Software Security

# Secure input handling

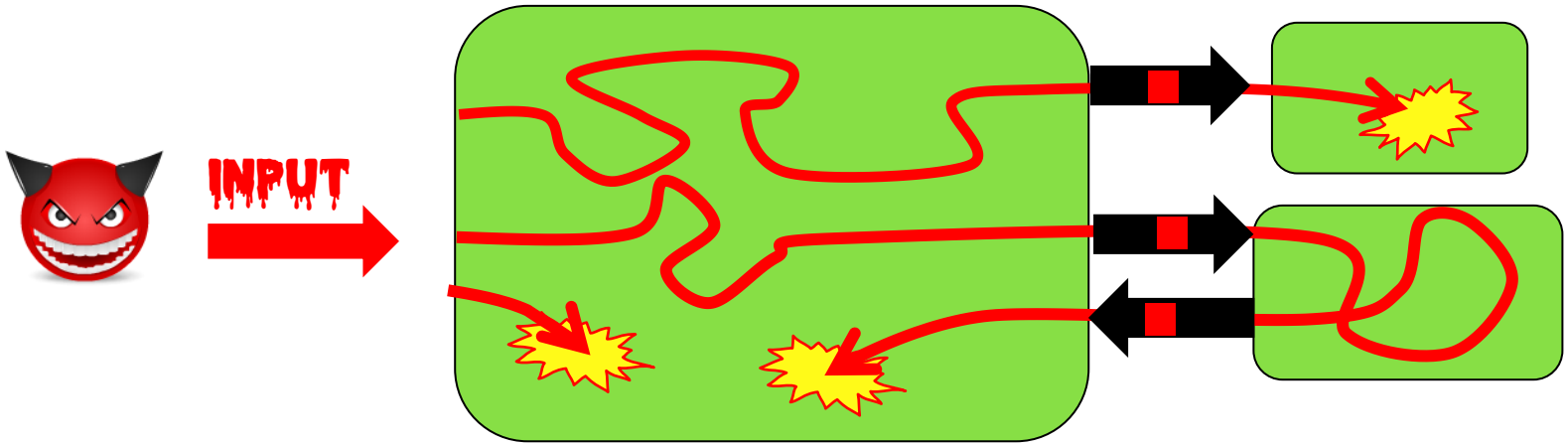
Erik Poll

Digital Security

Radboud University Nijmegen

# INPUT problems, which involve 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. TCP/IP packets, PDF, Word, JPEG, SQL, path/file names, URLs, ...

# Secure input handling

Last week: preventing input problems with

- Validation
- Sanitisation (aka encoding)
- Canonicalisation

This week: more 'structural/foundational' solutions to rule out

I. Tackling buggy parsing with LangSec

II. How (not) to tackle unintended parsing - ie injection flaws

   a) Input vs output sanitisation

   b) Taint Tracking

   c) Safe builders

      Case study: XSS

# How encoding can complicate matters

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

***How could this bug have been detected or prevented?***

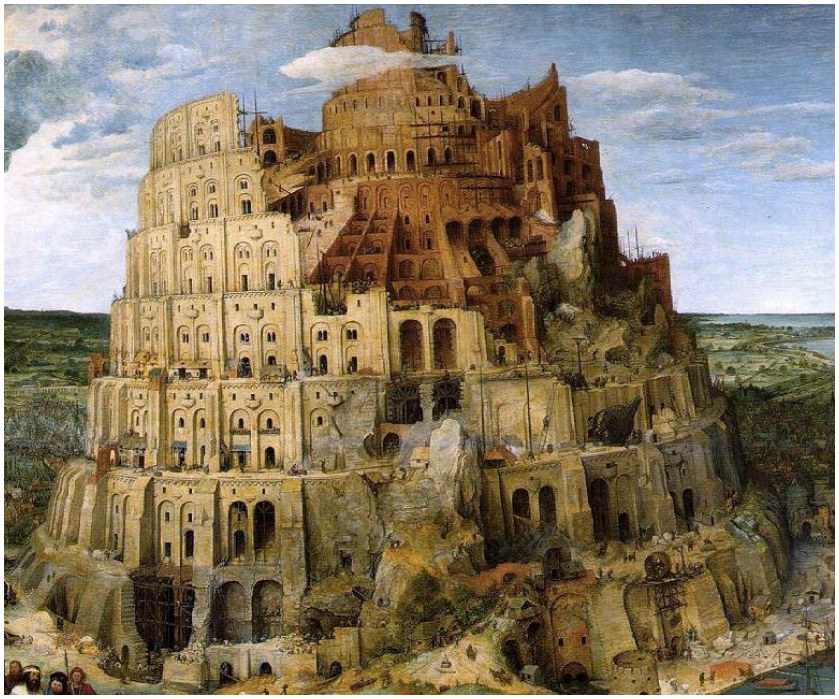
**Having encoded data around makes validation harder!**

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.

# I. Tackling buggy parsing
# -
# using the LangSec approach



# Buggy parsing

Here by buggy parsing we mean

1. insecure parsing

   Eg. buffer overflow in Office, PDF viewer, network stack, graphics library, ..

