

Don't use LMMs to produce prose

As the reader progresses through the following sections, they will gain valuable insights into the unique strengths and limitations of each fuzzing tool, in addition to gaining a comprehensive understanding of the security posture of the [REDACTED] tool. The findings in this report have the potential to contribute significantly to the overarching discourse on software security, enabling efforts aimed at increasing the resilience of critical software components.

By leveraging the formidable capabilities of fuzzing techniques and tools, this report represents a critical step in strengthening open-source applications like [REDACTED] and [REDACTED] and in strengthening the security and reliability of file processing within today's dynamic digital landscape.

Fuzzing project

marked on

- effort in fuzzing the target
- clarity of the report
 - in describing both the process and the results
 - don't use LLM to generate prose
- analysis of the results
- reflection on the process

Software Security

Secure **INPUT** handling

part 3

preventing XSS, incl. DOM-based XSS

Erik Poll

Digital Security

Radboud University Nijmegen

Encoding for the web - server-side

Many sites use **web templating framework** to generate web pages.

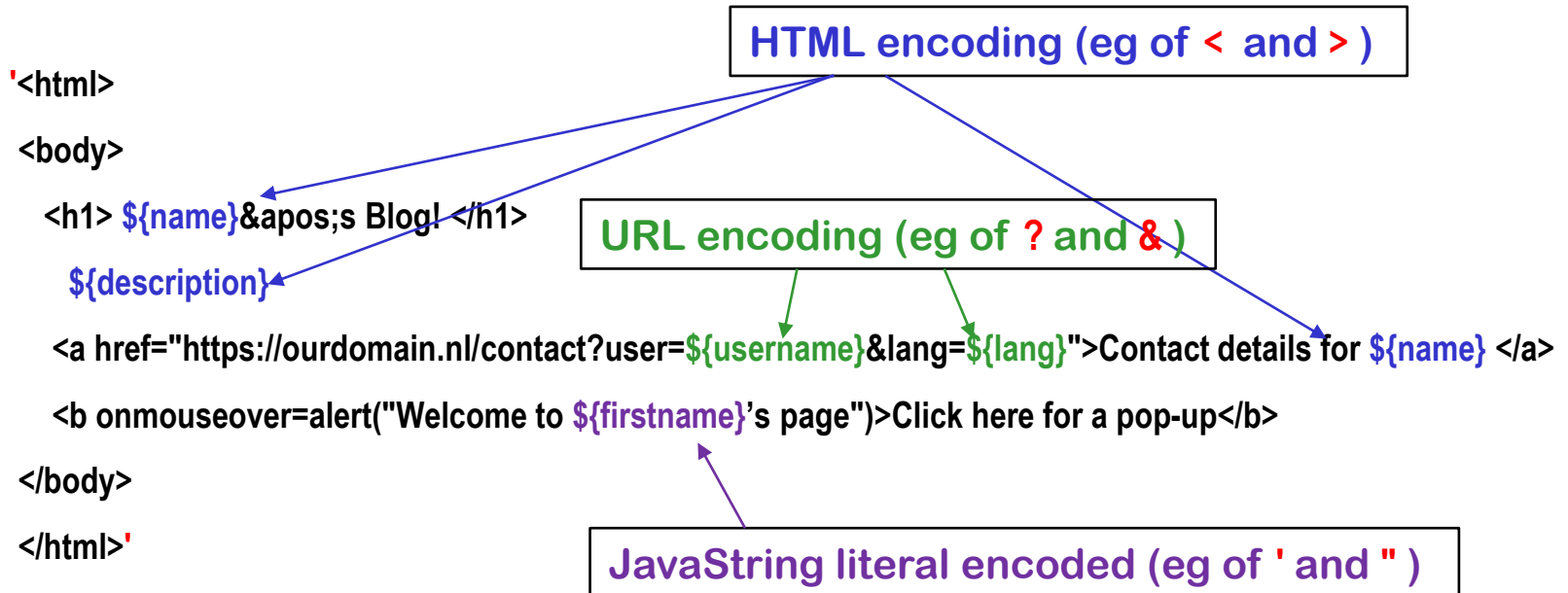
Below a web template for a web page with parameters written as **`${...}`**

```
1 '<html>
2 <body>
3   <h1> ${name}&apos;s Blog! </h1>
4   ${description}
5   <a href="https://ourdomain.nl/contact?user=${username}&lang=${lang}">User info for ${name} </a>
6   <b onmouseover=alert("Welcome to ${firstname}'s page")>Click here for a pop-up</b>
7 </body>
8 </html>'
```

Parameters should be **properly encoded** by the web server (aka web template engine) before being substitution.

