Software Security

Secure INPUT handling

Erik Poll

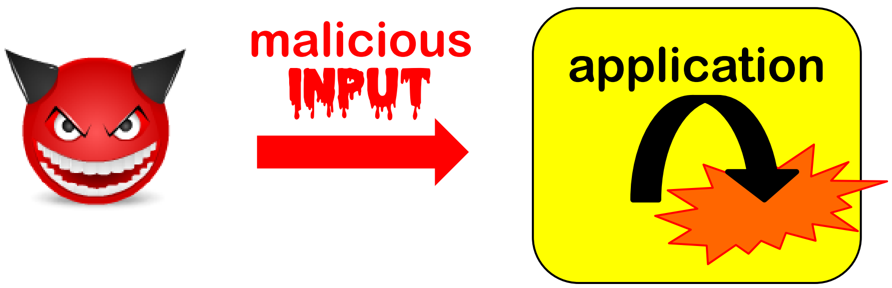
Digital Security

Radboud University Nijmegen

Recap: two types of input problems

1. Buggy, insecure parsing

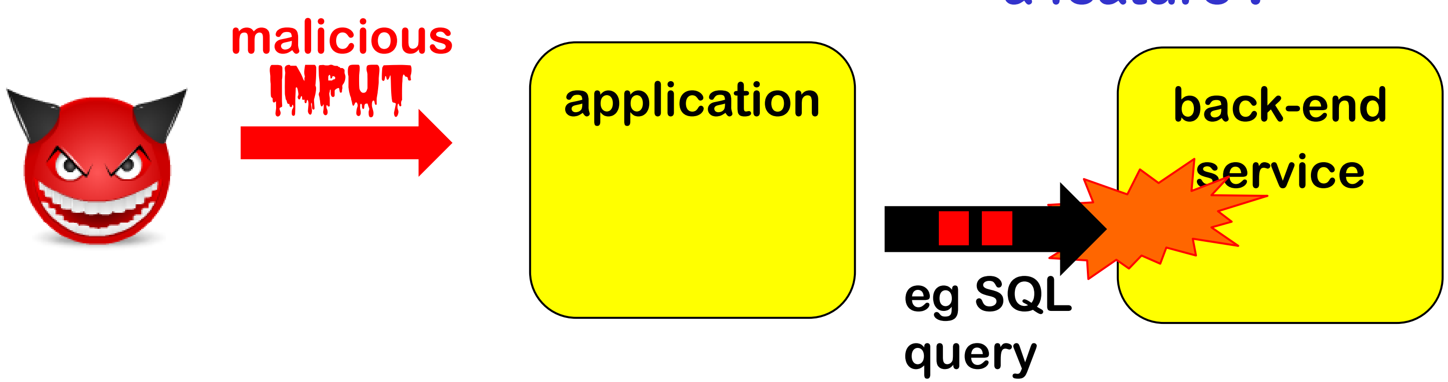
a bug !