2. incorrect parsing resulting in parser differentials,
   i.e. two libraries parsing the same URL in different ways

# *Can we use input validation?*

- **Suppose we have a buggy PDF viewer with memory corruption that allows RCE.**

  ***Can we use input validation as protection?***

- **Yes & no:**
  - **we could validate a PDF file before feeding it to our PDF viewer,**
  - **but… for that we need a correct & secure PDF parser, so we are back to the original problem**
  - **Still, for legacy applications it may be an improvement**

# LangSec (Language-Theoretic Security)

- Interesting look at root causes of large class of input handling bugs, namely buggy parsing
- Useful suggestions for dos and don'ts


Sergey Bratus &
Meredith Patterson
presenting LangSec at CCC 2012

'The science of insecurity'

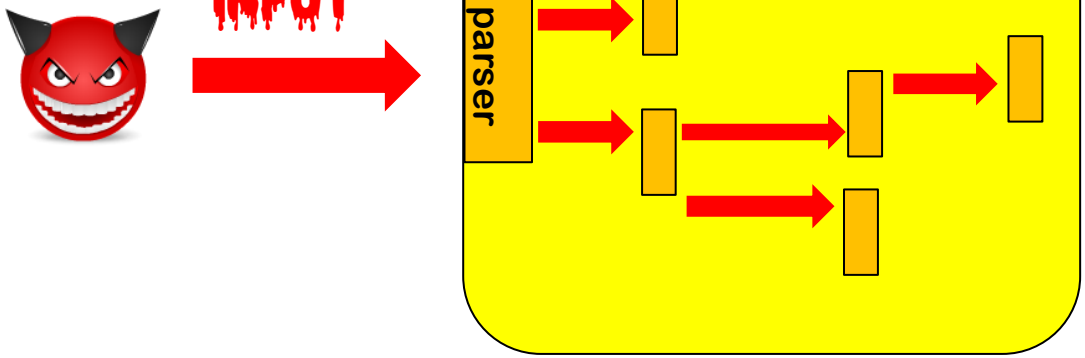
- The 'Lang' in 'LangSec' refers to *input* languages, not *programming* languages.

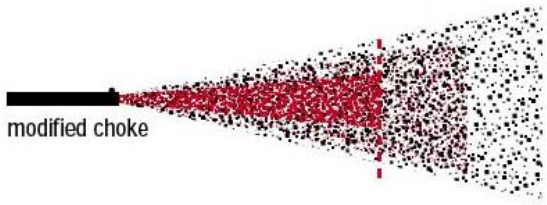
# Root causes / anti-patterns

- Complex input language or format
- Sloppy definitions of this input language or format

- Hand-written parser code
- Mixing input recognition & processing in shotgun parser

# Anti-pattern: shotgun parser




Fragments of input penetrate deeply, and any code that touches these fragments may contain exploitable input bugs.

Code incrementally parses & interprets input, in a piecemeal fashion, chopping it up for further parsing elsewhere

Fragments passed around as unparsed byte arrays or strings

# LangSec concepts

- **Shotgun parser:** shattershot approach to parsing data in bits and pieces, mixing recognition (i.e. the actual parsing) & processing

- **Weird machine:** a buggy parser provides a strange execution platform that can be 'programmed' with malformed input
  - This weird machine may even be Turing-complete (recall ROP programming with gadgets)
  - Cool example: executing code on a x86 processor just using page faults, without ever executing CPU instructions

    [Bangert, Bratus, Shapiro, and Smith, The Page-Fault Weird Machine: Lessons in Instruction-less Computation, USENIX WOOT 2014]

# LangSec principles to prevent buggy parsing

**No more hand-coded shotgun parsers, but**

1. ***precisely defined*** **input languages**

   **ideally with regular expression or context-free grammar (eg EBNF)**

2. ***generated*** **parser code**

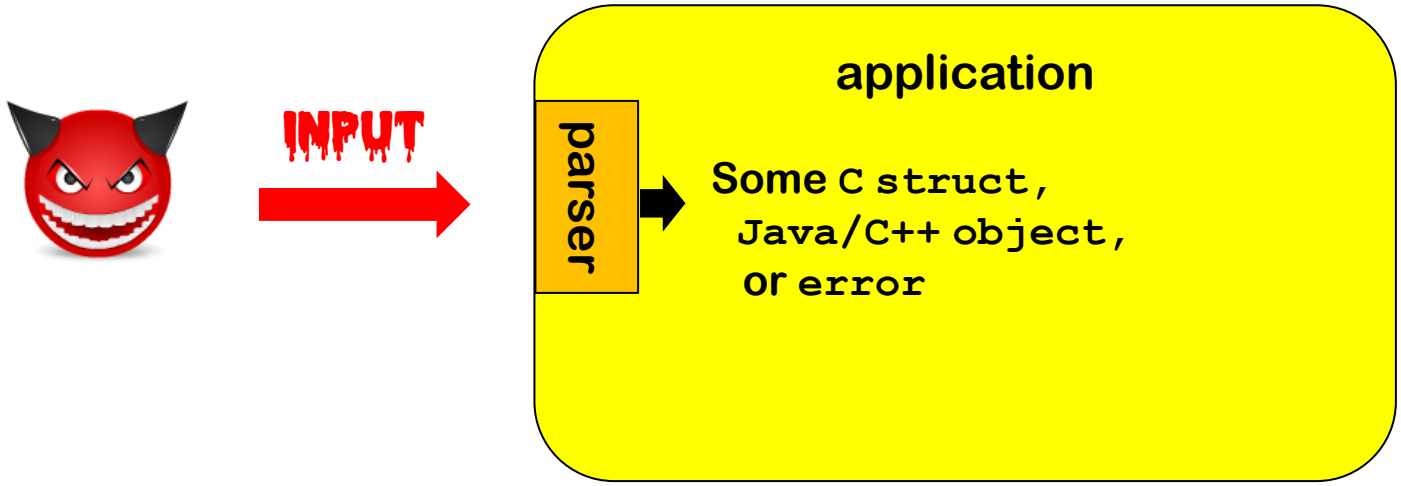
3. ***complete parsing before processing***

