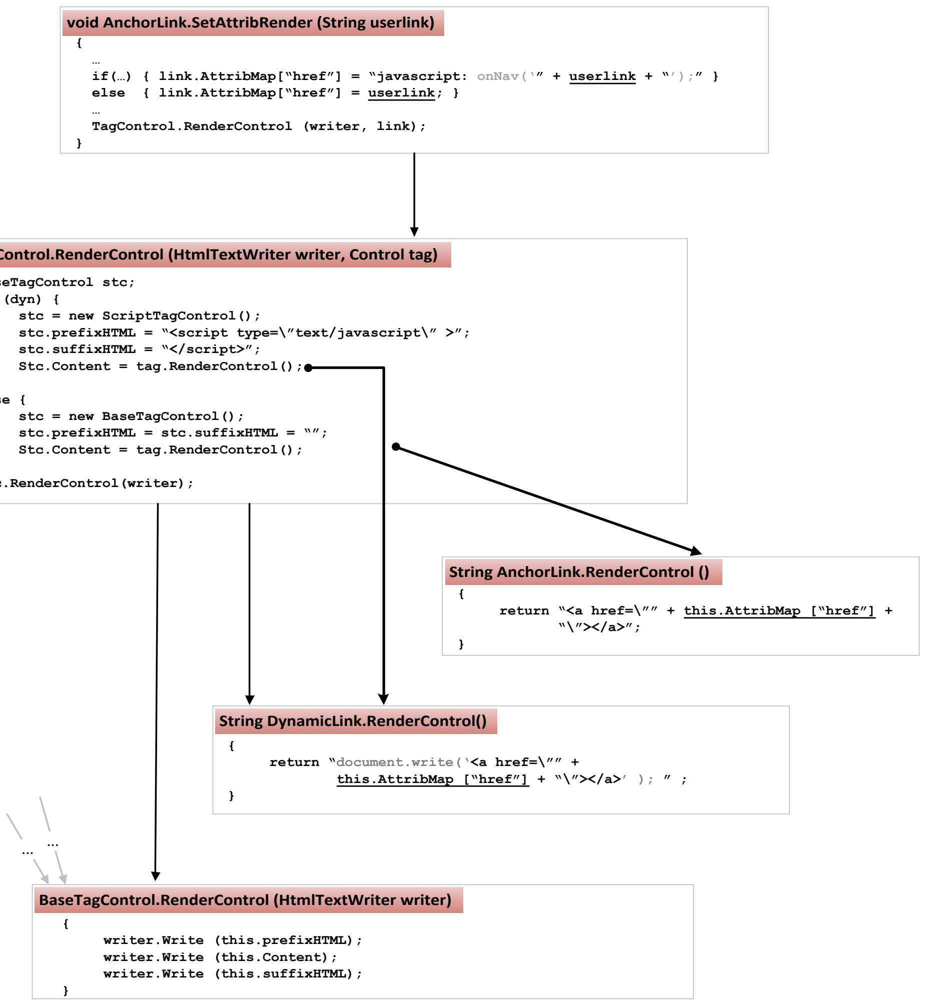
Intersecting data-flow paths. Note that the four paths share code blocks, including the HTML output operations — there may be other code fragments that share the HTML output function `BaseTagControl.RenderControl` which are not shown in the figure. In our experience, such sharing is common in real-world applications. Since different dataflow paths introduce different HTML templates a sequence of sanitizers applied for one dataflow path may be incorrect for the other paths.

2.3 Challenge: Sanitizer Placement

The problem of sanitizer placement is to select one or more sanitizers, from a set of given sanitizers, and apply them on the input data such that the data is safe for embedding in all the possible HTML output templates of the application. We assume that the sanitizers and their mapping to their associated HTML contexts is specified (an example from our test application is shown in Figure 8 later in the paper) and correct. We argue that the prevalent practice of manually deciding the placement of sanitizers (typically, at a subset of code locations) is hard to get right. Next, we discuss how two commonly employed sanitization strategies do not safely sanitize this example.

Relying on output sanitization. Consider the case when the developer applies a sanitizer for the data at the HTML output sink, namely at line 2 in function `BaseTagControl.RenderControl`. Later in the paper we derive the input specification for our test application, which we show in Figure 8. According to the specification, `HtmlAttribEncode` sanitizer matches the HTML context. Notice, however, that there is more than one embedding context possible at this output point due to dataflow path sharing. As a result, the picked sanitizer will be (a) consistent for templates resulting for the path that executes function `AnchorLink.RenderControl`, but (b) inconsistent for templates which result in `<SCRIPT>` block templates from paths that execute the function `DynamicLink.RenderControl`.

The reader may verify that moving the sanitizer application to one earlier (template-constructing) operation in the dataflow path is not a panacea either — if we pick an appropriate sanitizer to apply in either `AnchorLink.RenderControl` or `DynamicLink.RenderControl`, it may still be inconsistent. This is because the branch taken in the function `AnchorLink.SetAttribRender` decides whether



```
void AnchorLink.SetAttribRender (String userlink)
{
    …
    if(…) { link.AttribMap["href"] = "javascript: onNav('" + userlink + "');" }
    else  { link.AttribMap["href"] = userlink; }
    …
    TagControl.RenderControl (writer, link);
}
```

```
TagControl.RenderControl (HtmlTextWriter writer, Control tag)
BaseTagControl stc;
if (dyn) {
      stc = new ScriptTagControl();
      stc.prefixHTML = "<script type=\"text/javascript\" >";
      stc.suffixHTML = "</script>";
      Stc.Content = tag.RenderControl();
}
else {
      stc = new BaseTagControl();
      stc.prefixHTML = stc.suffixHTML = "";
      Stc.Content = tag.RenderControl();
}
Stc.RenderControl(writer);
```

```
String AnchorLink.RenderControl ()
{
      return "<a href=\"" + this.AttribMap ["href"] +
             "\"></a>";
}
```

```
String DynamicLink.RenderControl()
{
      return "document.write('<a href=\"" +
                this.AttribMap ["href"] + "\"></a>' ); " ;
}
```

```
BaseTagControl.RenderControl (HtmlTextWriter writer)
{
      writer.Write (this.prefixHTML);
      writer.Write (this.Content);
      writer.Write (this.suffixHTML);
}
```

Figure 2. Running example: Code fragment showing pseudo-code for dynamic web application output generation, illustrating the problem of automatic sanitizer placement. Underlined values must be sanitized.

`EcmaScriptStringEncode` (for `javascript: OnNav('` context) or `HtmlAttribEncode` (for attribute context) is the right sanitizer to apply. If the incorrect one is picked, it could break the intended functionality of the application. For instance, the `HtmlAttribEncode` function should eliminate `javascript:` from the input string to be correct. Unfortunately, if we applied `HtmlAttribEncode` to the path that legitimately places the data in a `javascript:` context, this would break the application's intended behavior. The choice of sanitizer is inherently dependent on the runtime behavior of the program.

Relying on eager sanitization. Developers sometimes rely on sanitizing data eagerly at the input interface. Consider the case when we pick the point for applying the sanitizer to be in the function `AnchorLink.SetAttribRender`. We claim that selecting a consistent sanitizer at this point is difficult because the input data is embedded in different nested contexts based on the final output template. Specifically, functions `AnchorLink.Ren derControl` and `DynamicLink.RenderControl` introduce additional but different HTML contexts in the output template; however, this is hard to discover from analyzing the code in `AnchorLink.SetAttribRender` function, which is two hops earlier in the static call chain and may even be in different compilation units.

2.4 Common Sanitization Errors

In practice, we observe two problems that result from erroneous sanitizer placement: *inconsistent sanitization* and *inconsistent multiple sanitization*. We explain them briefly in this section, and refer the reader to Section 5 for details on how often we empirically observed these errors in real-world code.

- *Inconsistent sanitization* refers to cases where a single sanitizer is applied in a manner inconsistent with the (possibly nested) HTML context where the data is emitted.
- *Inconsistent Multiple sanitization* errors refer to cases where a *sequence* of sanitizers is applied on a path, but the resulting transformation is inappropriate for the browser context. Inconsistent multiple sanitization arises for two reasons (a) for nested contexts, a sequence of one of more sanitizers are to be applied, and (b) most common sanitizers (such as the built-ins in .NET and PHP) are not idempotent or commutative. Therefore, one

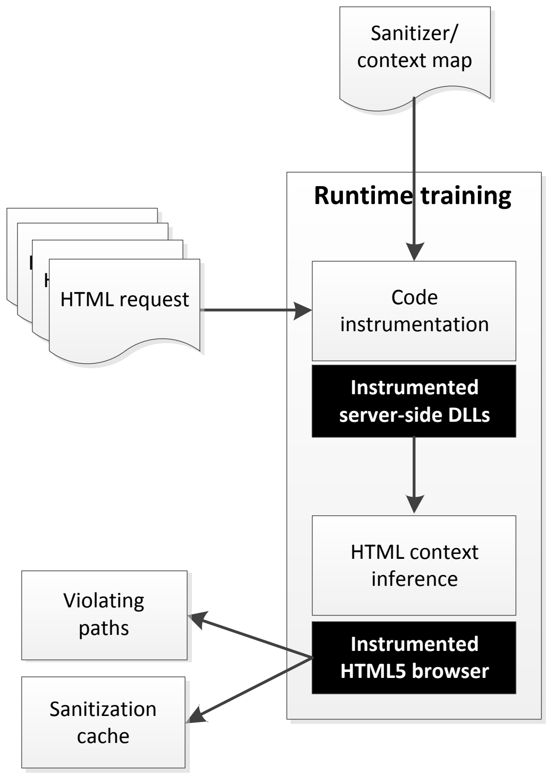


Figure 3. SCRIPTGARD architecture

```
document.write('<a href=" +
HtmlAttribEncode(EcmaScriptStringEncode(this.AttribMap["href"]))+...
```

(a) Method 1

```
document.write('<a href=" +
EcmaScriptStringEncode(HtmlAttribEncode(this.AttribMap["href"]))
+ ...
```

(b) Method 2

Figure 4. Different sanitization approaches.

ordering of sanitizers may be proper for a browser context, but the other is not.