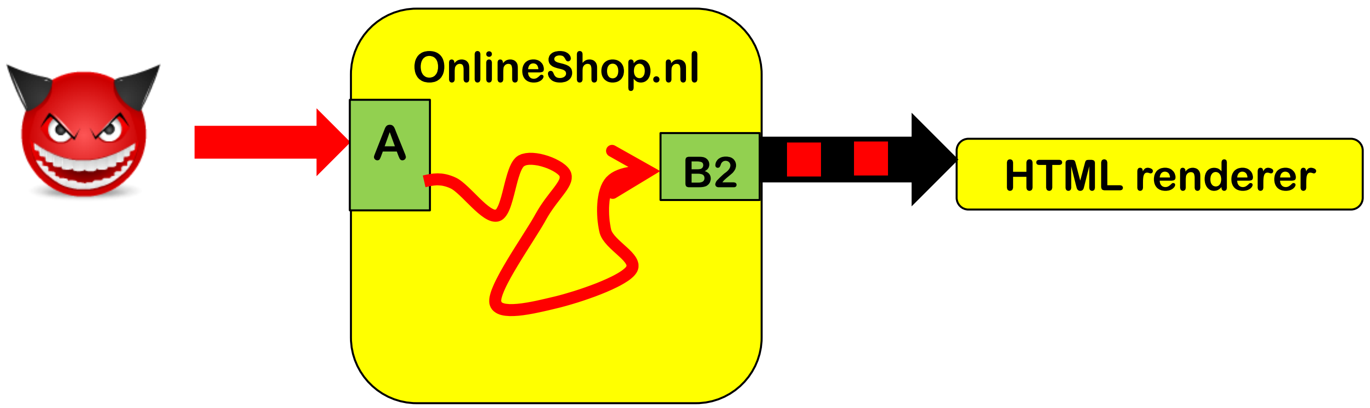
eg buffer overflow
in PDF viewer

2. Injection attacks:
correct but unintended parsing

(abuse of)
a feature !



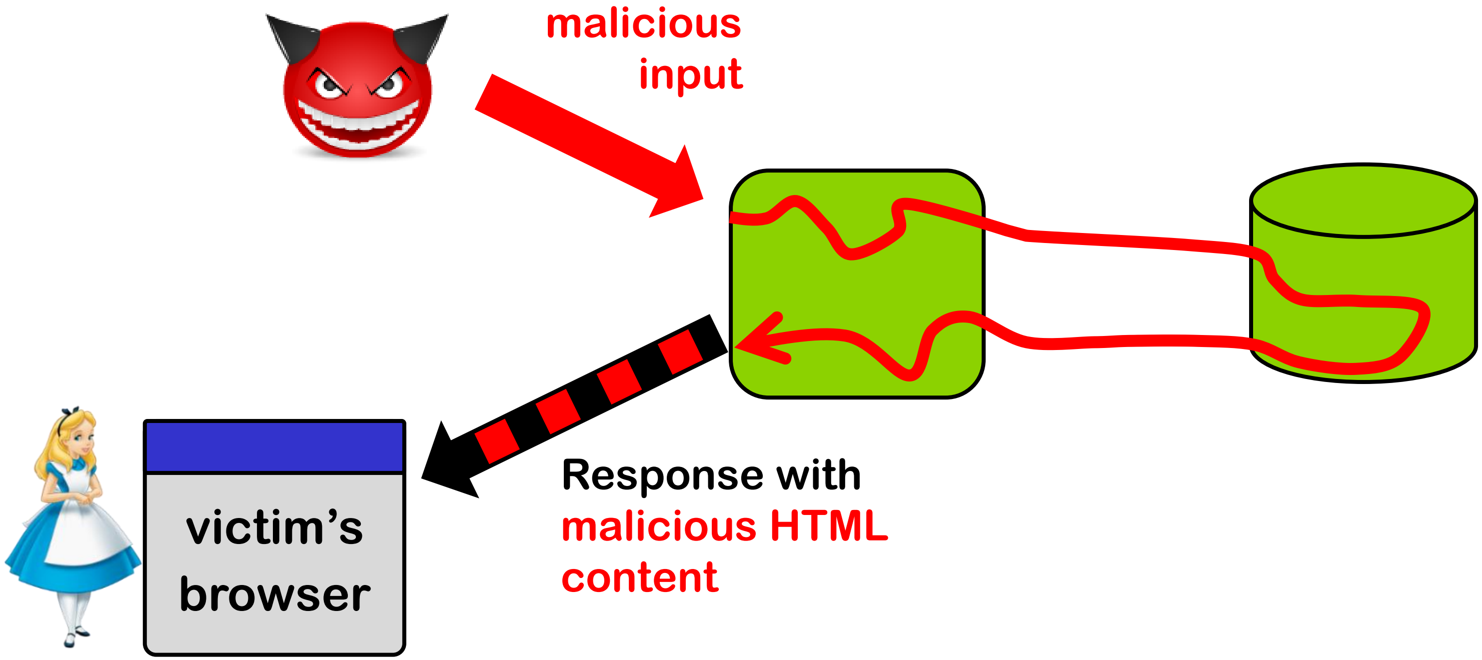
Output encoding



Output encoding needs be tailored to the context:

for HTML renderer `< > & script`

XSS attack


malicious input
Response with malicious HTML content
victim's browser

Encoding for the web

HTML encoding (eg of < and >)

URL encoding (eg of ?)