4. ***keep the input language simple & clear***

   **So that bugs are less likely**

   **So that you give minimal processing power to attackers**

# Preventing buggy parsing - the LangSec way




## LangSec approach:

- Clear & ideally formally defined input specification
- Generated parser code
- Complete parsing before processing

  rest of the program only handles well-formed data structures produced by parser

# LangSec in slogans

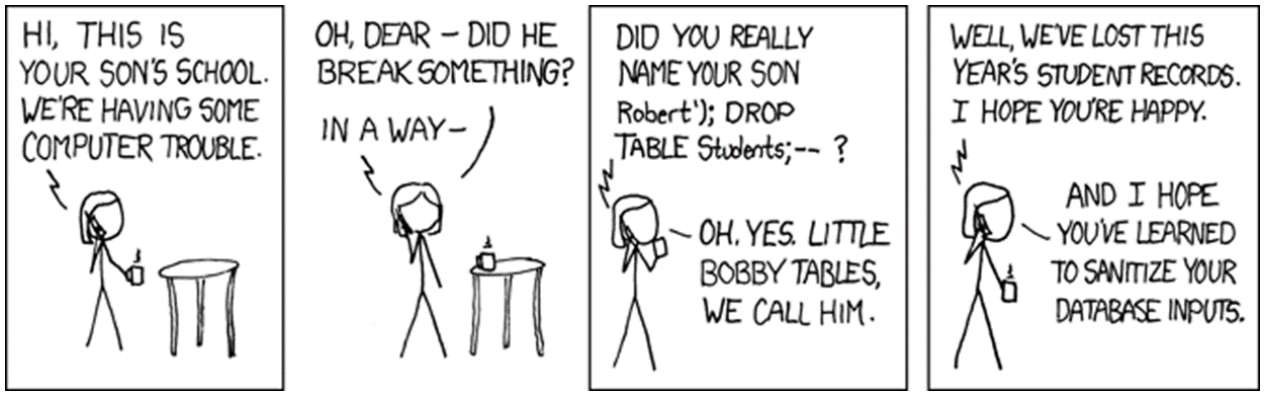

STOP
WEIRD
MACHINES
CONTEXT
FREE
OR
REGULAR


**Minimise the resources & computing power that input handling gives to attackers**


**All parsers should be equivalent.**

**And parsers should be the exact inverse of the pretty printers aka unparsers**

# II. How (not) to prevent injection attacks



# *How* & *where* to prevent injection attacks?


Suppose we are worried about SQL injection via a website

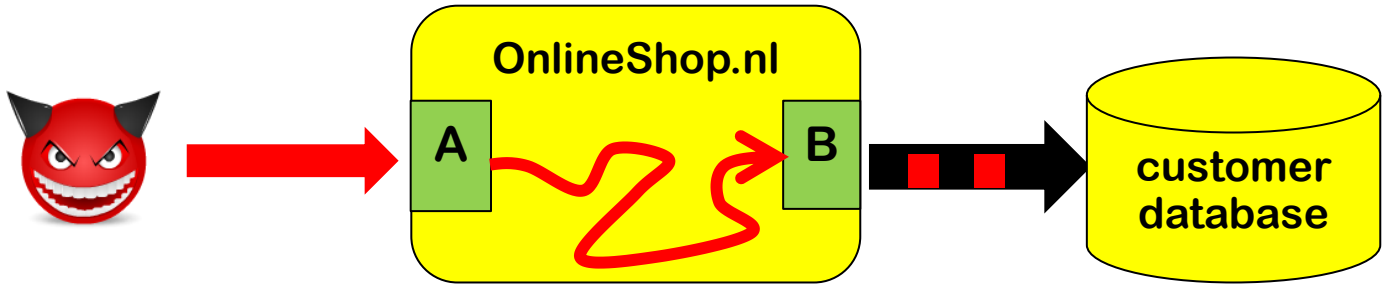
- Should we validate, sanitise, or both to prevent SLQi?
- if so, where? At point A or B?

We assume we know a perfect allow-list or deny-list of dangerous characters for SQL injection.

We ignore canonicalisation of name & address.

We ignore validation to make sure that eg. the address exists.

# Input *validation*?




Input validation, i.e. rejecting weird characters at point A

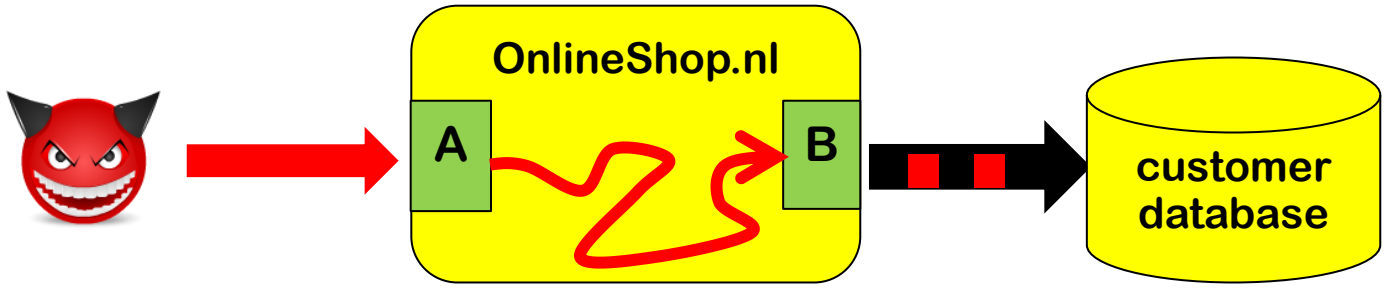
*Pros?*

- Eliminates problem at the source root, so application only has to deal with 'clean' data

