

Software Security
Secure input handling

Erik Poll

Digital Security

Radboud University Nijmegen

The messy business of preventing XSS

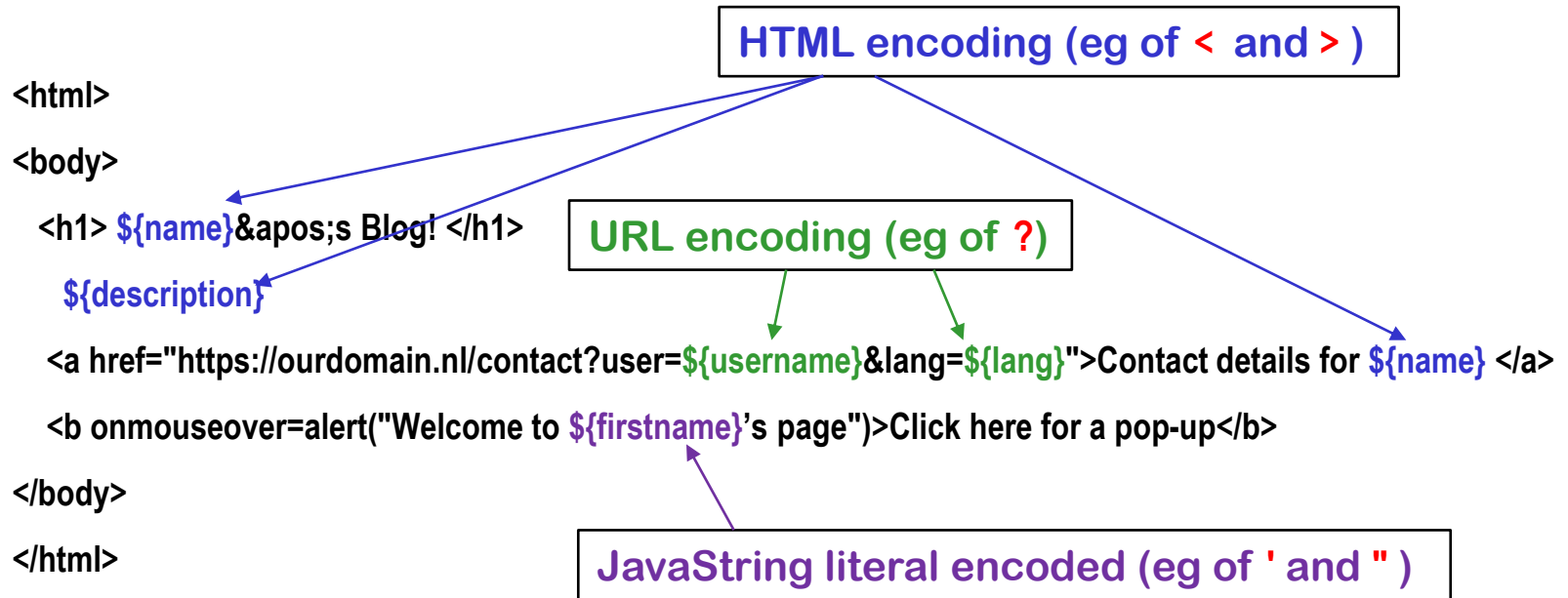
'Normal' XSS: reflected or stored

The web server sends victim a webpage with malicious script controlled by the attacker



- The malicious script is created **server-side**
- Web server can prevent this by careful **validation & encoding**
- **Context-sensitive auto-encoding** template engine can help

Encoding for the web - server-side



There are many **contexts!**

The web server has to apply the right encoding & validation for right context.