Example 2 (Inconsistent Multiple Sanitization) Consider the sanitizers to be applied for the function `DynamicLink.RenderControl` in the running example. The function places a string inside a double-quoted `href` attribute which in turn is placed inside a JavaScript string. According to the specification in Figure 8, two sanitizer functions, `EcmaScriptStringEncode` and `HtmlAttribEncode`, are to be applied for the JavaScript string context and the HTML attribute context, respectively. Let us assume a commonly recommended implementation for the sanitizers [28]; for instance, `EcmaScriptStringlEncode` transcodes all characters (including the `"` character) to Unicode encoding (`\u0022` for `"`), and, `HtmlAttribEncode` entity encodes characters (`&quot;` for `"`). There are two ways to compose the two sanitizers, which are shown in Figure 4.

From the description of the sanitizers, we notice that the sanitizers do not commute, i.e., the order of composition matters. It turns out that for this output template, the first sequence composition of sanitizers is *inconsistent* while the other is safe.

The key observation is that applying `EcmaScriptStringEncode` first encodes the `"` character as a Unicode representation `\u0022`. The Unicode representation is not transformed when sanitized with `HtmlEncode`.

When the web browser parses the HTML output template, the data is first placed in the JavaScript string literal and then subsequently enters the attribute context after the `document.write` executes. As a result of the inconsistent sequence of sanitizers, the string `" onclick=...` would be transformed to `\u0022 onclick=...`. This would be converted to its original form after the Unicode decoding occurs in the web browser after being processed as the JavaScript string literal. The `"` then will cause the web browser to transition out of the HTML `href` context. This unexpected transition could negatively impact the functionality of the web site, potentially even leading to a security vulnerability.

The opposite order of sanitizer composition would completely eliminate the possibility of such an unexpected browser transition. In this case, the `"` character would be caught by `HtmlEncode` before the string is passed to `EcmaScriptStringEncode`. ■

2.5 SCRIPTGARD Solution Overview

Figure 3 illustrates the SCRIPTGARD solution. We start with a provided sanitizer/context map. At the core of SCRIPTGARD, there is runtime training. We take a deployed version of the web application and run a series of requests against it. The application is instrumented to precisely keep track of data provenance. To mark the application of sanitizers, which in turn put the browser parser in different parsing contexts, we augment the output page with special SCRIPTGARD markup. Section 4.1.2 describes this process in more detail.

In other words, each HTTP request produces an HTTP response, which is the page that that application sends to the browser. Our approach is to use a specially modified browser parser that has been augmented to understand — and remove the special SCRIPTGARD markup, converting it to context information. This form of analysis results in us

- recoding inconsistent paths to be reported to the application developer or security analyst;
- for consistently sanitized paths, we record the sanitization strategy to be later used for runtime enforcement.

In the next two sections, we proceed to first formalize the problem of correct sanitizer placement and then describe the SCRIPTGARD implementation in much more detail.

3. Formalizing Sanitizer Placement

The goal of this work is to ensure that SCRIPTGARD is effective at preventing inconsistent sanitization. To do this, we are going to reason formally about what constitutes inconsistent sanitization and our strategy for detecting such inconsistencies. Our goal in this section is a formalization that is precise enough to prove theorems, high-level enough to abstract implementation details, but still close enough to the latter to remain meaningful.

We start with an abstract model of the browser. This allows us to define precisely what we mean by a *parsing context*. The notion of a context is closely tied to sanitizers that are used, as discussed previously in Section 2. For example, `HtmlAttributeEncode` will properly escape strings in the HTML attribute context, but it is inconsistent to use in other contexts. That in turn allows us to precisely characterize context-sanitizer mismatches. We then define

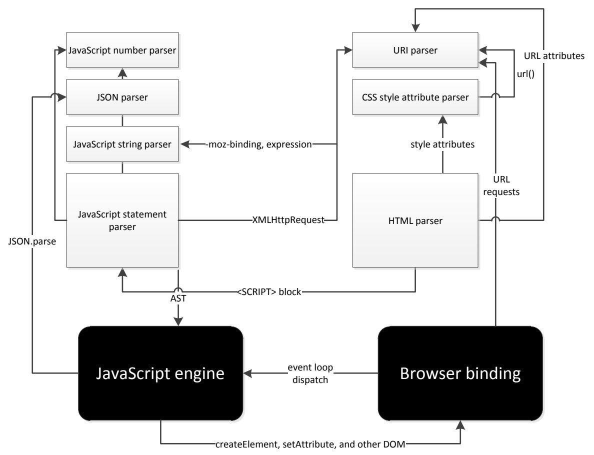


Figure 5. An abstract model of an HTML 5-compliant Web browser. Gray boxes represent various parsers for the browser sub-grammars. Black boxes are the major browser execution components.

what it means for a server-side program to prevent all such context-sanitizer mismatches. Finally, we show that our strategy in SCRIPT-GARD in fact transforms programs into ones that ensure (statically or dynamically) that no context-sanitizer mismatches are possible.

3.1 Browser Model: Definitions

We begin with a browser model as illustrated in Figure 5. For our purposes, we model a web browser as a parser consisting of sub-parsers for several languages. Of course, a real browser has a great deal of implementation issues and side effects to grapple with, but these are out scope of the problems we consider here.

More precisely, we treat the browser as a *collection* of parsers for different HTML standard-supported languages. Figure 5 shows the sub-grammars corresponding to the HTML language, JavaScript language, and the languages for web addresses and inline style attributes.

Because inconsistent application behavior may depend on context that are more fine-grained that regular HTML or JavaScript parsing, we can further divide each sub-grammar into partitions. For instance, the figure shows the JavaScript grammar further subdivided into the string literal sub-grammar, JSON sub-grammar, statement sub-grammar and the number sub-grammar. Formally, we model the browser as a composition of multiple sub-grammars.

DEFINITION 1. *Let $G_1, G_2, ...G_n$ be n sub-grammars, where each context-free grammar $G_i = (V_i, \Sigma_i, S_i, P_i)$ is a quadruple consisting of a set of non-terminals V_i, terminals Σ_i , start symbol S_i and productions P_i.*

Let $\mathcal{T}$ be a set of grammar cross-grammar transition symbols *and the grammar transition productions $P_{\mathcal{T}}$, be a set of productions of the form $A \rightarrow T_i$ or $T_i \rightarrow B$, such that $A \in V_i$, $B \in V_j$ $(i \neq j)$ and $T_i \in \mathcal{T}$.*

We define a web browser as a grammar $\mathcal{G} = \{V, \Sigma, \mathcal{S}, P\}$, with non-terminals $V = V_1 \cup V_2 ... \cup V_n$, terminals $\Sigma = \cup \Sigma_i$, start symbol $\mathcal{S}$ and a set of productions $P = P_{\mathcal{T}} \cup P_1 \cup P_2 ... P_n$.

Conceptually, parsers for various languages are invoked in stages. After each sub-parser invocation, if a portion of the input HTML document is recognized to belong to another sub-language, that portion of the input is sent to the appropriate sub-language parser in the next stage. As a result, any portion of the input HTML document may be recognized by one or more sub-grammars. Transitions from one sub-grammar to another are restricted through productions involving special transition symbols defined above as $\mathcal{T}$, which is key for our formalization of context. In a real web browser, each transition from one sub-grammar to another may be accompanied by a one or more transduction steps of the recognized input.

Example 3 For instance, data recognized as a JavaScript string is subject to Unicode decoding before being passed to the AST. Or, when certain part of the HTML document is recognized as a URI, HTML 5-compliant browsers subject the data to percent-encoding of certain characters before it is sent to the URI parser [10].

This form of encoding can be modeled using additional rules in either of the sub-grammars. While restricting the browser formalism to a context-free grammar might elide some of the real-world complexities, we find this to be a convenient way for defining the notion of context, which appears to match the reality quite well. ■

3.1.1 Parsing Contexts

We formally define the notion of a browser *parsing context* here, with reference to the grammar $\mathcal{G}$. Intuitively, a context reflects the state of the browser at a given point reading a particular piece of input HTML. Each step in the derivation, denoted by $\Rightarrow$ applies a production rule and yields a "sentential form", i.e., a sequence consisting of non-terminals and terminals. We model the parsing context as a sequence of transitions made by the parser between the sub-grammars in $\mathcal{G}$, only allowing the derivations that denote transition from one sub-grammar to another.