*Cons?*

- We may reject legitimate inputs, eg `'s-Hertogenbosch`

# Input *sanitisation?*




Input *sanitisation*, e.g. escaping weird characters at point A

Eg replacing `'` with `\'`

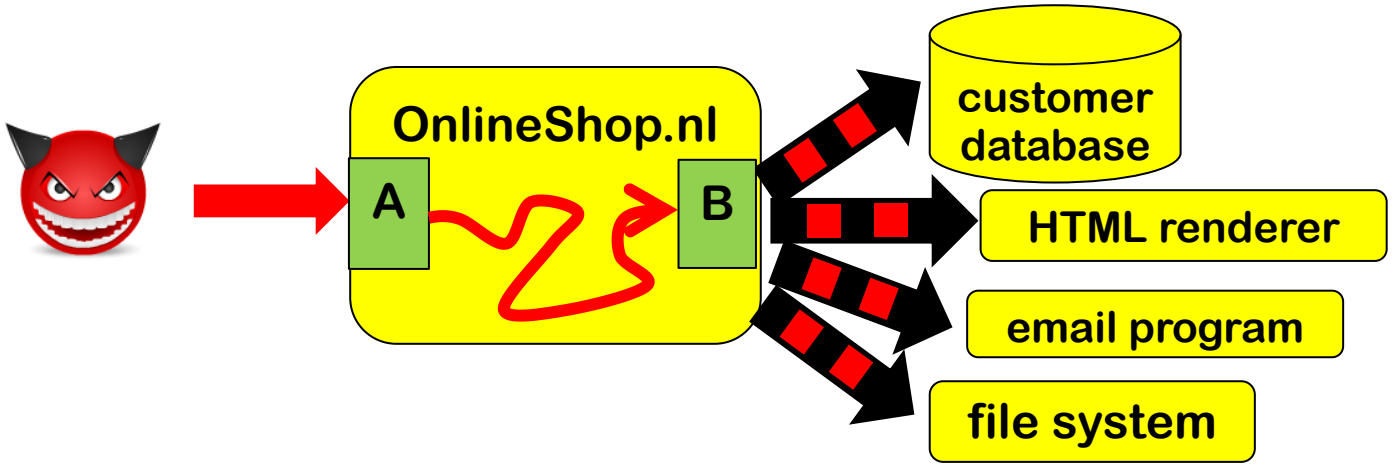
*Pros?*

- Eliminates problem at the source root, so application only has to deal with 'harmless' data
- We no longer reject legitimate input

*Cons?*

- We have some data in escaped form, `\'s-Hertogenbosch` and may need to un-escape it later
- Also, what if there are more back-end than just SQL dataset?

# Multiple backends/APIs introduce multiple contexts

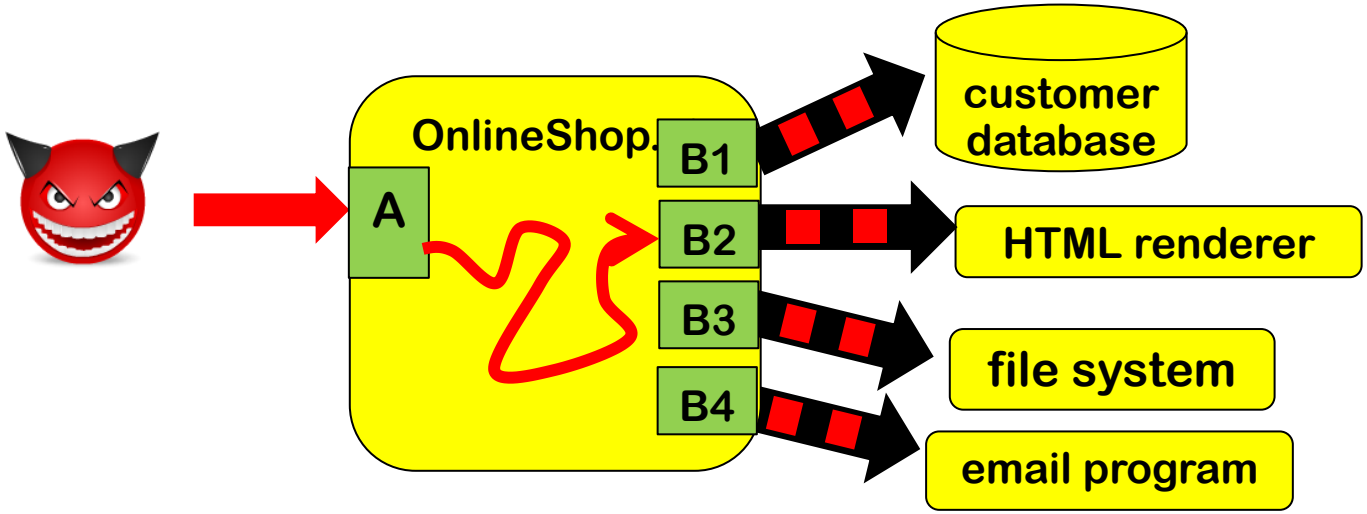



***Different*** escaping needed to prevent SQLi, XSS, path traversal, OS command injection, …

Eg SQL database may be attacked with username `Bobby; DROP TABLE`
but file system with username `../../etc/passwd`
and email program with username `john@ru.nl; & rm -fr /`

For most systems, it's a fallacy to think that ***one*** input sanitisation routine can solve ***all*** injection problems

# *Output* sanitisation! aka output encoding




If we sanitise outputs instead of inputs then sanitisation can be tailored to the context:

| | |
|---|---|
| for SQL database | `; ' " DROP TABLE` |
| for HTML renderer | `< > & script` |
| for file system | `. .. / \ ~` |
| for OS command | `& \| \|\| < >` |

# Output encoding to prevent injection attacks

We can prevent injection attacks by careful output encoding
- in the right place, using the right encoding function.

However, this is easy to get wrong…

More structural approaches to prevent or spot mistakes:

a) Prepared statements aka Parameterised queries

   Easy to get right – as it gets rid of the problem.