How should the parameters be encoded here?

Encoding for the web - server-side



- Variants of these encodings may be needed,
eg to still allow *some* HTML mark-up in the \${description}
- All these encodings can be done **server-side**

Getting this right is tricky!

Encodings for different *contexts* on the web

- **HTML encoding** `< > & '` replaced by `> < & '`

Complication: encoding of attribute inside HTML tag may be different

- **URL encoding aka %-encoding**

`/ ? = % #` replaced by `%2F %3F %3D %25 %23`

`space` replaced by `%20` or `+`

Try this out with e.g. `https://duckduckgo.com/?q=%2F+%3F%3D`

Complication: encoding for **query segment** different than for initial part

- **JavaScript string literal encoding** `'` or `"` replaced by `\' \"`
- **CSS encoding**
- ...

These encodings can be combined with **validation** or **filtering**

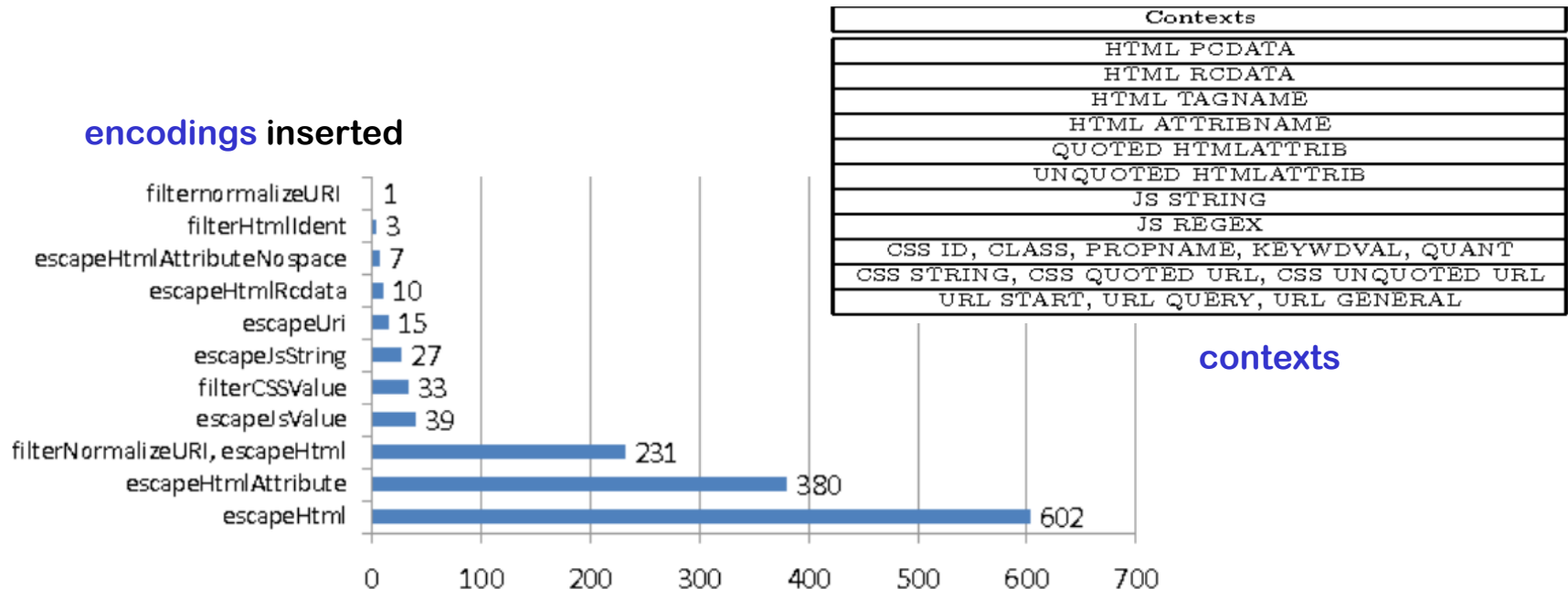
- eg to **remove** HTML tags instead of encode them,
to allow some HTML tags, or
to **reject** certain types of URL

Context-sensitive auto-escaping

Context-sensitive auto-escaping web template engines try to figure out & insert the right encodings.