DEFINITION 2. *Let derivation $D : \mathcal{S} \Rightarrow^* \gamma$ correspond to the sequence $(P_1, P_2, ...P_k)$ of production rule applications to derive a sentential form γ from the start symbol $\mathcal{S}$.*

A browser context $\mathcal{C}_{\mathcal{D}}$ induced by a derivation D is defined as a projection $(P_1, P_2, \dots P_k) \rightarrow_{\downarrow} (P'_1, P'_2 \dots P'_l)$, that preserves only the productions P'_i in the set of grammar transitions $P_{\mathcal{T}}$.

Our grammars are deterministic, so the notion of inducing a parsing context is well-defined. The browser enters a particular context as a result of processing a portion of the input.

DEFINITION 3. *We say that an input I induces browser context $\mathcal{C}$, if*

- *$D : S(I) \Rightarrow^* \gamma$ (on input I, $\mathcal{S}$ reduces to γ following derivation $\mathcal{D}$), and*
- *$\mathcal{D}$ induces context C.*

3.1.2 Sanitizers

A complex modern web application typically has a variety of both server- and client-side *sanitizers*. We make the simplifying assumption that sanitizers are *pure*, i.e. lacking side-effects. Our review of dozens of real-life sanitizers confirms this assumption. We model sanitizers as abstract functions on strings. Formally,

DEFINITION 4. *A* sanitizer *is a function f : string $\mapsto$ string.*

DEFINITION 5. *A context-sanitizer map is*

$$\psi(\mathcal{C}) = \vec{f}$$

where $\mathcal{C}$ is a context and $\vec{f}$ is a sequence of sanitizers.

The goal of sanitization is typically to remove special characters that would lead to a sub-grammar transition. For example, we often do not want a transition from the HTML parsing context to the JavaScript parsing context, which would be enabled by inserting a `<SCRIPT>` block in the middle of otherwise non-offending HTML. Of course, this but one of many ways that the parser can transition to a different context. Next, we define the *correctness* of a sequence of sanitizers. The intuition is that is after sanitization, the state of parsing is confined to a single context.

DEFINITION 6. *Let input $\mathcal{I}$ consist of the* parsed *and* non-parsed *portion: $\mathcal{I} = [\mathcal{I}_P \circ \mathcal{I}_{NP}]$. Let input $\mathcal{I}_P$ induce browser context $\mathcal{C}$ such that $\psi(\mathcal{C}) = \vec{f}$.*

Then we say that the context-sanitizer map is correct *if when $\vec{f}(\mathcal{I}_{NP})$ is reduced, the grammar never leaves context $\mathcal{C}$.*

In other words, applying the correct sequence of sanitizers "locks" a string in the current context. In particular, a string locked in the HTML context cannot cause the browser to transition to the JavaScript context, leading to a code injection.

3.2 Server-side Program: Definitions

So far, our discussion has focused on parsing HTML strings within the browser regardless of their source. Our goal is to produce HTML on the server that will always have consistent sanitization. Server-side programs take both untrusted and trusted inputs. Untrusted inputs are the well-known sources of injection possibilities such as HTML form fields, HTML headers, query strings, cookies, etc. Trusted inputs are often read from configuration files, trusted databases, etc. Note that the notion of what is trusted and what is not is often not clear-cut. Section 4.1.1 describes how SCRIPTGARD addresses this problem. Next, we define what it means for a program to properly sanitize its inputs.

DEFINITION 7. *A server-side program $\mathcal{P} : \mathcal{I} \rightarrow \mathcal{O}$ defines a relation from untrusted user inputs I to output string $\mathcal{O}$. The program interprocedural data flow graph is a graph $\langle \mathcal{N}, \mathcal{E} \rangle$ with designated sets of nodes*

$$\langle Src, Snk, San \rangle \subseteq \langle \mathcal{N} \times \mathcal{N} \times \mathcal{N} \rangle$$

where Src are the sources that introduce an untrusted inputs in $\mathcal{P}$, Snk are the sinks that write strings to the output HTTP stream, and San are the sanitizers used by the program.

Without loss of generality, we assume that sink nodes either write untrusted strings to the output stream or trusted strings, but never strings containing both. Sink operations with mixed content can be translated to an equivalent dataflow graph with only exclusively trusted or untrusted sink nodes using sink node splitting: the output of a mixed string $Snk(q_1 + q_2 + \ldots + r_1 + \ldots q_n)$ can be split into a sequence of exclusive sink writes $Snk(q_1)$, $Snk(q_2) \ldots, Snk(r), \ldots, Snk(q_n)$.

DEFINITION 8. *An* untrusted execution trace *t of program $\mathcal{P}$ is a sequence of executed nodes*

$$\vec{t} = n_1, \ldots, n_k \in \mathcal{N}$$

such that $n_1 \in Src$, $n_k \in Snk$.

DEFINITION 9. *Let t be an untrusted execution trace $\vec{t} = n_1 \ldots n_k$ and let $\vec{f} = f_1, \ldots, f_m$ be a sequence of sanitizers such that $f_1, \ldots, f_m$ is a subsequence of $n_2, \ldots, n_{k-1}$.*

For all inputs $\mathcal{I}$, let O be the total output string just before the execution of the sink node in $\vec{t}$. We say that trace $\vec{t}$ is properly sanitized *if $\mathcal{O}$ induces context $\mathcal{C}$ and $\psi(\mathcal{C}) = \vec{f}$.*

In other words, for all possible trace executions, we require that the proper set of sanitizers be applied on trace for the expected parsing context. Note that *trusted* trace are allowed to change the browser context. A common example of that is

```
output.WriteLine("<SCRIPT>");
output.WriteLine("alert('hi');");
output.WriteLine("</SCRIPT>");
```

where each string is a sink and the first and third lines correspond to browser state transitions.

THEOREM 1. *If untrusted trace $\vec{t}$ is properly sanitized, assume the browser has read string $\mathcal{O}$ which induces context $\mathcal{C}$. Then reading the rest of the string output produced by $\vec{t}$ cannot induce any contexts $\mathcal{C}' \neq \mathcal{C}$.*

Proof: *Let input $\mathcal{I} = [\mathcal{I}_P \circ \mathcal{I}_{NP}]$. By Definition 9, for all input-output pairs $\mathcal{I}_P \rightarrow \mathcal{O}$, $\vec{t}$ contains sanitizers $\vec{f}$ correct for any context $\mathcal{C}$ inducible by $\mathcal{O}$. By Definition 6, we know that applying $\vec{f}$ to the remainder of the input $\mathcal{I}_{\mathcal{NP}}$ cannot leave context $\mathcal{C}$.* ■

For reasons of correctness, we wish to ensure that all untrusted execution traces are properly sanitized.

DEFINITION 10. *A server-side program $\mathcal{P}$ is* properly sanitized *if for every untrusted execution trace $\vec{t}$ of $\mathcal{P}$, $\vec{t}$ is properly sanitized.*

As an obvious corollary, if the program is properly sanitized, then no untrusted input to the server program can force the browser to change its context.

4. Analysis and Runtime Recovery

So far, we have discussed the abstract formalization of sanitizer placement. This section focuses on what it takes to get SCRIPTGARD to work in practice. The overall architecture for SCRIPTGARD is shown in Figure 3. SCRIPTGARD has two main components: (a) a pre-deployment analysis phase, and (b) a runtime auto-sanitization componenent. The results of our analysis are cached in a *sanitization cache*, which serves a basis for runtime sanitization of the application during deployed operation. The primary motivation for this architecture is to enable separation of expensive analysis to be performed prior to deployment, leaving only low-overhead components enabled at runtime.

SCRIPTGARD requires a map between browser parsing contexts and sanitizer functions appropriate for those contexts. In practice this map is specified by security architects or other experts and can be done once and for all. The sanitizers fall into two broad classes.

1. sanitizers that restrict untrusted data to be syntactically confined to one sub-grammar (or even further restrictively, to a set of permitted non-terminals in a sub-grammar).
2. sanitizers that compensate for the transduction along grammar transition productions.

Figure 8 shows the example sanitization specification for the running example as well as the applications we study. In our example application, an example of (1) is a sanitizer called `SimpleHTMLFormatting`, that transforms the input such that its output can only contain certain permitted HTML tags such as `<b>`, `<a>` and so on. An example of (2) is a function called `EcmaScriptStringEncode`. This function takes JavaScript literals and converts them to Unicode. Such conversion is necessary because the JavaScript parser converts Unicode to some other representation for string data. Another example of (2) is function `UrlPathEncode` which performs percent-encoding. This percent encoding is required because the URL parser will decode URLs on entry to the parser.

Recall that correctness of a sequence of sanitizers is a property of the execution trace. SCRIPTGARD is capable of determining the correct sequence given only knowledge of which sanitizers are appropriate for individual parsing contexts.

Given a trace of execution, SCRIPTGARD is capable of mapping the trace to a static program path. This allows us to determine the correct sequence of sanitizers that should be applied on this program path.