   But… only works in simple settings

b) Tainting

   Using DAST or SAST tool to spot or add missing encodings

c) Safe Builders

   Using type system to prevent missing or wrong encodings

# a) Prepared Statements

# Dynamic SQL vs Prepared statements

Interface with SQL database can use

- **Dynamic SQL**:
  one string, which includes user input, is provided as SQL query

```
"SELECT * FROM Account WHERE Username = " + $username
                          + "AND Password = " + $password
```

- **Prepared statements** aka **parameterised queries**:
  a string with placeholders is provided as query,
  and user inputs are provide as separate parameters

```
"SELECT * FROM Account WHERE Username = ? AND Password = ?“
$username
$password
```

# Dynamic SQL & prepared statements in Java

Code vulnerable to SQLi using dynamic SQL

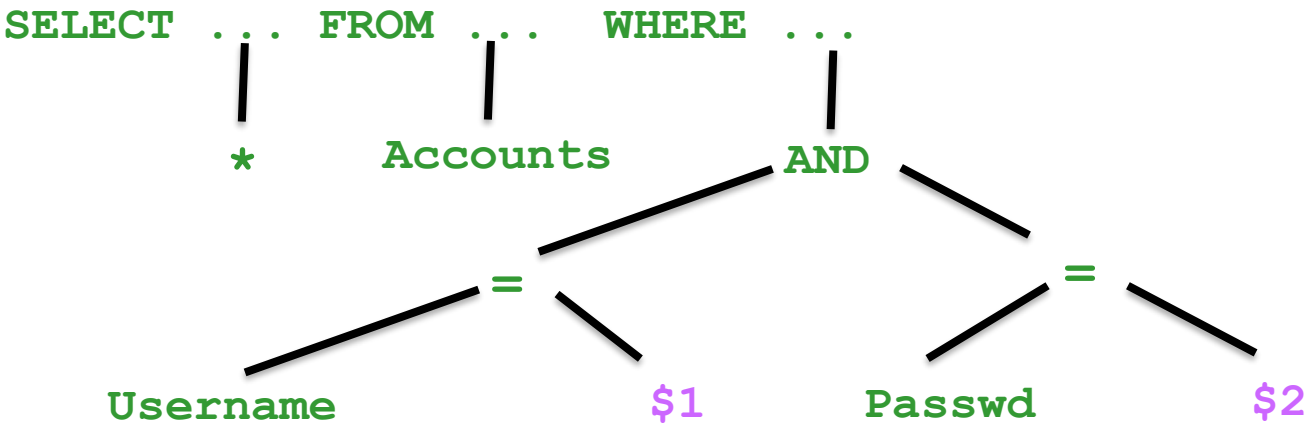
```
String updateString =
  "SELECT * FROM Account WHERE Username"
   + username + "AND Password =" + password;
stmt.executeUpdate(updateString);
```

Code *not* vulnerable to SQLi using prepared statements

```
PreparedStatement login = con.preparedStatement("SELECT
* FROM Account
    WHERE Username = ? AND Password = ?" );
login.setString(1, username);
login.setString(2, password);
login.executeUpdate();
```

bind variable

# The idea behind prepared statemens
## (aka parameterised queries)




- Prepared Statements: the query is parsed *first* and then parameters are substituted later
- Dynamic SQL: parameters are substituted first and then the result is parsed & processed

Key insight: we do not parse the parameters as SQL,
so the substitution becomes less dangerous

# Limitation of this approach, more generally

as general technique to prevent injection attacks

- Requires custom solution for each injection-prone API method
  - Eg for safe LDAP queries, safe XPath queries,....

- Only works for simple situations that
  1. involve just one encoding function
  2. involve only simple substitution patterns

  This means we cannot use it to combat XSS (more on that later)

  Also, it may not be able to express some highly configurable fancy SQL queries

# Prepared Statements not quite fool-proof

Prepared statements are easy to use, but not quite fool-proof

```
PreparedStatement login = con.preparedStatement
    ("SELECT * FROM Account WHERE Username"
      + username + "AND Password =" + password);
login.executeUpdate();
```

# b) Tainting

# Tainting aka Taint analysis

Core idea is to use data flow analysis:

- we track & trace user inputs – aka tainted data
- If tainted data ends up in a dangerous API, we give a warning
- Like SAL annotations SA_Pre[Tainted=True] in PREfast, but inferred automatically

Such an analysis needs to know

- all sources & sinks
- all operations that combine data and propagate taint
  - eg concatenation of two strings is tainted if one of them is
- all operations that sanitise data and remove taint
  - eg SQLencoding removes taint (as far as SQLi is concerned)

Taint analysis can be done dynamically (DAST) or statically (SAST)

# Dynamic & static taint analysis

- Perl scripting language first introduced a taint mode in 1989
  - external input are marked & tracked
  - perl execution engine aborts when tainted data is fed to dangerous functions

  It looks like Perl 6 will remove taint mode
- Windows/Microsoft Office does taint tracking of documents using the Mark of the Web to then block / warn users about macros in tainted document

  Rules have been tightened in March 2022; maybe macros attacks will become a thing of the past?
- Most SAST tools (incl. Fortify, discussed in SIO lecture) use static taint analysis to provide warnings about inputs reaching dangerous sinks (without being validated/encoded).

# *Tainting limitations?*

- Multiple sanitisation operations, for different types of data/different sinks (eg SQL vs HTML), complicate matters

  Accurate analysis requires different kinds of taint

- There may be *many* sources, *many* sinks and *many* operations that remove or propagate taint, or *possibly* propagate taint
  - Missing one is easy, resulting in false negatives or positives.
  - Too much data may get tainted, resulting in unworkable number of false positives.

- Static taint analysis of large programs becomes *complex*.

  False positives or false negatives may be unavoidable.

  Doing intra-procedural analysis (i.e. per method/function) instead of inter-procedural analysis (i.e. whole program) may keep things feasible, typically at the expense of precision

# c) Safe builders

# Safe Builder approach

- Effectively the opposite approach to tainting:

  instead of tracking tainted / dangerous data,
  we track untainted / safe data.

- Key idea: we use type system of the programming language to distinguish
  1. 'trusted' data that does not to be encoded
  2. 'untrusted' data that needs to be encoded
  3. data encoded *for a specific context*
     with a different type for each context

  One special addition to conventional type systems:
  distinguishing compile-time constants (esp. string literals)

Used by Google's Trusted Types in Chrome to combat DOM-based XSS.