Some of the encodings for the web

- **HTML encoding**

`< > & " '` replaced by `> < & " '`

- **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; for example, the initial `//` of a URL should obviously *not* be encoded

- **JavaScript string literal encoding**

`'` replaced by `\'`

Eg `"this is a JS string with a \' in the middle"`

Complication: JavaScript allows `'`, `"` and ``` for strings

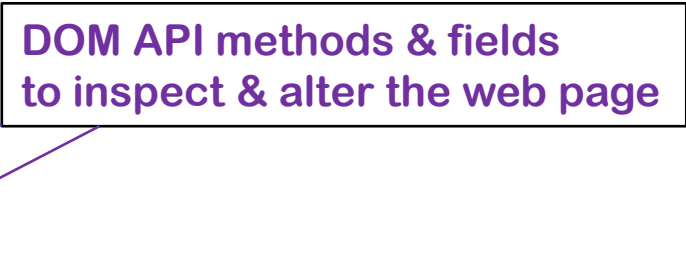
- **CSS encoding**

- ...

Complication: the DOM API

JavaScript inside a web page can dynamically alter it using the **DOM API**

```
<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>
```



Spot the XSS!

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

If newName is untrusted user input, it needs to be **encoded** by JavaScript code:

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

This is called **DOM-based XSS**, as scripts is injected via **DOM API**

Tricky details for script injection via `.innerHTML`

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

XSS via `newName` is not quite as simple as suggested on previous slide:

HTML spec says `<script>` in assignments to `.innerHTML` should not be executed

Read <https://html.spec.whatwg.org/#the-script-element:dom-innerhtml>
aka page 629 (out of 1357!) of <https://html.spec.whatwg.org/print.pdf>

So we have to obscure the script a bit

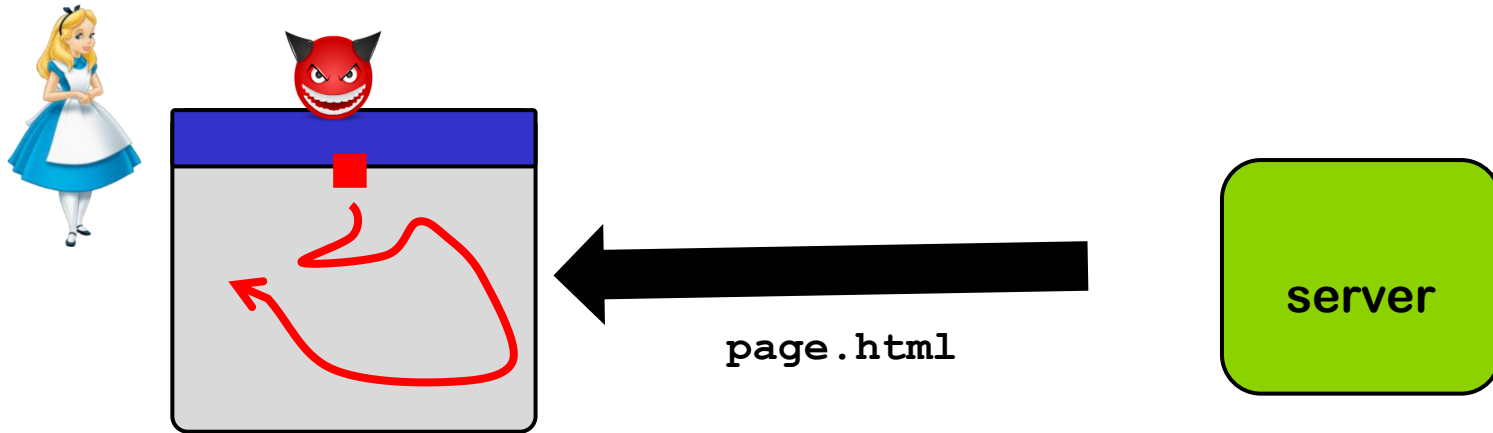
A malicious `newName` would be

```
Eve</h1> 
```

DOM-based XSS

1. The web page sent to victim does not contain scripts – yet ...
It may contain some attacker-controlled content, or not
2. Execution of JavaScript inside the browser, using attacker controlled input, creates scripts