E.g. **Google Closure Templates**, using contexts & encodings below

Many template engines are not context sensitive!



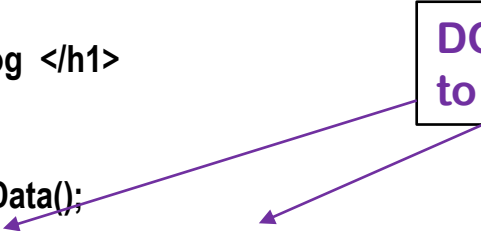
[Samuel, Saxena, and Song, Context-sensitive auto-sanitization in web templating languages using type qualifiers, CCS 2017]

Complication: 'dynamic' webpages via the DOM API

JavaScript inside a web page can dynamically alter that web page using the **DOM API** (or interact with other Web APIs provided by browsers)

```
<html> <body>
  <h1 id=title> ${name}&apos;s Blog </h1>
  ...
  <script> let newName = getSomeData();
            document.getElementById("title").innerHTML = newName + "&apos;s Blog!";
  </script>
</body> </html>
```

DOM API methods & fields to inspect & alter the web page



Spot the XSS, if attacker can inject data via getSomeData()

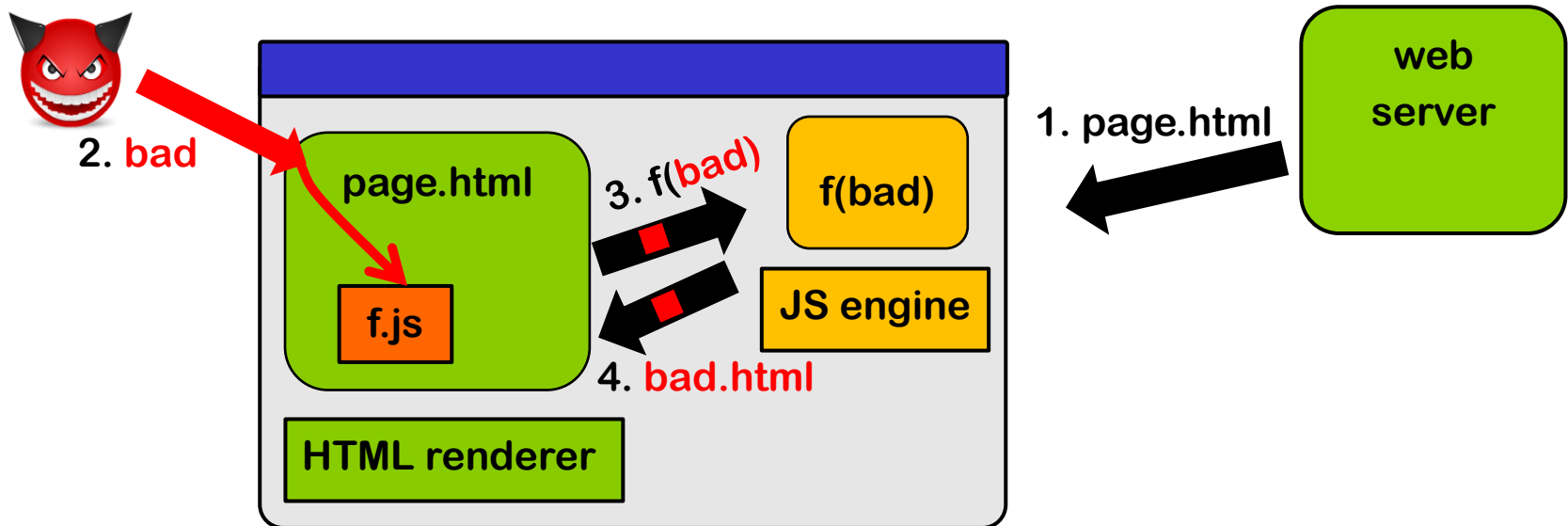
A malicious newName could be **Eve</h1><script someAttackScript();</script> //**

If getSomeData() is untrusted user input, it needs to be **encoded**, by the JS code:

```
document.getElementById("title").innerHTML = htmlEscape(newName) + "&apos;s Blog!"
```


DOM-based XSS attacks

Here JS code in a webpage is fed malicious input (**client-side!**) and uses that input to change the webpage (**client-side!**)



Input can come from i) local user input, ii) parameter in the URL, iii) the server (as in stored XSS), iv) another web server,...