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

JavaString literal encoded (eg of ' and ")

Some of the encodings for the web

- **HTML encoding**

  < > & " '   replaced by   > lt; & &quot &#39

  Complication: encoding of attributes inside HTML tags 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, eg for / aka %2F

- **JavaScript string literal encoding**

  '   replaced by   \'

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

  Complication: JavaScript allows both ' and " for strings

- **CSS encoding**
- ...

Context-sensitive auto-escaping

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

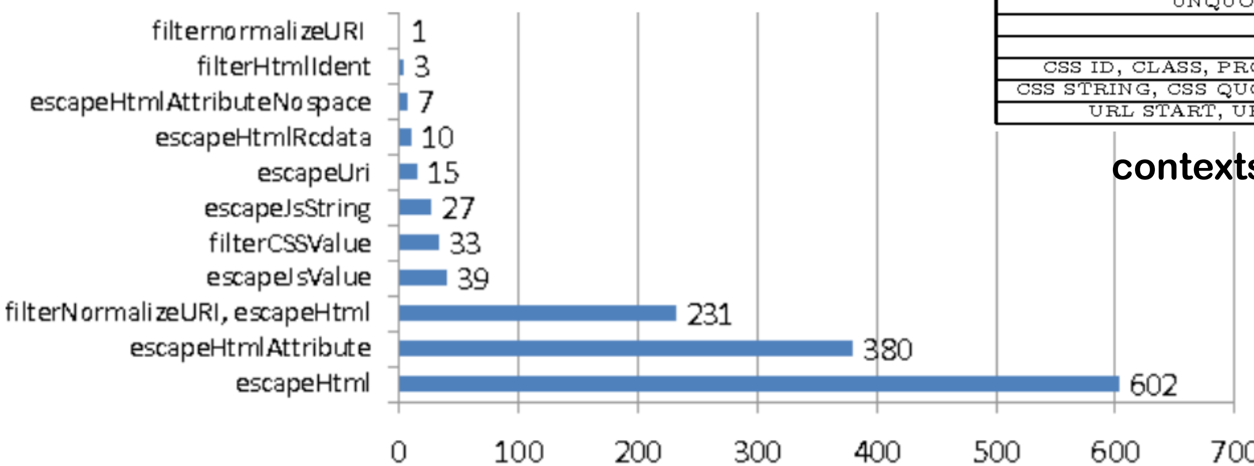
E.g. Google Closure Templates, using context & encodings below

Many template engines are not context sensitive!

Contexts
HTML PCDATA
HTML RCDATA
HTML TAGNAME
HTML ATTRIBNAME
QUOTED HTMLATTRIB
UNQUOTED HTMLATTRIB
JS STRING
JS REGEX
CSS ID, CLASS, PROPNAME, KEYWDVAL, QUANT
CSS STRING, CSS QUOTED URL, CSS UNQUOTED URL
URL START, URL QUERY, URL GENERAL

contexts (type qualifiers)

encodings inserted



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

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!