Reasoning about the localized correctness and completeness properties of the context-sanitizer mapping is an independent problem of interest; techniques for such correctness checking are an area of active research [1, 20, 15]. In this work, we assume the functional completeness and correctness for the specifications.

4.1 Sanitizer Placement Analysis

As explained in Section 2, applying sanitization to early or too late may result in inconsistent sanitization. Our key observation highlighted in Figure 2 is that the sanitizer placement must be *path-sensitive* due to the path-sensitive nature of HTML output in real-world applications we study.

SCRIPTGARD employs a dynamic analysis which analyses each executed path as a sequence of traces (as defined in Section 3). Each trace is conceptually a sequence of dataflow computation operations that end in a write to the HTTP stream. SCRIPTGARD's dynamic analysis aims to check if the sanitization applied on the any untrusted trace is correct. For each untrusted trace observed during program execution, SCRIPTGARD determines

- a mapping for each program trace $\vec{t}$ to a sequence of sanitizer functions, $f_1, f_2, \ldots, f_k$, to apply, and
- the portion of the output string that should be sanitizer

We call the first step *context inference* and the second step is achieved by a technique called *positive taint-tracking*. If the sequence of sanitizers applied on a trace does not match the inferred sequence, SCRIPTGARD discovers a violating path and it adds the corrected sanitizer sequence for this path to the sanitization cache. We describe these two steps in more detail next.

4.1.1 Positive Information Flow

So far, we discussed untrusted execution traces in the abstract, we have not talked about how SCRIPTGARD determines which traces are untrusted and, therefore, need to be sanitized. In real-world applications, exhaustively identifying all the sources of untrusted data can be challenging [22]. Recent work has shown that failure to identify non-web related channels, such as data read from file-system, results in cross-channel scripting

attacks [5].

Instead of risking having an incomplete specification, thereby missing potential vulnerabilities, SCRIPTGARD takes a conservative approach to identifying untrusted data: it employs *positive information flow*, which is modification of traditional (or negative information flow) used in several previous systems [32, 27, 13, 21, 35, 36]. Instead of tracking untrusted (negative) data as it propagates in the program, we track all safe data. Positive information flow is conservative because if input specifications are incomplete, unknown sources of data are treated as unsafe by default.

The source of "taint" in positive tainting are constants in the program source, such as integers and strings as well as certain input interfaces that are known to be safe from attacker control (such as the System GUID generator, random-number generator, and so on). In the applications that we have studied, the conservative sanitization has not impacted the intended functionality and has led us to identify several "gray" sources of potentially untrusted data. For instance, if the application interfaces with an central LDAP authentication system that stores users info or the Windows logon identity database, this could be an untrusted channel if we consider insider threats. Similarly, OS filenames or paths may be treated as untrusted if the application offers a non-web channel (such as FTP) for untrusted inputs [5]. In addition, data may pass through library modules which may implemented in other languages or as binaries. The problem of a heterogeneous code base spanning multiple languages is a reality in real-world application, and are a significant challenge to the adoption of language-based techniques of data tracking. Positive taint-tracking conservatively reasons about these flows of information through untracked code modules. We argue that conservative nature of positive information flow lends itself to safer refinements of specifications for identifying untrusted data, as compared to relying on manual identification of all sources of untrusted data a priori.

Implementation. Due to the string-heavy nature of the application and problem domain, we focus our implementation on tracking positive taint for strings. The .NET runtime supports two kinds of string objects: *mutable* and *immutable* objects. Immutable objects, instances of the `System.String` class, are called so because their value cannot be modified once it has been created [8]. Methods that appear to modify a `String` actually return a new `String` containing the modification. The .NET language also defines mutable strings with its `System.Text.StringBuilder` class, which allows in-place modification of string value; but all access to the characters in its value are mediated through methods of this class [9]. In essence, all strings in .NET are objects, whose values are accessed through public methods of the class — the language does support a primitive `string` type but the compiler converts `string` type to the `String` object and uses class methods whenever the value of a primitive `string` type is manipulated.

Using the encapsulation features offered by the language, we have implemented the taint status for each string object rather than keeping a bit for each character. The taint status of each string object maintains metadata that identifies if the string is untrusted and if so, the portion of the string that is untrusted. Our implementation maintains a weak hash table for each object, which keys on weak references to objects, so that our instrumentation does not interfere with the garbage collection of the original application objects and scales in size. Entries to freed objects are therefore automatically dropped.Taint prorogation, capturing direct data dependencies between string objects, is implemented by using wrapper functions for all operations in string classes. Each wrapper function updates the taint status of string objects at runtime.

We use CCI metadata [34], a robust static .NET binary rewriting infrastructure to instrument each call to the string object constructors and taint prorogation methods. The .NET language is a stack-based language and CCI Metadata provides the ability to interpose on any code block and statically rewrite it. Using this basic facility, we have implemented a library that allows caller instrumentation of specified functions, which allows redirection of original method calls to static wrapper methods of a user-defined class. Redirection of virtual function calls is handled the same way as static calls with the exception that the wrapper function accepts the instance object (sometimes referred to as the `this` parameter) is received as the first argument to the wrapper function. The user defined class is implemented seperately as a C# code and a reference to the user-defined .NET DLL is added to the original application via CCI's rewriting API.

Soundness Considerations. We explain how our positive information flow implementation is sound, i.e., does not miss identifying untrusted data, with exceptions identified in point 5 below. We show that these exceptions are rare in our test program.

1. The language encapsulation features provide the guarantee that all access to the string values are only permitted through the invocation of methods defined in the string classes.
2. All constant strings are created by invoking the constructors for the string classes or by creating a primitive string type using the operation `ldstr`. Any modification to the primitive value by the program is compiled to a conversion to an object of the string class, which invokes the string class constructors. Thus,

we can safely track all sources of taint by instrumenting these constructors.

3. Conversion between the two string classes is possible. This involves the call to the `Object.ToString` generic method. Statically, we instrument all these calls, and use .NET's built-in reflection at runtime to identify if the dynamic instance of the object being converted to a string and perform the taint metadata update.
4. The string classes `System.String` and `System.Text.StringBuilder` are both *sealed* classes; that is, cannot be inherited by other classes. This eliminates the possibility that objects that we do not track could invoke methods on the string classes.
5. String class constructors which convert values from non-string types are treated as safe (or positively tainted) by default. This is because we do not currently track taint status for these types presently. In principle, this is a source of potential unsoundness in our implementation. For example, the following code will lead our tracking to treat untrusted data as trusted:

```
String untrusted = Request.RawUrl;
var x = untrusted.ToCharArray();
....
String outputstr = new String(x);
httpw.Write(outputstr);
```

   Fortunately, these constructors are rare in practice. Figure 6 shows the distribution of functions instrumented by SCRIPTGARD. The key finding is that potentially unsound constructions occur only in 42 out of 23,244 functions instrumented for our application. Our implementation ignores this source of false negatives presently; we can imagine verifying that these do not interfere with our needs using additional static analysis or implement more elaborate taint-tracking in the future.

Output. The result of SCRIPTGARD's analysis is three pieces of information. First, SCRIPTGARD marks the portion of the server's output which is not positively tainted. The untrusted texts are delimited using special markers consisting of characters that are outside the alphabet used by the application legitimately. Second, for each string written to the HTTP output stream, it records the sequence of propagators (such as string concatenation, format-string based substitution) applied on the output text fragment. In essence, this allows SCRIPTGARD to (a) separate strings that are used for constructing HTML output templates from other strings, and, (b) identify the propogator that places the untrusted data into an output template. Third, it records a path string identifying the control flow path leading up each HTML output sink operation.

In addition to the above information, SCRIPTGARD's gathers the sequence of sanitizers applied to a given untrusted data. To do this, each sanitizer is instrumented similarly to surround the input data with additional special markup identifying that sanitizer's application to the input data. The knowledge of the untrusted data along with the nesting of sanitizers is thus encoded in the HTML output of the server. This output is then subject to the context inference step, which is described next.

4.1.2 Context Inference

For a given analyzed path the server outputs a HTML response encoding the information identifying sub-strings that are untrusted, as well as, the sequence of sanitizers applied. SCRIPTGARD employs a web browser to determine the contexts in which untrusted data is placed, in order to check if the sanitization sequence is consistent with the required sequence.

In our implementation, we use the Microsoft Research "Cloud Computing Client," or C3. C3 includes an HTML 5 compliant parser that has been developed from scratch to be as close to the current specification as possible, plus a fast JavaScript engine. The C3 parser takes an HTML page as input. In the page, untrusted data is identified by special markup. The special markup is introduced at the server's output by our positive taint-tracking.

We augment the C3 parser to track the sequence of contexts (or sub-grammars) in which data marked as untrusted appears. In our implementation for HTML, we treat each context to be the (a) state of the lexer (as defined in the HTML 5 draft specification), (b) the stack of open elements (as defined in the HTML 5 draft specification), and (c) specific information about the local state of parse tree (such as the name of current tag or attribute being processed).

We apply techniques similar to string accenting for tracking untrusted data in other contexts [6]; DOM nodes that correspond to untrusted data are also marked (or accented). Similar context tracking is applied for the JavaScript parser. For the policy of our applications and the policies identified in previous work [23], we have found this level of tracking to be adequate. As result of this tracking, the browser outputs the context information for each untrusted data: a list of sub-grammars that parse untrusted data and the sequence of transitions between them.

