Software Security

# Secure input handling

# -

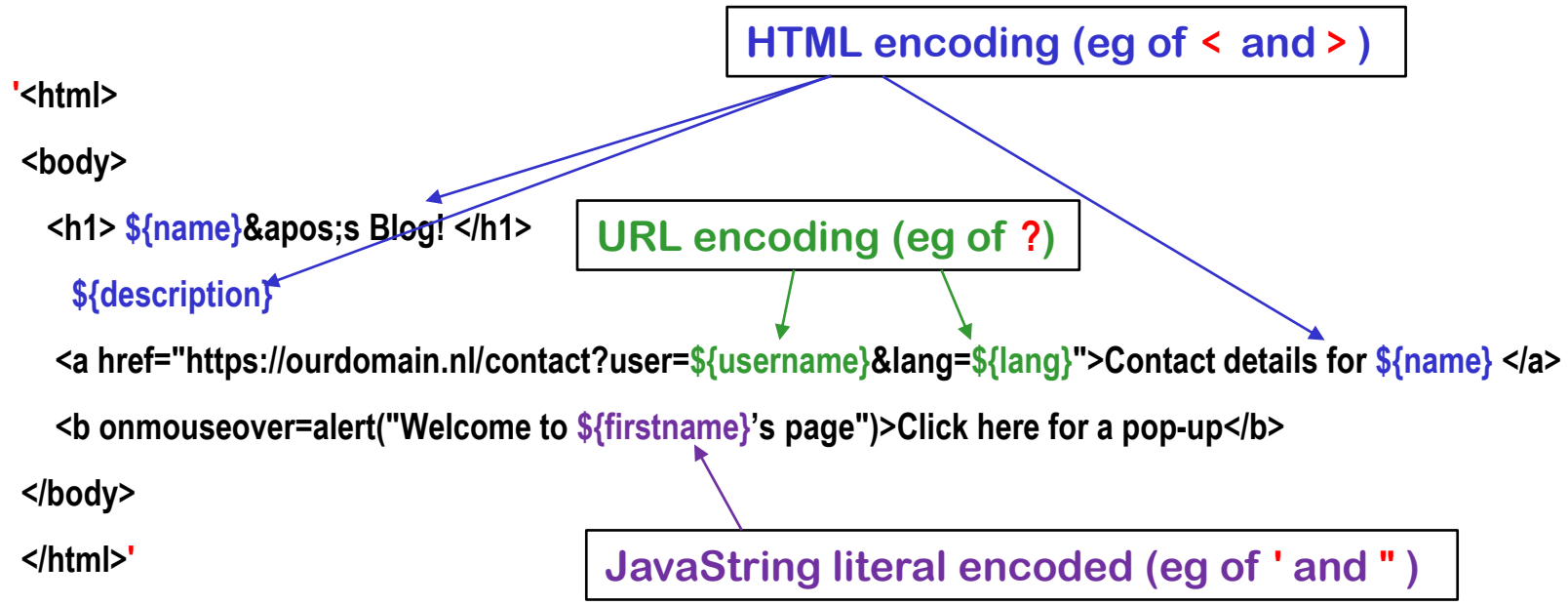
# DOM-based XSS


**Erik Poll**

Digital Security

Radboud University Nijmegen

# Contexts & encoding for the web




**Getting encodings right is tricky and depends on the context**

# Extra complication: the DOM API

JavaScript inside a web page can dynamically alter that web page using the DOM API (or do other interactions with other Web APIs)

```
<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!*

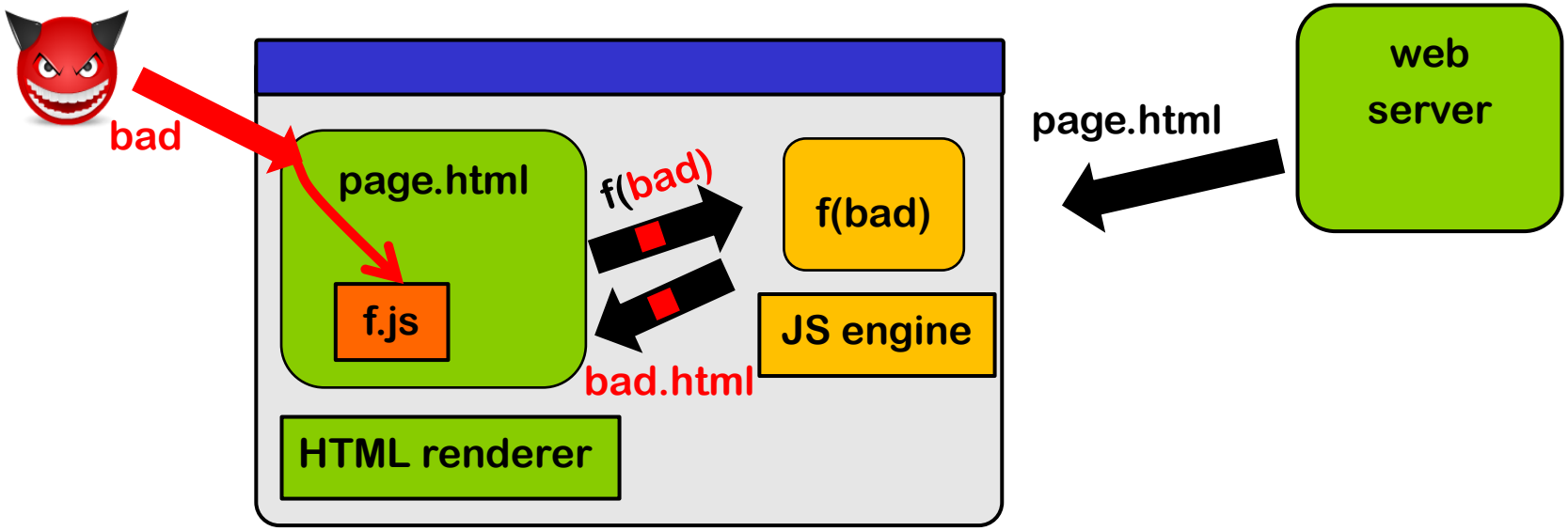
A malicious newName could be Eve</h1><script someAttackScript();</script> //

If newName 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

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




Input can come 1) via local user input, 2) as parameters in the URL, 3) from the server (as in stored XSS), 4) from another web server, …

Server cannot validate or encode such inputs! (Except in case 3?)
It has to be done by JS code inside the web page.

# Escaping inside JavaScript

Suppose JavaScript code modifies an HTML element elem to show a user-supplied name that executes JS code createAlbum('name') when clicked, i.e.

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

Insecure JS code to do this

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

*Spot the XSS bug!*

A malicious name to insert `' '; someAttackScript(); //`

*How to 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>';
```

Attack: enter malicious name `');attackScript();//`

HTML-escaped this becomes `&#39;);attackScript();//`

JS-escaped this remains `&#39;);attackScript();//`

So innerHTML becomes

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

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

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

so attackScript(); will be executed

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

# Preventing DOM-based XSS

**Moral of the story: 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, to provide fancy webpages**

**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)

**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

1. introduce a custom type, SafeForumPosting