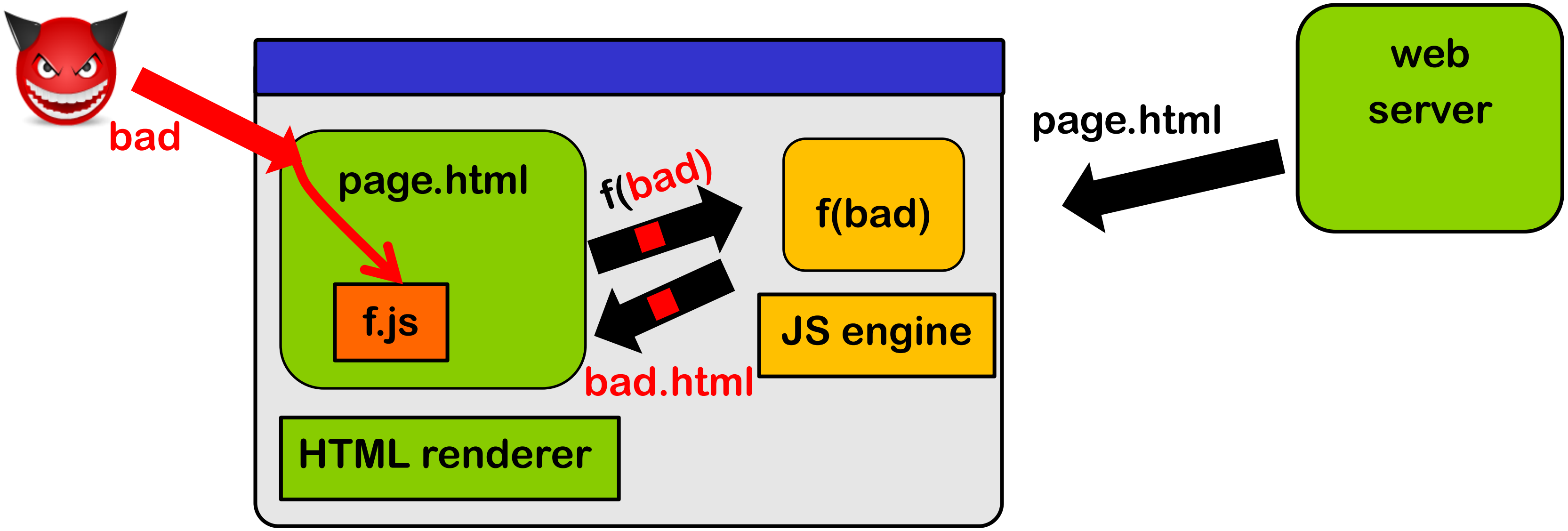
A malicious newName could be Eve</h1><script someAttackScript();</script> //

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!)



Malicious input can enter as 1) local user input, 2) URL parameters, 3) from the web server, 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 we want JS code to create/change an HTML element elem into a link, labelled with 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!

As malicious name 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 `');attackScript();//`

JS-escaped this remains `');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

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 exacty?

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

- 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, of if its 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

Conclusions

Languages & Parsing

- Parsing of many languages (formats, representations, ...) is where 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 - insecure or incorrect - incorrect parsing
- Safe builder approach, which generalises parameterised queries, can prevent injection attacks
  - Injection possibilities become type errors

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 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, HTML, …

This also make strings *dangerous:*

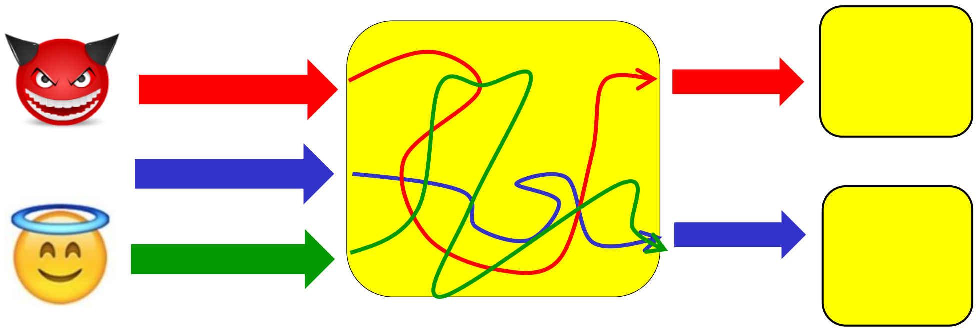
1. **Strings are unstructured data that still needs to be parsed**
2. **The same string may be handled & interpreted in many – possibly unexpected – ways**
3. **Strings may or may not be validated or encoded, ...**
4. **A single string parameter in an API call often hides an expressive & powerful language**

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!