Determining a sanitization sequence. Using context information, SCRIPTGARD decides the correct sequence of sanitizers to apply for a given untrusted execution trace. Specifically, it tracks which sub-grammars (or contexts) is the untrusted string parsed in and their sequence. To determine the correct sanitizer sequence (constructing ψ on demand), SCRIPTGARD applies for each context in the context-chain (in order of the sequence): (a) the appropriate sanitizer from the input sanitization specification, and (b) an appropriate sanitizer to account for the transduction of the data associated with the sub-grammar transition. The inferred chain of sanitizers is ensured to be of correct ordering as that of the context-chain, thereby eliminate multiple sanitization errors that could have resulted from manual placement.

Sanitizers that are rendered superflous due to the application of a sanitizer previously in the inferred chain can be removed. For instance, the application of sanitizer `EcmaScriptStringEncode` makes subsequent applications of `HtmlEncode` superfluous as the set of characters transformed by the former to unicode encoded format is a strict superset of the latter. This subsetting relation between the sanitizers is precomputed for the sanitizers; for the sanitizers defined in Figure 8, we have precomputed this relation to be $K \subset E \subset P \subset H \subset S$ and $A \subset H$, where K, E, P, H, S, A are sanitizers `UrlKeyValueEncode`, `EcmaScriptStringEncode`, `UrlPathEncode`, `HtmlEncode`, `SimpleHtmlFormatting` and `HtmlAttribEncode`. We point out that this step is not necessary for the sanitizer chain correctness, but it optimizes the inferred chain. This final elimination of superfluous sanitizers as per these constraints determines the correct sanitizer sequence that is consistent with the context-chain. The correct sanitizer chains inferred for the example in Figure 2 is given below:

path	Sanitizer sequence
Path 1	`EcmaScriptStringEncode`, `EcmaScriptStringEncode`
Path 2	`EcmaScriptStringEncode`
Path 3	`HtmlAttribEncode`
Path 4	`HtmlAttribEncode`, `EcmaScriptStringEncode`

4.2 Automatic Sanitization at Runtime

We now sketch how sanitization can be applied at runtime in a production system. We stress that we have not yet benchmarked this proposal, but it shows a way to reduce runtime overhead.

Context inference determines for a given substring in the server's HTML output, whether the applied sequence is correct or not. If the sanitizers applied on the path do not match the inferred sanitizer chain, then tool checks whether the applied sequence could be dangerous or is merely superfluous.

If the sanitizer chain is dangerous, the execution path is reported as a *violating path*. For each violating path, the correct sequence of sanitizers is added to a *sanitization cache.*

Recall that the runtime instrumentation encodes information about the executed path string until the HTML sink operation, say S, that outputs the substring. In addition, the encoding also identifies the code location of the dataflow propagator (say P) (typically, a string concatenation or format-string based string substitution) that places the untrusted data in the trusted output template emitted at S. In other words, we view the output template construction as a sequence of string propagators. P is the propogator the embeds the untrusted string into the template.

The auto-sanitization scheme is as follows. SCRIPTGARD enables a low-overhead runtime path detector in the application. The path detector tracks the execution path in the application. The path executed uptil the sink S is checked in the sanitization cache. If an entry exists, SCRIPTGARD sautomatically sanitizes the untrusted substring(s) in the output template emitted at S. If path is not in the sanitization cache, SCRIPTGARD leaves the application's behavior unchanged.

Rewriting untrusted output. To achieve the path-sensitive output rewriting, SCRIPTGARD maintains a *shadow copy* of the untrusted data in addition to the actual value that the program computes. If the path is not in the sanitization cache, the actual value of the untrusted data is used at S. However, if the path is in the sanitization cache, the shadow copy is selected, and applied the correct sanitizer sequence before emiting at S. In our application, output template-constructing operations typically consist of string concatenation and format-string based substitutions, which do not perform arbitrary transformation on the HTML template string. This enables the following instrumentation-based program transformation.

1. At string propogator P, create a shadow copy of the untrusted data. A simple way to achieve this is the create two copies of the untrusted value, surround the values with special "*unforgable markup*", and concatenate them both in the template. The first value is the shadow copy, while the second will be the actual value. To make the markup resistent to attacks, we plan to use markup randomization [26].
2. Sanitizers are re-implemented to be markup-aware. Specifically, for untrusted content identified by special markup, the sanitizers will only transform the actual copy, leaving the shadow copy untouched.
3. At S, if the path executed is in the sanitization cache, SCRIPTGARD strips the actual value and markup out, applies the sanitization on the shadow copy and writes it to the output stream.
4. At S, if the path is not in the sanitization cache,SCRIPTGARD strips the special markup and the shadow value out, and writes the actual value to the output stream thereby leaving the application behavior unchanged for paths that are safe or not analyzed.

Consider the example in Figure 1 when following "Path 4." The proposed system would first detect that the path taken is "Path 4." Then the system would look up in the sanitization cache the correct sequence of sanitizers. As we showed above, this sequence is `HtmlAttribEncode`, followed by `EcmaScriptStringEncode`. The untrusted text is surrounded with special markup at the string concatenation in function `DynamicLink.RenderControl` and the shadow copy is created. This allows SCRIPTGARD to automatically apply the two sanitizers in sequence at the HTML output sink when it is determined that path 4 is executed.

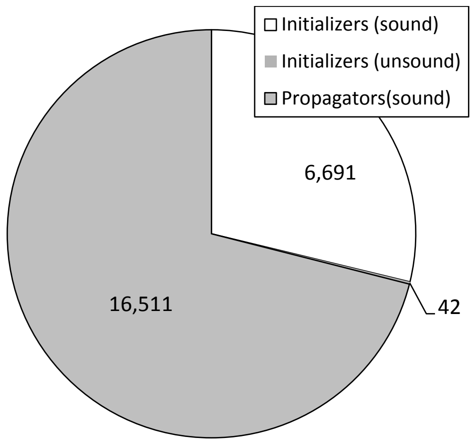


Figure 6. Classification of functions used for SCRIPTGARD instrumentation.

HTML Sink Context	Correct sanitizer that suffices
HTML Tag Context	`HTMLEncode`, `SimpleTextFormatting`
Double Quoted Attribute	`HTMLAttribEncode`
Single Quoted Attribute	`HTMLAttribEncode`
URL Path attribute	`URLPathEncode`[1]
URL Key-Value Pair	`URLKeyValueEncode`
In Script String	`EcmaScriptStringEncode`
CDATA	`HTMLEncode`
Style	`Alpha − numerics`

Figure 8. Sanitizer-to-context mapping for our test application.

For our low overhead path detector, we plan to use techniques from the Holmes project. That project developed low-overhead techniques for detecting paths related to bugs of interest encountered during a testing phase. They show overheads of less than 3% in some cases, because the branches related to the bugs of interest can be isolated and tested for quickly.

5. Evaluation

This section is organized as follows. Section 5.1 describes our experimental setup. Section 5.2 details the results of SCRIPTGARD analysis and presents our findings.

5.1 Experimental Setup

Our evaluation focuses on a large legacy application of over 400,000 lines of lines of server-side C# code. To perform the runtime training described in Section 4, we manually interacted with each of the SCRIPTGARD-enabled applications exploring distinct features and configuration settings of each applications. A current version of the platform was installed on a Microsoft Windows Server 2008 workstation. We used the application's debug configuration. Our test system also had an install of Microsoft SQL Server to hold the application database and connected to a server set up to manage user identities. All three components of the testbed were previously tested to ensure maximum compatibility.

In the course of this runtime interaction, we accessed 53 distinct web pages, which we subjected to SCRIPTGARD analysis. Figure 7 shows the size of the various web pages in terms of the number of

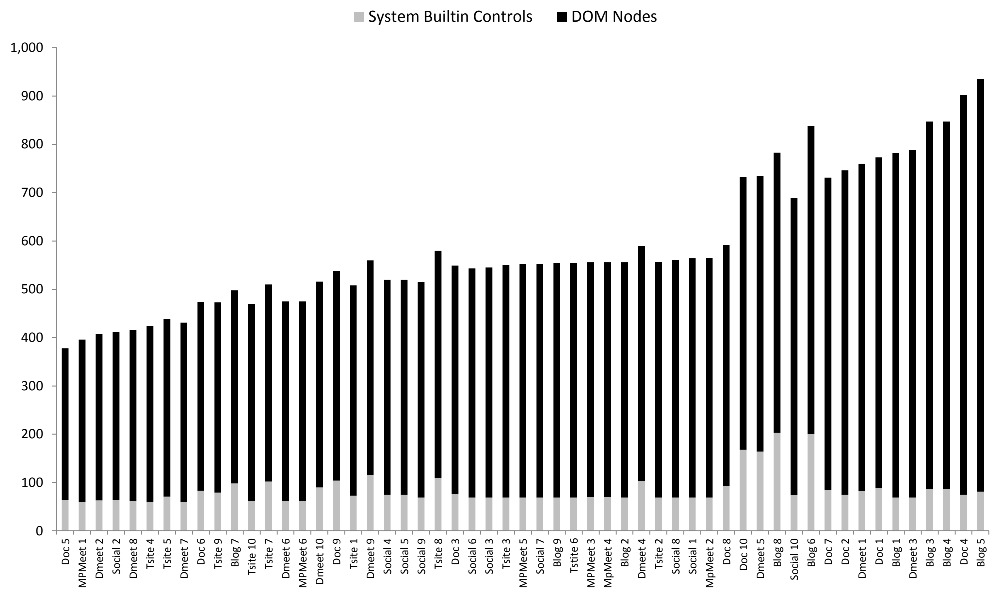


Figure 7. Distribution of DOM sizes, in nodes, across our training HTML pages.