Server cannot validate or encode such inputs! (Except in case (iii)?)
It has to be done by the JS code inside the web page.

Escaping by JavaScript code

Suppose that for a user-supplied **name** we want JS code to create a link, labelled with **name**, that executes JS code `createAlbum('name')` when clicked, i.e.

```
<a onclick="createAlbum('name')">name</a>
```

in an HTML element elem.

Insecure JS code to do this

```
elem.innerHTML = '<a onclick="createAlbum(\' + name + \')"> + name + </a>';
```

Spot the XSS bug!

As malicious **name** attacker can inject `''; someAttackScript(); //`

How should JS code escape name for the two different contexts here?

```
var escapedName = goog.string.htmlEscape(name); // HTML-encoding
```

```
var jsEscapedName = goog.string.escapeString(escapedName); // JS string literal encoding
```

```
elem.innerHTML = '<a onclick="createAlbum(\' + jsEscapedName + \')"> + escapedName + </a>';
```

Spot the XSS bug!

Spot the XSS bug!

```
var escapedName = goog.string.htmlEscape(name); // HTML-encoding
var jsEscapedName = goog.string.escapeString(escapedName); // JS string literal encoding
elem.innerHTML = '<a onclick="createAlbum(\' ' + jsEscapedName + '\')">' + escapedName + '</a>';
```

Attacker can enter malicious name `');attackScript();//`

HTML-escaped this becomes `');attackScript();//`

JS-escaped this remains `');attackScript();//`

Note: JS-escaping does not escape `'` but would escape `'`

So innerHTML becomes

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

The browser HTML-*unescape*s value of onclick attribute before evaluation as JS

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

so `attackScript();` will be executed

[Example from Christoph Kern, Securing the Tangled Web, CACM 2014]

Preventing DOM-based XSS

Writing JavaScript code that properly validates and encodes user input is hard!

Modern web pages use a *LOT* of client side JS code, using large libraries, for dynamic web pages

The DOM API methods take **strings** as arguments, but for these strings it is hard to trace

- where they come from? (are they user input?)
- have they been validated? if so, how exactly?
- have been encoded? and if so, how exactly?

Here we can use the safe builder approach!

API hardening for the DOM API (aka Trusted Types)

Uses the **Safe builder approach** for JavaScript & DOM API

- use TypeScript rather than JavaScript
- use different types instead of just **String**,
e.g. **TrustedHtml**, **TrustedJavaScript**, **TrustedUrl**, **TrustedScriptUrl** ...
- replace string-based DOM API with new typed API where operations take the right 'safe' type as parameter
 - eg `innerHTML` takes **TrustedHtml** instead of a **String**
- Typing guarantees proper escaping & validation 😊
 - This is checked statically
- DOM API must be replaced & all JS code needs to be rewritten ☹️
 - but ... this can be done incrementally, using old & new APIs side by side

[<https://github.com/WICG/trusted-types>]

[Released as a Chrome browser feature in 2019]

<https://developers.google.com/web/updates/2019/02/trusted-types>]

Custom tweaks

The Trusted Types / API hardening approach can be customised/extended to specific application:

For example, Brightspace allows a restricted set of HTML tags in forum postings.

To do this we would introduce

- introduce a custom type, **SafeForumPosting**,
- specify which functions require input of this type
- define custom operations to generate data of this type, using validation and/or encoding. This code should be rigorously reviewed to make sure it is bullet-proof!

Yet another complication: different kind of URLs

Suppose we let users **add a link to jump to their homepage** on another website

```
<html> <body>
  <h1> ${name} &apos;s Blog! </h1>
  ${description}
  ...
  <script> function goHome() { window.location.href = ${homeUrl} ;} </script>
  <button type="button" onclick="goHome()">Click here to go to ${name} 's home page!</button>
  ...
```

Spot the XSS, if we allow users to specify any `${homeUrl}`

Browsers support **pseudo URLs** starting with **javascript:**, e.g. `javascript:alert('Hi!')`.

Assigning such a URL to `location.href` will execute the script!