# Safe builder for SQL injection

Suppose we have an unsafe API method

```
void executeDynamicSQLQuery (String s)
```

We define a new 'wrapper' String type `SQLquery` and a function that executes such a wrapped string

```
void safeExecuteSQLQuery (SafeSQLquery s){
    executeDynamicSQLCommand(the string in s );
}
```

We now define functions to create `SafeSQLquery`

1. any compiled-time constant can be turned into a `SQLquery`

   ```
   SafeSQLquery create (@CompiletimeConstant String s)
   ```

2. we can append a string to an `SafeSQLquery` using a function

   ```
   SafeSQLquery appendSQL (SafeSQLquery q, String s)
   ```

   which will apply the right encoding to `s`

Type system guarantees that user inputs in queries are properly escaped.

We disallow use of the old unsafe `executeDynamicSQLQuery` .

# Safe builders for several contexts

If we use string-like data in several contexts, each with their own encoding, we can introduce  a different String-like typesa for each, e.g.

```
SafeSQLquery, SafeHTML,  SafeOSCommand, SafeFilename
```

with association constructors or factory methods for each, e.g.

```
SafeHTML create (@CompiletimeConstant String s)
SafeHTML concatHTML (SafeHTML h1, SafeHTML h2)
SafeHTML appendHTML (SafeHTML h, String s)
```

`appendHTML(h,s)` and `appendSQL(h,s)` would use different encodings for the parameter `s`

We could introduce unsafe loopholes that we evaluate by hand

```
SafeHTML unsafeCreate (String s)
```

# Positive vs negative security models

The choice between positive vs negative security models comes back in several places