DOM nodes they generate from their initial HTML output (ignoring dynamic updates to the DOM via JavaScript, etc.). Typically, page sizes range from 350 to 900 nodes.

In addition to using the built-in rendering features (web and HTML controls) provided by an underlying web framework, our application defines and uses custom objects that handle their own rendering. Figure 7 indicates that a majority of the DOM nodes in the applications' outputs are derived from custom controls. The language-based solution of SCRIPTGARD, as opposed to a framework-based solution, allows it to be directly applicable to ensuring correctness of custom features of enterprise applications.

The application permits only a small set of contexts for embedding untrusted data. Figure 8 shows the mapping between contexts and sanitization functions; we note this is a strict subset of the mapping defined in previous work [23]. In particular, it does not permit unquoted attribute values, which are known to be notoriously hard to sanitize, and permits only quoted attributes which have well-defined rules for sanitization [28]. Furthermore, it always sets the page encoding to UTF-8, eliminating the possibility of character-set encoding attacks [14]. The tested application also does not use XHTML pages — several browsers automatically HTML entity-decode attribute values and in HTML elements, thereby undermining the assumptions made by its sanitization functions. Admittedly, the sanitizer-context mapping is quite tricky to construct. We arrived at the result in Figure 8 after several interactions with the application's product team and its security engineers.

Note that the application we tested relies on its own custom sanitizers, as shown in Figure 8, which cover application-specific issues. These custom sanitizers have been subject to severe internal security audits. For the purposes of this work, we assume their individual functional correctness, i.e. they work correctly for a specific context. As argued previously, a range of complementary techniques exist that focus on ensuring this correctness.

These sanitizers are similar to those found in several other major web platforms such as Google CDT, AutoEscape, OWASP recommendations, or the Microsoft Anti-XSS library. We believe the SCRIPTGARD approach would also apply to these platforms.

5.2 Analysis Results

This section describes our findings on the basis of running 53 SCRIPTGARD-annotated pages and looking for inconsistent and multiple sanitization errors described in Section 2.4. Our data demonstrates that manual sanitizer placement is prone to errors, even if the sanitizers are individually correct. Next, we discuss the two classes of errors SCRIPTGARD found as a result of context-sanitizer mismatches.

5.2.1 Inconsistent sanitization

Figure 9 shows that SCRIPTGARD exercised 25,209 paths on which sanitization was applied. Of these, 1,558 paths (or 6.1%) were improperly sanitized. Of these improperly sanitized paths, 1207 (4.7% of the total analyzed paths) contained data that could not be proved safe by our positive taint tracking infrastructure, so therefore are candidates for runtime automatic choice of sanitization.

We used Red Gate's .NET Reflector tool, combined with other decompilation tools, to further investigate the executions which SCRIPTGARD reported as improperly sanitized. Our subsequent investigation reveals that errors result because it is difficult to manually analyze the calling context in which a particular potion of code may be invoked. In particular, the source and the sink may be separated by several intervening functions. Since SCRIPTGARD instruments all string operations, we can count how far sources and sinks are removed from each other. In Figure 12, we graph the distribution of these lengths for a randomly selected sample of untrusted paths. This shows that a significant fraction of the chains are long and over 200 of them exceed 5 steps.

Our data on the length of def-use chains is consistent with those reported in previous static analysis based works [21]. As explained in Section 2, the sharing of dataflow paths can result in further ambiguity in distinguishing context at the HTML output point in the server, as well as, in distinguishing trusted data from untrusted data. As a result, we observed some paths along which sanitization of trusted strings was performed. In our investigation we observed the following cases:

- In several cases, a single sanitizer was applied, but the sanitizer did not match the context. In almost all of such cases analyzed,

Web Page	Sanitized Paths	Inconsistently sanitized Total	Highlight
Home	396	14	9
A1 P1	565	28	22
A1 P2	336	16	11
A1 P3	992	26	21
A1 P4	297	44	35
A1 P5	482	22	17
A1 P6	436	23	18
A1 P7	403	19	13
A1 P8	255	22	18
A1 P9	214	16	12
A1 P10	1,623	18	14
A2 P1	315	16	12
A2 P2	736	53	47
A2 P3	261	21	16
A2 P4	197	16	12
A2 P5	182	22	18
A2 P6	237	22	18
A2 P7	632	20	16
A2 P8	450	23	19
A2 P9	802	26	22
A3 P1	589	25	21
A3 P2	2,268	18	14
A3 P3	389	16	12
A3 P4	477	103	15
A3 P5	323	24	20
A3 P6	292	51	45
A3 P7	219	16	12
A3 P8	691	25	21
A3 P9	173	16	12
A4 P1	301	24	20
A4 P2	231	30	25
A4 P3	271	28	22
A4 P4	436	38	32
A4 P5	956	36	24
A4 P6	193	24	18
A4 P7	230	36	32
A4 P8	310	24	20
A4 P9	200	24	18
A4 P10	208	24	20
A4 P11	498	34	29
A4 P12	579	34	29
A4 P13	295	25	20
A4 P14	591	104	91
A5 P1	604	61	55
A5 P2	376	25	21
A5 P3	376	25	21
A5 P4	401	26	21
A5 P5	565	31	26
A5 P6	493	34	29
A5 P7	521	34	29
A5 P8	427	24	20
A5 P9	413	24	20
A5 P10	502	28	23
Total	25,209	1,558	1,207

Figure 9. Characterization of the fraction of the paths that were inconsistenly sanitized. The right-most column indicates which fraction of those paths were highlighted as not be proved to use only safe values during our dynamic testing.

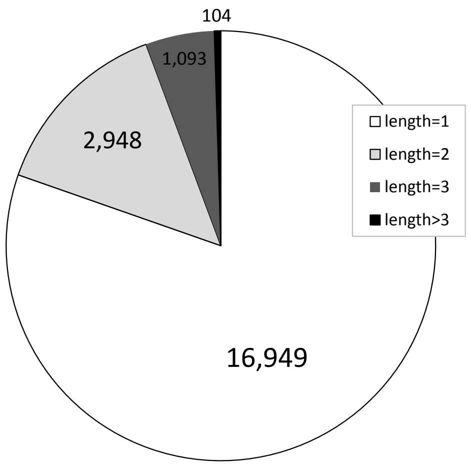


Figure 10. Distribution of the lengths of applied sanitization chains, showing a sizeable fraction of the paths have more than one sanitizer applied.

the sanitizer applied was in a different function from the one that constructed the HTML template in which to embed the untrusted data. This suggests that developers may not fully understand how the context — a global property — impacts the choice of sanitizer, which is a local property. This is not surprising, given the complexity of choices in Figure 8.

- A small fraction of cases sanitized trusted data. While this is unlikely to lead to a vulnerability, we still report these cases because they point to developer confusion. On further investigation, we determined this was because sinks corresponding to these executions were shared by several dataflow paths. Each such sink node could render potentially untrusted data on some executions, while rendering purely trusted data on others.
- In some cases more than one sanitizer was applied, but the applied sanitizers was not correct for the calling context of the sanitizer in which the data was placed [2].

5.2.2 Inconsistent Multiple Sanitization

Nesting of parsing contexts is fairly common. For example a URL may be nested within an HTML attribute. This nesting may require multiple sanitizers to correctly filter untrusted inputs. Figure 10 shows the histogram of sanitizer sequence lengths observed. The inferred context for a majority of these sinks demanded the use of multiple sanitizers.

We observe that none of sanitizers employed by the tested application are *commutative.* The sanitizers produce different outputs if composed in different orders. In addition, none of them are idempotent, that is, repeated application of the same sanitizer results in different outputs. As a result, if the developer determines the context correctly but applies an incorrect ordering of sanitizers, it could affect the application's intended functionality and even introduce vulnerabilities. Yet, as Figure 11 shows, the use of multiple sanitizers in the application is widespread, with sanitizer sequences such as `UrlPathEncode` `HtmlEncode` being most popular.

We found a total of 3,245 paths with more than one sanitizer. Of these, 285 (or 8%) of the paths with multiple sanitization were inconsistent. The inconsistent paths fell into two categories: first, 273 instances of

[2] Errors where the combination was correct but the ordering was inconsistent with the context are reported as inconsistent multiple sanitization errors.