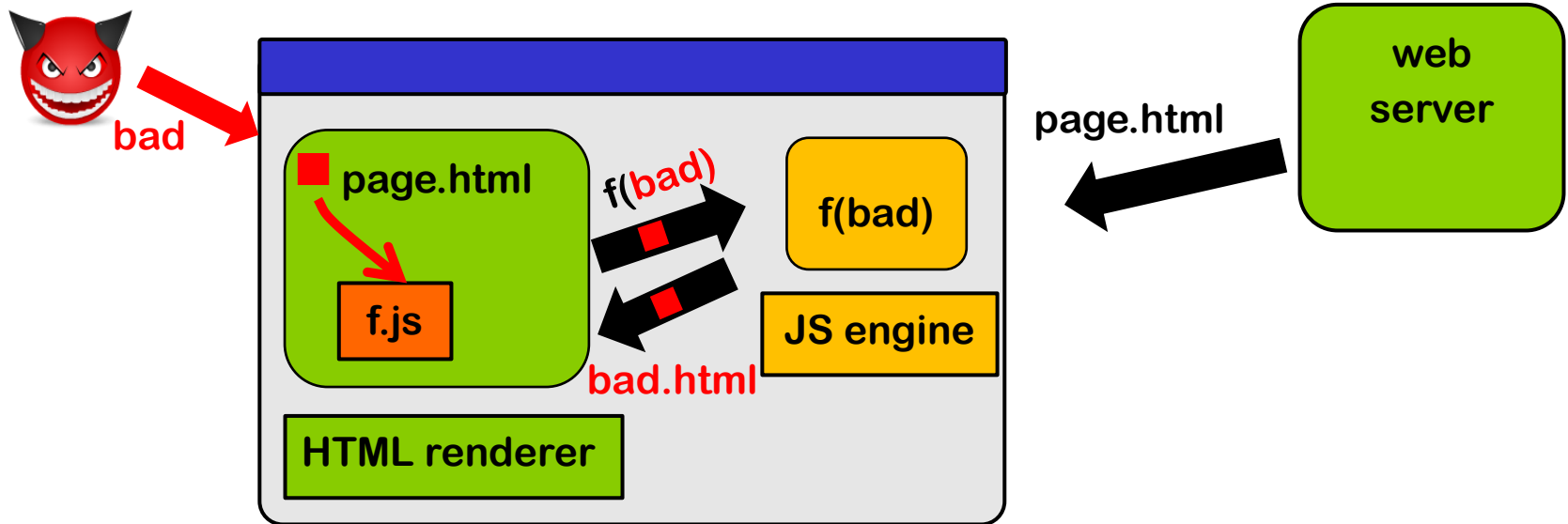
This input may come via the server, or not