User-supplied URLs have to be **validated** to check for **javascript:** URLs:

- server-side or -- if it's passed around in JS -- client-side in JS code

The Trusted Types API uses special type **TrustedResourceUrl** for sinks, such as `location.href`, where (pseudo) URLs can trigger execution of scripts

Recap: Why XSS is so tricky to prevent

- Many sources & sinks, with complex data flows between them
- Many different types of data
 - URLs, URL parameters/query string, javascript: pseudo URLs, HTML, attributes in HTML tags, JavaScript, JavaScript strings, CSS, ...

with different trust levels, eg

- HTML with scripts that we trust,
unsafe HTML, possibly with scripts,
safe HTML without scripts,
links that we trust in places where they might trigger scripts,
links that we trust except in places where they might trigger scripts, ...

and different association forms of encoding and validation, eg

- HTML-encoding, JavaScript-literal encoding, URLs validated not to start with javascript:, ...

that can be done server-side or client-side

Conclusions

Languages & Parsing

- Parsing of many languages (formats, representations, ...) is where input problems happen, due to
 1. insecure parsing
 2. incorrect parsing, i.e. parsing differentials
 3. unintended parsing, i.e. injection attacksespecially if languages are complex, poorly defined, and very expressive
- LangSec approach can prevent 1 & 2
- Safe builder approach, which generalises parameterised queries, can prevent 3

Validation vs Sanitisation/Encoding/Escaping

- **Validation** and **sanitisation/encoding/escaping** are two very different operations
- *Output encoding* makes more sense than *input sanitisation*, because encoding/sanitisation depends on **context**
- Ideally, **don't validate but parse**
- Ideally, use **'safe' APIs** that are immune to injection attacks using **types** to enforce proper sanitisation & validation

Anti-pattern: **STRING CONCATENATION**



Standard recipe for security disaster:

1. concatenate several pieces of data, some user input,
2. pass the result to some API

Note: string concatenation is *inverse* of parsing

Anti-pattern: STRINGS

The use of strings in a warning sign

not just `String` but also `char*`, `char[]`, `StringBuilder`, ...

Strings are *useful*, because you use them to represent many things:

eg. username, file name, email address, URL, javascript URL, HTML, ...

This also make strings *dangerous*:

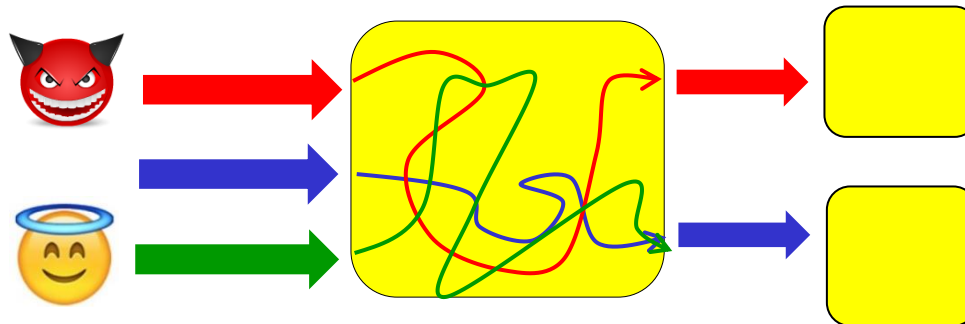
- Strings are unstructured data that still needs to be parsed
- The type does not tell if it has been validated, encoded, or what the intended use – ie. how it should be parsed
- A single string parameter in an API call often hides an expressive & powerful language

Pattern: Use Types

Types can record & ensure various aspects of data

- **origin of data**, and hence the **trust** we have in it
 - special mention: **compile-time constants**
- **language/format it is intended for**
- **validated or not, and how exactly?**
- **encoded or not, and how exactly?**

preventing ambiguity & confusion



To read

- Wang et al., **If It's Not Secure, It Should Not Compile: Preventing DOM-Based XSS in Large-Scale Web Development with API Hardening**, ICSE'21, ACM/IEEE, 2021
- Lectures notes on Secure Input Handling



Getting things wrong: double en/decodings

Chrome used to crash on the URL `http://%%30%30`

- `%30` is the URL-encoding of the character `0`
- So `%%30%30` is the URL-encoding of `%00`
- `%00` is the URL-encoding of null character
- So `%%30%30` is a double-encoded null character

Cause of the crash: code deep inside Chrome performs a second URL-*de*coding (as well-intended ‘service’ to its client code?) and then some other code crashes on the resulting null character.

How could this bug have been detected or prevented?

Having encoded data around makes validation harder!

Double encoding is a common way to get past validation checks.

Note that encoding is the opposite of canonicalisation:
it introduces *different* representations of the *same* data.

Problem: keeping track of which data is encoded / may be decoded can be tricky in larger programs. Typing can help!