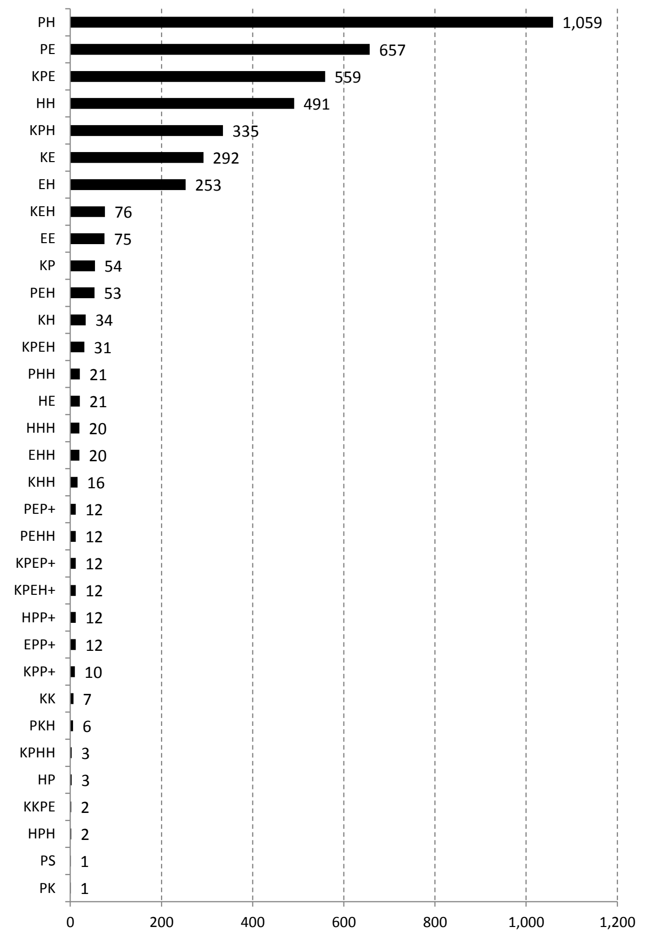


Figure 11. Histogram of sanitizer sequences consisting of 2 or more sanitizers empirically observed in analysis, characterizing sanitization practices resulting from manual sanitizer placement. E,H,U, K,P,S denote sanitizers `EcmaScriptStringLiteralEncode`, `HtmlEncode`, `HtmlAttribEncode`, `UrlKeyValueEncode`, `UrlPathEncode`,`SimpleHtmlEncode` respectively.

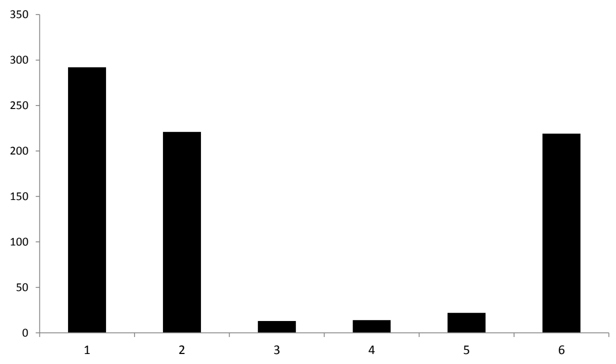


Figure 12. Distribution of lengths of paths that could not be proved safe. Each hop in the path is a string propagation function. The longer the chain, the more removed are taint sources from taint sinks.

`(EcmaScriptStringLiteralEncode ) (HtmlEncode)+` pattern. This pattern is inconsistent, as explained in example 2 in Section 2.4. Second, we found 12 instances of the `(EcmaScriptStringLiteralEncode} (UrlPathEncode)+` pattern. This pattern is inconsistent because it does not properly handle sanitization of URL parameters.

We found an additional 498 instances of multiple sanitization that were superfluous, i.e., sanitizer A applied before sanitizer B already nullified attacks, rendering sanitization B superfluous. While not a security bug, this multiple sanitization could break the intended functionality of the applications. For example, repeated use of `UrlKeyValueEncode` could lead to multiple percent encoding causing broken URLs. Repeated use of `HtmlEncode` could lead to multiple HTML entity-encoding causing incorrect rendering of output HTML.

Other attacks involving multiple URL encoded inputs can also be dangerous. SCRIPTGARD's formalism can handle these errors; however, our implementation does not track the flow of data from the server to browser and back. The issue of multiple percent encoding could be dangerous if the applied decodes on the server-side do not match the number of encodes. We leave automatic identification and repair of those cases to future work. For the purposes of this section, we report such cases as safe.

We emphasize that in our tests, none of the inconsistent sanitizations led to actual security vulnerabilities. Had we done so, we would have contacted the vendor of the tested application and revealed our findings to that company. We believe this reflects the quality of previous testing of the application. Furthermore, as we discuss in more detail in the next section, the goal of SCRIPTGARD is not to find vulnerabilities. Instead, the goal of our approach is to mitigate all potential vulnerabilites with provable guarantees.

6. Discussion

SCRIPTGARD provides a systematic foundation for ensuring sound sanitization that can eliminate scripting attacks in web applications by construction. In this section, we explain some of design choices and discuss their impact on the results.

Bug Finding vs. Mitigation. SCRIPTGARD has discovered several instances of paths along which the sanitization applied was inconsistent. On further inspection, however, none of these paths have led to real security issues. The reason is that our positive tainting is conservative: if we mark some data as untrusted, it does not necessarily imply that the adversary controls the data. Contrast our information flow tracking with negative taint tracking, which traces an execution directly from the adversary-controlled input to a taint sink.

Recall that SCRIPTGARD's goal is not to find bugs. Instead, SCRIPTGARD aims to preclude bugs from having a security-critical effect at modest cost. SCRIPTGARD's positive information flow is a way to separate trusted HTML template code from the (potentially untrusted) variables at the output. This goal and conservative treatment of non-constant data is consistent with other frameworks, such as Django and Google Auto-escape, that aim to secure web applications by construction. Many previous techniques have assumed the specifications of untrusted input as complete.

Specifying all the sources of untrusted inputs is a serious practical challenge in complex systems and has not received due attention in research [22]. While identifying some sources of untrusted data is acceptable for a bug-finding tool, for SCRIPTGARD missing untrusted data leads to missing attacks. Finally, we believe that SCRIPTGARD, when used as an aid to human security analysis, is useful because it suggests corrections of paths that are already sanitizing data but in an inconsistent way.

Browser Variations. Our implementation uses one specific HTML5 compliant parser, the "Cloud Computing Client" developed at Microsoft Research. The Cloud Computing Client was developed from scratch to be as close to the HTML5 specification as possible, using memory safe languages and runtime correctness enforcing techniques such as code contracts. However, this does not guarantee that we will parse HTML identically to other browsers. Variants among browsers are a notorious source of problems in sanitization [38].

The problems in sanitization arising from browser variation fall into two classes in our context. The first class consists of sanitizers which are not correct for a given context because of a browser variation. This class falls under the question of correctness of the sanitizer, which we have assumed as a basis of this work. Extensive community effort [28, 38], internal scrutiny in industrial code as well as research works [20, 1] have focussed on developing sanitizers for common HTML contexts that work across a majority of the present browsers. The second class, however, arises if browsers disagree on the parsing context for a specific untrusted string. In that case, the correct sequence of sanitizers depends on which browser specifically is in use.

To address these problems, we imagine but have not implemented a design where multiple parsers are implemented in SCRIPTGARD. At runtime, the specific parser to use can be determined by querying the User-Agent string of the browser. Of course, browsers are free to lie about their User-Agent. Because this could only lead to additional attacks on the browser, there is no incentive for a user to do so.

7. Related Work

Related work falls into three broad categories. First, we discuss mitigations, both browser and server based. Second, we discuss bug finding in web applications using programming language techniques. Finally, we describe previous work on sanitizer correctness.

7.1 Mitigations

Our work follows on previous attempts to build security mitigations into web browsers and web applications. Browser-based mitigations, such as Noncespaces, XSS Auditor, or "script accenting," make changes to the web browser that make it difficult for an adversary's script to run [26, 12, 2]. These approaches can be fast, but they require all users to upgrade their web browsers, which is a slow and unevenly distributed process.

A server side mitigation, in contrast, focuses on changing the web application logic instead of changing the browser. BLUEPRINT is a recent example that describes mechanisms to ensure the safe construction of the intended HTML parse tree on the client using JavaScript in a browser-agnostic way [23]. These mechanisms, however, require a significant change to the way applications emit HTML, which may require a significant application and associated runtime library rewrite. This is a major concern for legacy applications. While a manual enforcement of BLUEPRINT's primitives was shown possible, the issue of applicability of these primitives to large existing applications is still at large.

In contrast, SCRIPTGARD abstraction is closer to what legacy applications implicitly already enforce — SCRIPTGARD focuses on fortifying sanitizer-based defense. We detect mismatches between sanitizer and the context in which the sanitizer is used and sketch a design that dynamically picks the correct sanitizer at runtime, leveraging the fact that developers have already indicated the points in need of sanitization. Our fine-grained incremental deployment does not, therefore, require a major output rewriting engine.

XSS-GUARD [4] proposes techniques to learn allowed scripts from unintended scripts. The allowed scripts are then whitelisted to filter out unintended scripts at runtime. Like SCRIPTGARD, it employs a web browser for its analysis, but the two approaches are fundamentally different. SCRIPTGARD's defense is based on automatic server-side sanitizer placement, rather than browser-based whitelisting of scripting in server output. XSS-GUARD's techniques are intended for applications that allow rich HTML, where designing correct sanitizers becomes challenging. SCRIPTGARD target applications with fairly restrictive policies that have already addressed the sanitizer-correctness issue; we empirically motivate the orthogonal aspect of sanitizer placement.