- Tainting = data is 'safe' unless tainted,

  Safe builders = data is 'unsafe' unless type says otherwise

- allow lists vs deny lists

- security requirements vs attack scenario/threat

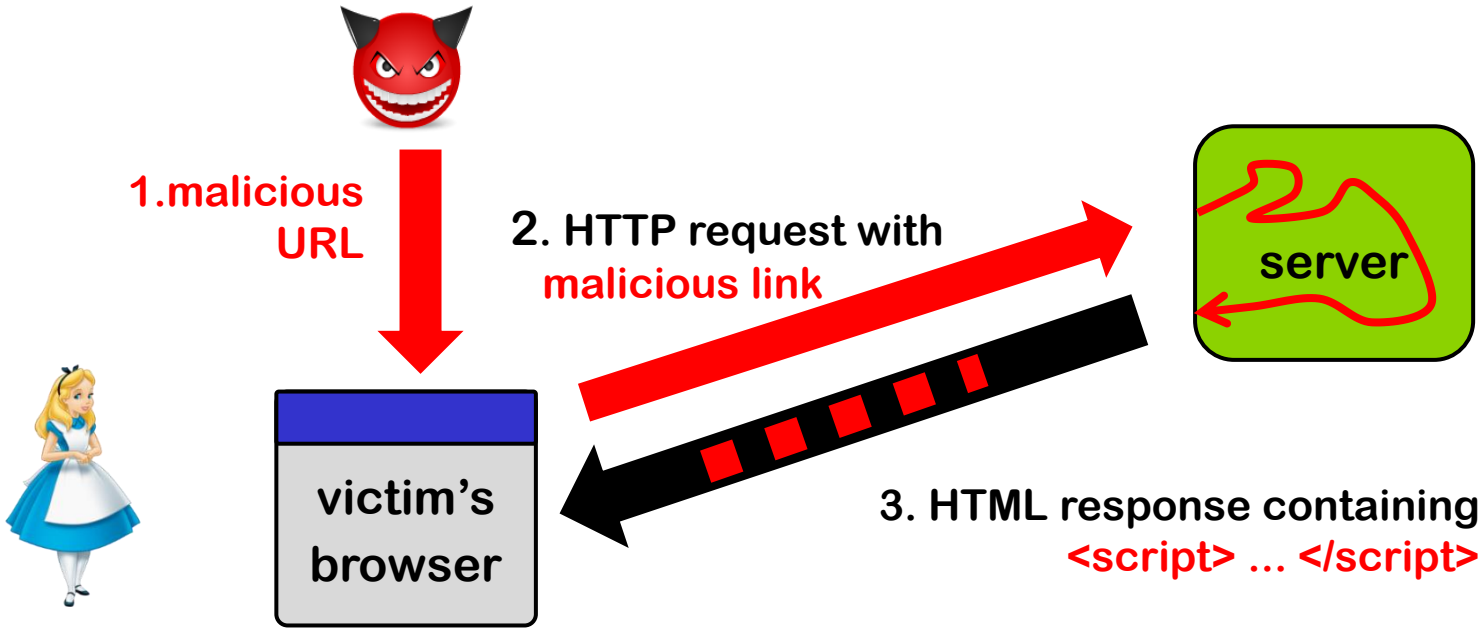
# The messy business of preventing XSS

# Reflected XSS attack

Attacker crafts malicious URL containing JavaScript

`https://google.com/search?q=<script>...</script>`

and tempts victim to click on this link

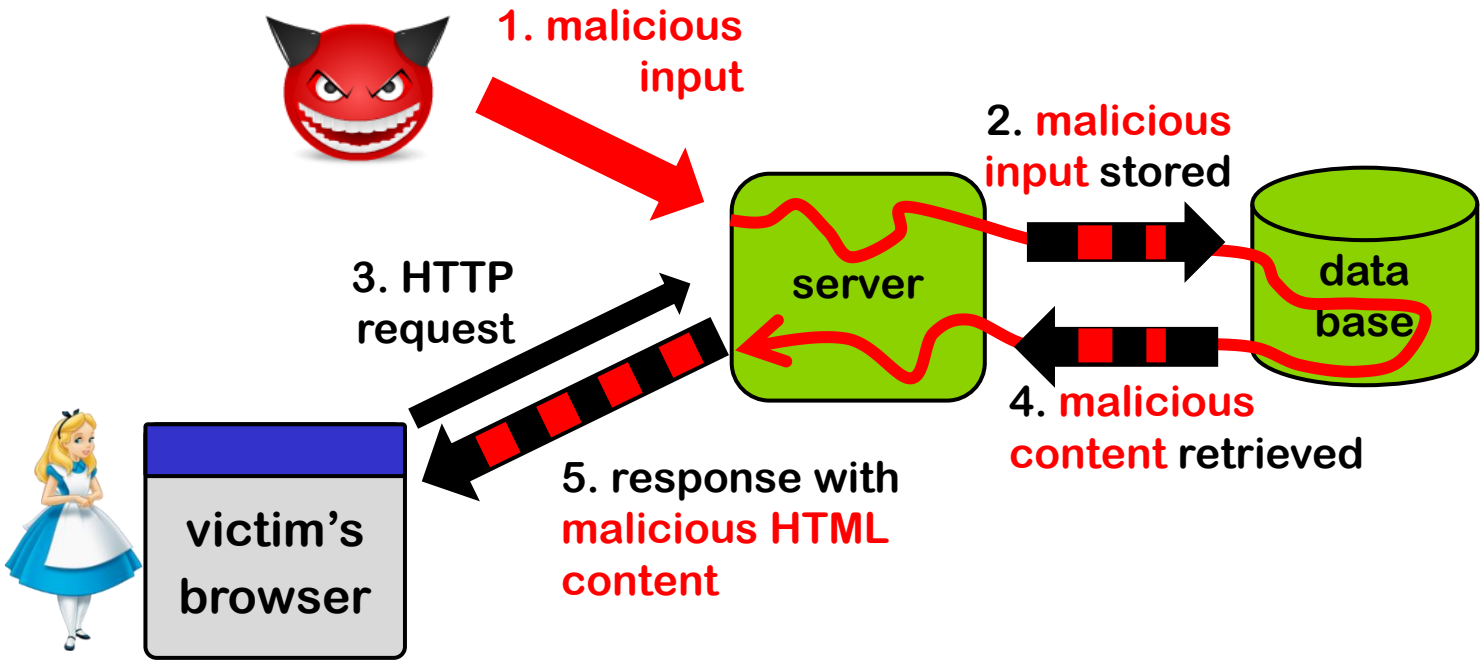



*Could careful web server prevent this?*

Yes, by validating & rejecting and/or encoding content in query!

# Stored XSS attack

Attacker injects HTML into a web site, eg forum posting in Brightspace, which is stored and echoed back *later* when victim visit the same site




*Could careful web server prevent this?*

Yes, by rejecting and/or encoding content when it is stored or retrieved

# Encoding HTML content - server-side

Many sites use web templating framework to generate web pages.