2. specify which functions require input of this type
3. define custom operations to generate data of this type, with built-in 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, javascript: pseudo URLs,
  (snippets of) HTML and 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 even 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, JavaString-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 the input problems happen, due to
  - insecure parsing
  - incorrect parsing, i.e. parsing differentials
  - unintended parsing, i.e. injection attacks

  especially if languages are complex, poorly defined, and very expressive
- LangSec approach can prevent buggy parsing which can be insecure parsing or incorrect parsing
- Safe builder approach, which generalises parameterised queries, can prevent injection attacks

# Lack of input validation?

**Beware of people talking about 'lack of input validation'**

- **Do they really mean *rejecting* invalid inputs or do they actually mean *encoding/escaping/sanitising* them?**
  - **If so, *output* encoding makes more sense than *input* encoding, because it depends on context**
  - **Ideally, don't validate but parse**
  - **Ideally, use 'safe' APIs that are immune to injection and/or us types to enforce proper encoding & validation**

# Pattern: Use Types!

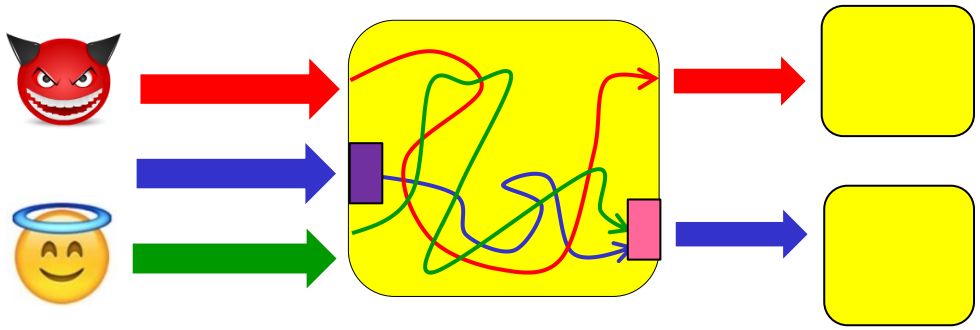
Types can record & ensure different aspects of data

1. language/format
2. origin of data, and hence the trust we have in it
   - special mention: compile-time constants

This can track & make explicit if data

- validated or not, and how exactly?
- encoded or not, and how exactly?

Overall aim: preventing ambiguity & confusion



# 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, HTML, …

This also make strings *dangerous:*

1. Strings are unstructured data that still needs to be parsed
2. The same string may be handled & interpreted in many – possibly unexpected – ways
3. Strings may or may not be validated or encoded, …
4. A single string parameter in an API call often hides an expressive & powerful language

# 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