Securifly translates bug specifications written in a special program query to runtime instrumentation that detects these bugs [24]. Just as in SCRIPTGARD, this approach allows making guarantees about runtime correctness, relative to the correctness of the specification and the instrumentation. Our approach, however, uses a web browser implementation to allow us to reason about the browser parsing context as well as the server state. This allows us to tackle sanitizer placement problems that can not be reasoned about using the server code alone.

7.2 Software Security Analysis of Web Applications

Software security focuses on using program analysis to find security critical bugs in applications. The WebSSARI project pioneered these approaches for web applications. WebSSARI unsound static and dynamic analysis in the context of analyzing PHP programs [17]. WebSSARI has successfully been applied to find many SQL injection and cross-site scripting vulnerabilities in PHP code. Several projects that came after WebSSARI improve on the quality of static analysis for PHP [35, 19].

The Griffin project proposes a scalable and precise sound static and runtime analysis techniques for finding security vulnerabilities in large Java applications [21, 24]. Based on a vulnerability description, both a static checker and runtime instrumentation is generated. Static analysis in Griffin also reduces the overhead in most cases. The runtime system allows vulnerability recovery by applying user-provided sanitizers on execution paths that lack them. Multiple other systems for information flow tracking have been proposed, including Haldar *et al.*for Java [13] and Pietraszek *et al.* [29] and Nguyen-Tuong *et al.*for PHP [27].

Typically information flow systems use *negative tainting* to specifically identify untrusted data in web applications applications [36, 24, 21, 22]. While negative taint is preferable for finding bugs, it is less desirable for mitigations because it requires specifying all sources of taint. This can be challenging in a large legacy application. Our information flow design distinguishes itself from previous work in that it tracks *positive taint*, which is conservative default fail-close approach, and side-steps identifying all sources of taint.

A second distinguishing factor of our work is its focus on *both* browser and server state. Many previous systems focus purely on server state [21, 35, 19], Web 2.0 applications that make use of AJAX often fetch both data and JavaScript code from many sources. As a result, the final HTML is only available within the browser. Moreover, when executable content is transferred to the client as XML or JSON and composed into displayable HTML by JavaScript, previously proposed on-the-wire rewriting tactics become ineffective [37, 30].

To address this problem, multiple proposals have focused on changing the browser to add additional isolation mechanisms or programming abstractions [18, 11, 37, 16, 25]. A common theme is to give web developers isolation mechanisms similar to processes in operating systems. As we argued above, these approaches are important, but they require users to upgrade their web browsers while SCRIPTGARD does not.

Erlingsson *et al.*make an end-to-end argument for the client-side enforcement of security policies that apply to client behavior [11, 25]. Their proposed mechanisms use server-specified, programmatic security policies that allow for flexible client-side enforcement, even to the point of runtime data tainting. Unlike SCRIPTGARD, their technique can enforce some necessary, but not sufficient conditions for establishing distributed application integrity. Our approach can provide runtime guarantees, relative to the soundness of our browser implementation and our instrumentation. We do not, however, push enforcement into the browser itself.

7.3 Sanitizer Correctness

Correctness for a web sanitizer means that the sanitizer is present where needed, and that the sanitizer is correct for the given context. Livshits et al. developed methods for determing which functions in a program play the role of sanitizer [22]. Their Merlin system is also capable of detecting missing sanitizers. Balzarotti et. al show that sanitizer routines are often incorrect, using symbolic test generation techniques [1]. The Cross-Site Scripting Cheat Sheet shows over two hundred examples of strings that exercise common corner cases of web sanitizers [31]. Hooimeijer et al. focus on domain specific languages for writing sanitizers [15].

As we argued previously, SCRIPTGARD's analysis is complementary to this work. Sanitizers may be present, and they may be functionally correct for contexts they are intended to be used in. Incorrect placement, however, can introduce errors. To our knowledge, this class of errors has not been previously identified, nor have techniques to eliminate these errors been described.

8. Conclusions

SCRIPTGARD is motivated by the desire to prevent inconsistent sanitization from adversely affecting large web applications. Our approach is a combination of testing and low-overhead runtime recovery.

We evaluate both the testing approach as well as runtime auto-sanitization on a large-scale web application with over 400,00 lines of code. We performed our security testing on a set of 53 large web pages, each containing 350–900 DOM nodes. These pages were produced by 7 components of the test platform. We found 285 multiple-encoding issues, as well as 1207 instances of inconsistent sanitizers. SCRIPTGARD provides provable guarantees that the right sanitizer will be picked at runtime, avoiding all these issues.

Acknowledgments

We would like to give special thanks to Patrice Godefroid, who has been most instrumental in helping us define the problem that SCRIPTGARD solves. Hermann Venter provided invaluable assistance with the CCIMetadata infrstructure. Peli de Halleux and Nikolai Tillmann patiently answered questions regarding runtime profiling in .NET applications. Other Microsoft colleagues provided important feedback throughout the project, for which we are grateful.

References

[1] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

[2] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side xss filters. *International World Wide Web Conference (WWW)*, 2010.

[3] P. Bisht and V. N. Venkatakrishnan. XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.

[4] P. Bisht and V. N. Venkatakrishnan. XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.

[5] H. Bojinov, E. Bursztein, and D. Boneh. XCS: Cross channel scripting and its impact on web applications. In *CCS*, 2009.

[6] S. Chen, D. Ross, and Y.-M. Wang. An analysis of browser domain-isolation bugs and a light-weight defense mechanism. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2007.

[7] M. Corporation. Microsoft code analysis tool .NET, 2009. `http://www.microsoft.com/downloads/en/details.aspx?FamilyId=0178e2ef-9da8-445e-9348-c93f24cc9f9d&displaylang=en`,.

[8] M. Corporation. String class (system), 2010. `http://msdn.microsoft.com/en-us/library/system.string.aspx`.

[9] M. Corporation. Stringbuilder class, 2010. `http://msdn.microsoft.com/en-us/library/system.text.stringbuilder(v=VS.80).aspx`.

[10] H. . draft specification. Web addresses in html 5. `http://www.w3.org/html/wg/href/draft#ref-RFC3986`, 2009.

[11] U. Erlingsson, B. Livshits, and Y. Xie. End-to-end web application security. In *Hot Topics in Operating Systems (HotOS)*, 2007.

[12] M. V. Gundy and H. Chen. Noncespaces: using randomization to enforce information flow tracking and thwart cross-site scripting attacks. *16th Annual Network & Distributed System Security Symposium*, 2009.

[13] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for Java. In *Proceedings of the Annual Computer Security Applications Conference*, Dec. 2005.

[14] Y. HASEGAWA. Utf-7 xss cheat sheet. `http://openmya.hacker.jp/hasegawa/security/utf7cs.html`, 2005.

[15] P. Hooimeijer, M. Veanes, P. Saxena, and D. Molnar. Modeling imperative string operations with transducers. Technical report, Microsoft Research, 2010. MSR-TR-2010-96.

[16] J. Howell, C. Jackson, H. J. Wang, and X. Fan. MashupOS: Operating system abstractions for client mashups. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2007.

[17] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing Web application code by static analysis and runtime protection. In *Proceedings of the International Conference on World Wide Web*, 2004.

[18] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the International Conference on World Wide Web*, 2007.

[19] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting Web application vulnerabilities (short paper). In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[20] A. Kieżun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for string constraints. In *International Symposium on Software Testing and Analysis*, 2009.

[21] B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the Usenix Security Symposium*, 2005.

[22] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: Specification inference for explicit information flow problems. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2009.

[23] M. T. Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[24] M. Martin, B. Livshits, and M. S. Lam. SecuriFly: runtime vulnerability protection for Web applications. Technical report, Stanford University, 2006.

[25] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE*

Symposium on Security and Privacy, May 2010.

[26] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *NDSS*, 2009.

[27] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening Web applications using precise tainting. In *Proceedings of the IFIP International Information Security Conference*, 2005.

[28] Open Web Application Security Project. The ten most critical Web application security vulnerabilities. `http://umn.dl.sourceforge.net/sourceforge/owasp/OWASPTopTen2004.pdf`, 2004.

[29] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the Recent Advances in Intrusion Detection*, Sept. 2005.

[30] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proceedings of the Operating System Design and Implementation*, 2006.

[31] RSnake. XSS cheat sheet for filter evasion. `http://ha.ckers.org/xss.html`.

[32] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[33] Ter Louw, Mike and V.N. Venkatakrishnan. BluePrint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[34] H. Venter. Common compiler infrastructure: Metadata API, 2010. `http://ccimetadata.codeplex.com/`.

[35] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the Usenix Security Symposium*, 2006.

[36] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. *USENIX Security Symposium*, 2006.

[37] D. Yu, A. Changer, H. Inamura, and I. Serikov. Enabling better abstractions for secure server-side scripting. In *WWW*, 2008.

[38] M. Zalewski. *Browser Security Handbook*. Google Code, 2010.