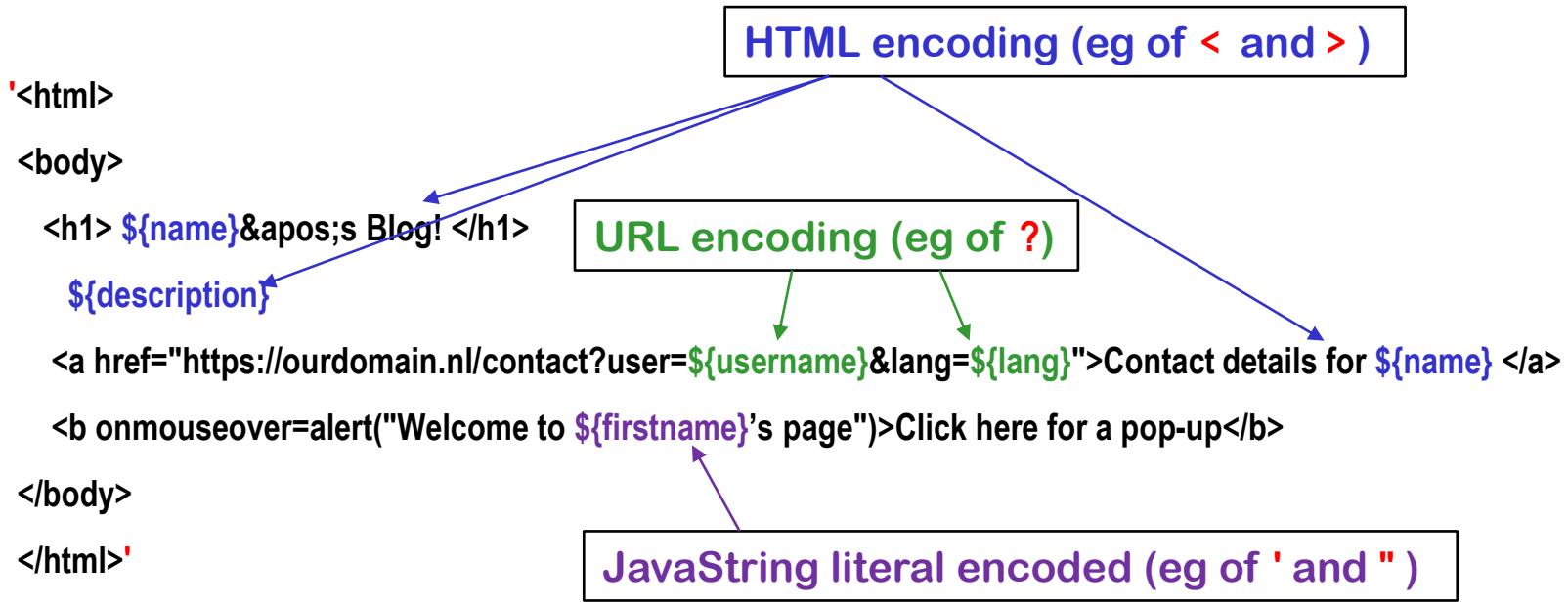
Below a web template for a web page with parameters written as ${...}

```
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 – properly encoded – are filled by web server / templating engine.

*How should the parameters be encoded here?*

# Encoding for the web - server-side




**NB all these encodings can be done server-side**

***Getting this right is tricky!***

# Some of the encodings for the web

- HTML encoding

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

  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, 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

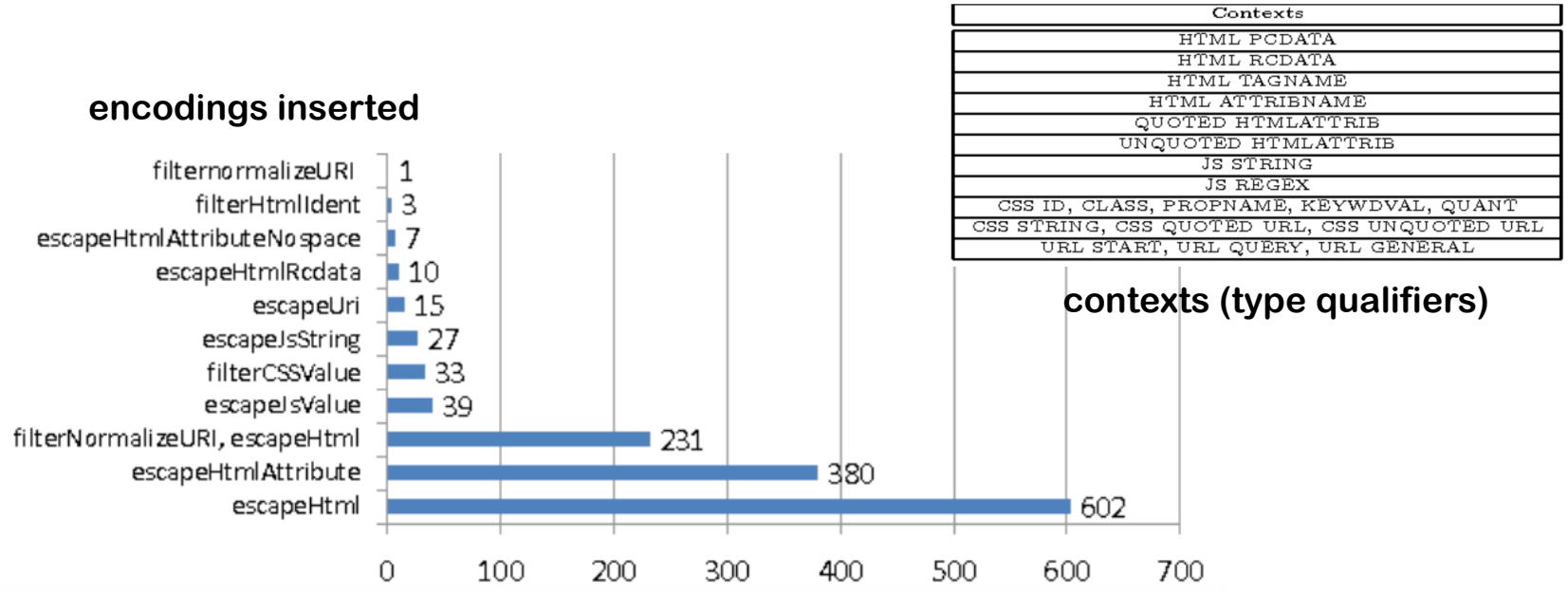
- CSS encoding
- ...

# 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, and hence insecure!**




**[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!*

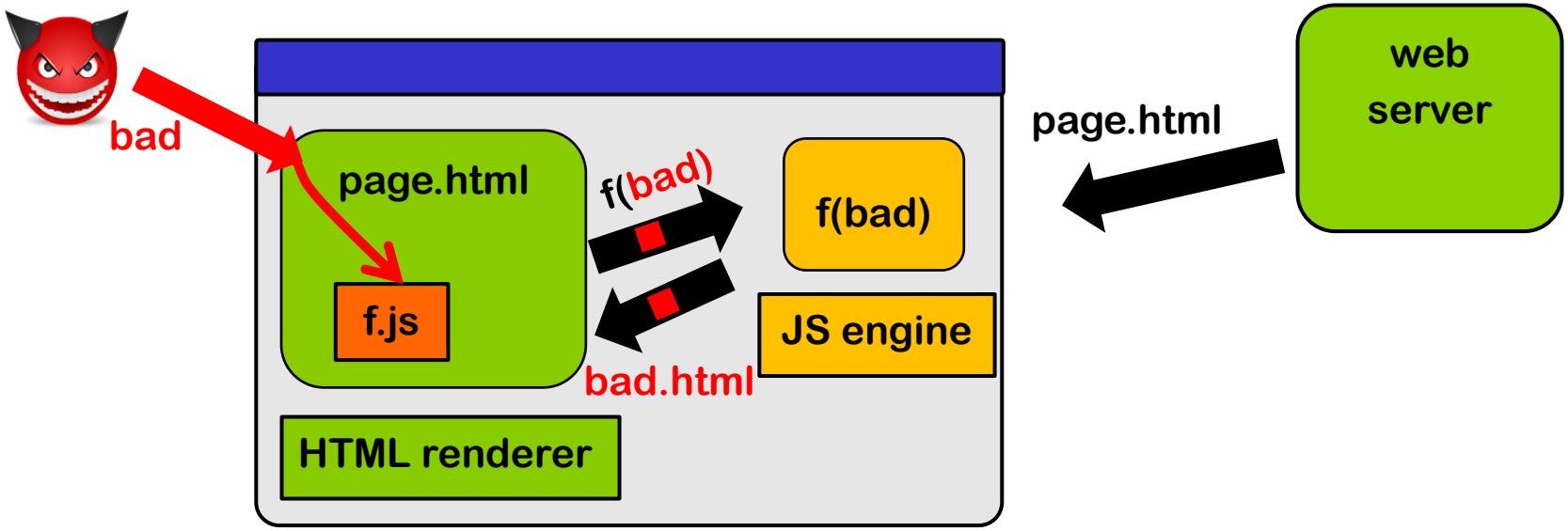
A malicious newName could be Eve</h1><script someAttackScript();</script> //

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!)




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.