The malicious payload is constructed **client-side**

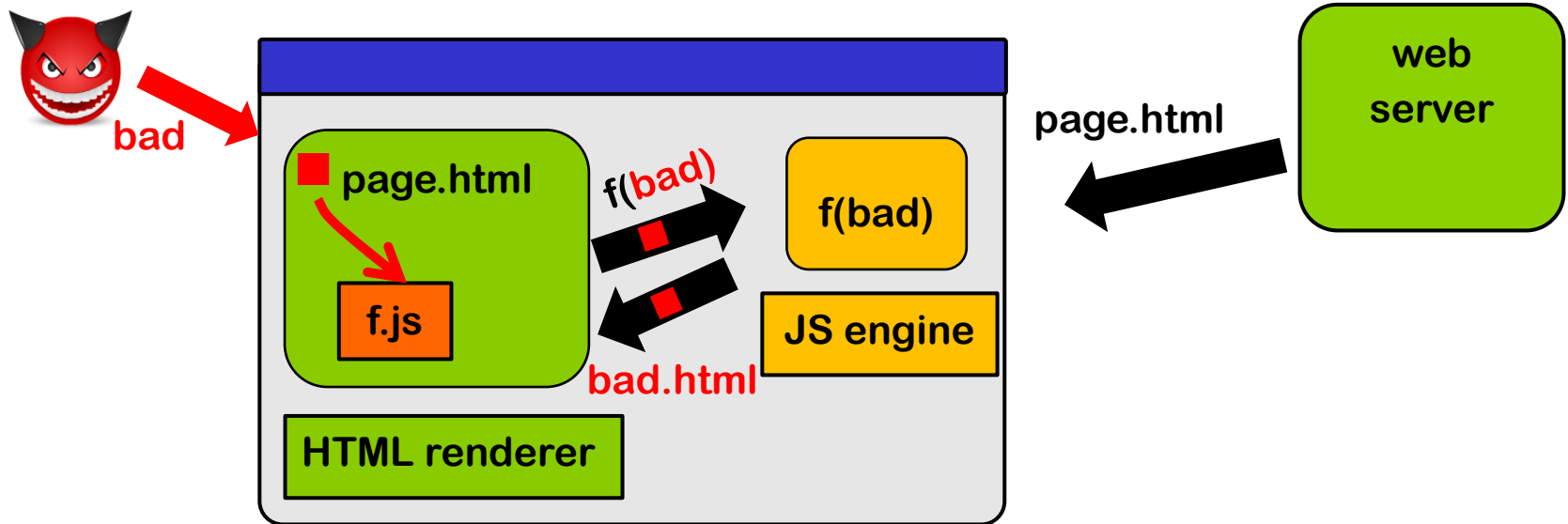
- This exploits a flaw in JavaScript code inside the web page
- This code could be in a third-party JS library, or in the interaction of 1st & 3rd party JS code

DOM-based XSS



1. The web page sent to victim does not contain scripts – yet ...
but is does contain scripts
2. Attacker input ends up in the web page, client-side
This input may come via server, or not
3. Execution of JavaScript inside the browser, using attacker controlled input, creates scripts that are injected into the page

DOM-based XSS



- 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.

Encoding inside JavaScript is tricky

JS code to create an HTML element `elem` containing a link, labelled with a user-supplied `name`, that executes JS code `createAlbum('name')` when clicked,

i.e. `name`

JS code to do this

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

Spot the XSS bug!

As malicious `name` insert `' ; someAttackScript(); //`

How to encode 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!

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

So innerHTML becomes

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

The browser HTML-*unescape*s 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]

Yet another complication: different kind of URLs

Suppose we let users add a link to their homepage

```
<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 my home page! </button>
  ...
```

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

Provide as `${homeUrl}` a **pseudo-URL** of the form `javascript:alert('Hi!')`

Browsers support such pseudo URLs;

assigning such a URL to `location.href` will execute the script inside it!

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

- **server-side** or, if they are passed around in JS, **client-side** in JS code

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

Why XSS is so tricky to prevent

- **Many sources & sinks, with complex data flows between them**
 - **Many different types of data**
 - **URLs, javascript: pseudo URLs, HTML, JavaScript, JavaScript strings, CSS, ...**
- with different trust levels, eg**
- **HTML with scripts that we trust,**
unsafe HTML, possibly with scripts,
safe HTML without scripts,
URLs we trust even in places where they might trigger scripts,
URLs 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: so that they can not trigger script execution,...**
- that can be done server-side or client-side**
- **Modern web pages use *LOTS* of client-side JS libraries!**

Preventing DOM-based XSS

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

The DOM API methods take **STRINGS** as arguments.

For these strings it is hard to trace

- **Where does they come from?**
 - is it an untrusted, user-supplied string or a trusted, constant string supplied by the server ?
- **Have they been validated? If so, how?**
 - is some string we use as URL validated not to start with javascript: ?
- **Have they been encoded? And if so, how exactly?**
 - has it been HTML-encoded, JS-encodes, first HTML-encodde and then JS encoded, or the other way around?

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 to a 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,
 - 1) from **compile-time constants**
 - 2) from **validated & encoded strings**

SafeForumPosting validateAndEncode (String s)

This code should be rigorously reviewed to make sure it is bullet-proof!

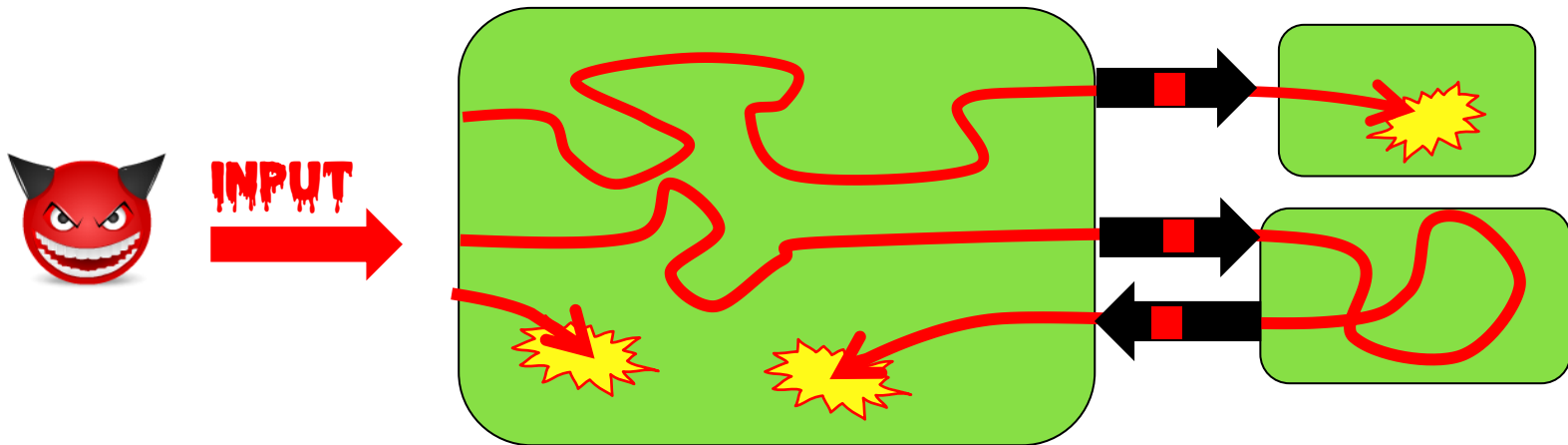
To read

- 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

Conclusions – of last 3 weeks

INPUT problems - due to parsing

Most security problems are input problems, where input is passed around to trigger **bugs** or **features**



This always involves **parsing** of some **format/language**

e.g. PDF, SQL, Word, path/file names, URLs, TCP/IP packets, ...

Parsing problems: types & root causes

Security problems can arise due to

1. **insecure parsing**,
e.g. buffer overflows parsing JPEG
2. **incorrect parsing**, i.e. **parsing differentials**
e.g. when parsing URLs, X509 certificates, ...
3. **unintended parsing**, i.e. **injection attacks**
e.g. SQL injection, XSS, Word Macros, SMB relays, ..

Especially if

- a) there are **many** languages,
- b) these are **complex**,
- c) **poorly defined**, and/or
- d) **very expressive**

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**

Preventing input problems

- **LangSec approach** to prevent **insecure or incorrect parsing**
 - have a clear spec & generate parser
- **Type-based safe builder approach** to prevent **injection attacks**

using **types** to guarantee the systematic **validation** and **output encoding** of untrusted data fed into dangerous APIs

 - generalises the idea of parameterised queries
 - aka **API hardening** or **safe APIs**

Anti-pattern: STRINGS



The use of **STRING CONCATENATION**, or even just **STRINGS**, is 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*:

- Strings are **unstructured data** that still needs to be parsed
- The same string may be **handled & interpreted in many – possibly unexpected – ways**
- **Strings may or may not be validated or encoded, ...**
- A single string parameter in an API call often hides **an expressive & powerful language**
- Note that **string concatenation is the inverse of parsing**



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

