Software Security

Secure INPUT handling

Erik Poll

Digital Security

Radboud University Nijmegen

Recap: most flaws are INPUT handling flaws

Input problems dominate Top N lists,
esp. memory corruption & injection attacks

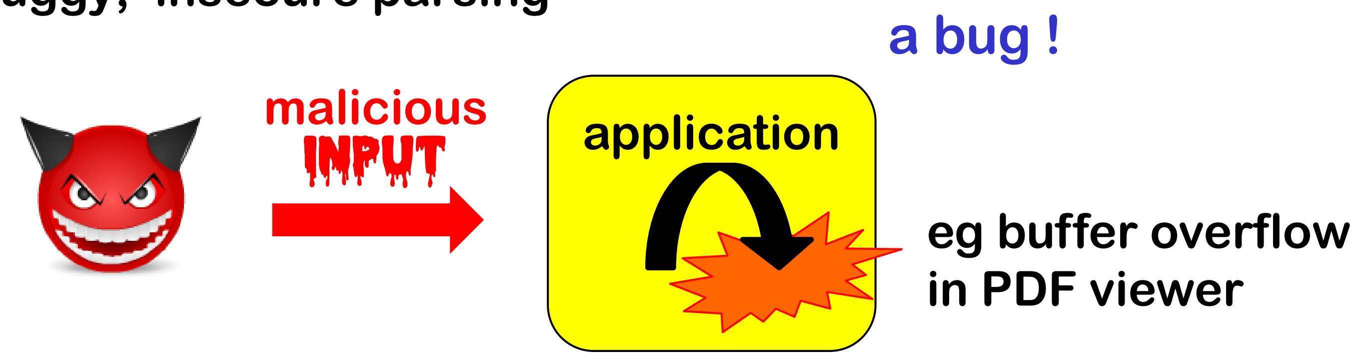
Most common other kinds: access control flaws

1. Out-of-bounds Write (CWE-787)	10. Unrestricted Upload of Dangerous File Type (CWE-434)	18. Hardcoded Credentials (CWE-798)
2. Cross Site Scripting (XSS) (CWE-79)	11. Missing Authorization (CWE-862)	19. Server-Side Request Forgery (CWE-918)
3. SQL injection (CWE-89)	12. NULL Pointer Deference (CWE-476)	20. Missing Authentication (CWE-306)
4. Use After Free (CWE-416)	13. Improper Authentication (CWE-287)	21. Race Condition (CWE-362)
5. OS Command Injection (CWE-78)	14. Integer Overflow or Wraparound (CWE-190)	22. Improper Privilege Management (CWE-269)
6. Improper Input Validation (CWE-20)	15. Deserialization of Untrusted Data (CWE-502)	23. Code Injection (CWE-94)
7. Out-of-bounds Read (CWE-125)	16. Command Injection (CWE-77)	24. Incorrect Authorization (CWE-863)
8. Path Traversal (CWE-22)	17. Improper Restriction of Operations on Memory Buffer Bounds (CWE-119)	25. Incorrect Default Permissions (CWE-276)
9. Client-Side Request Forgery (CSRF) (CWE-352)		

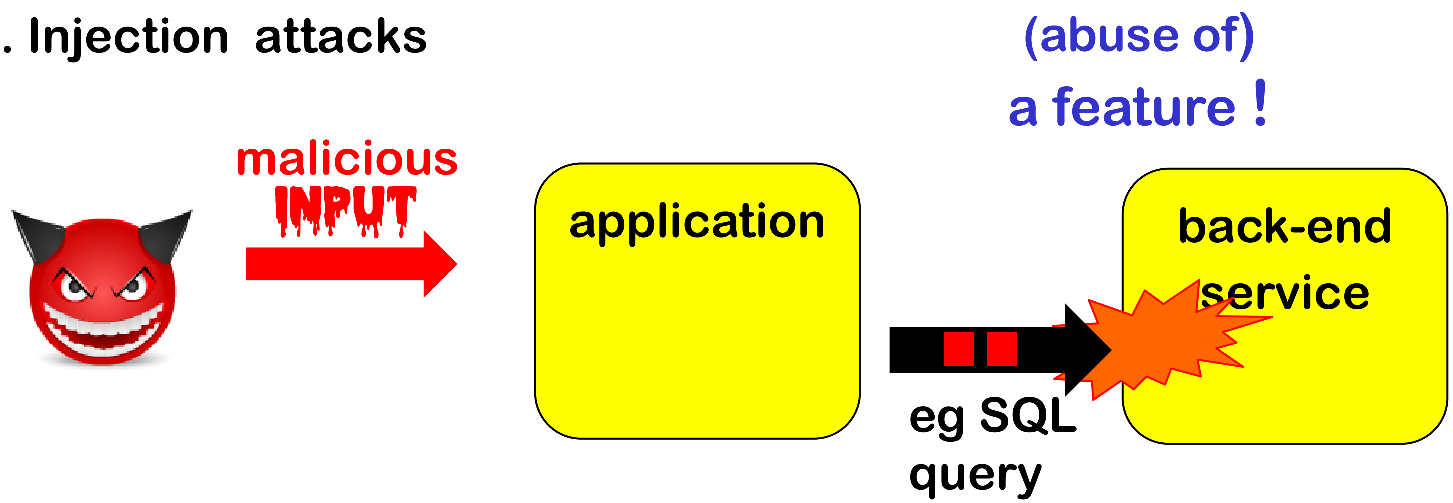
CWE Top 25 (2023 edition)

Two types of input problems: bugs vs features

1. Buggy, insecure parsing

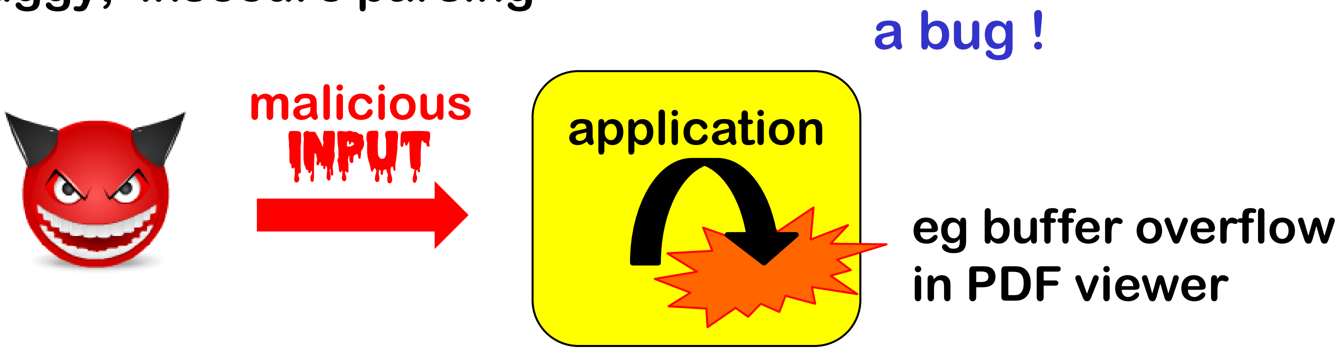


2. Injection attacks



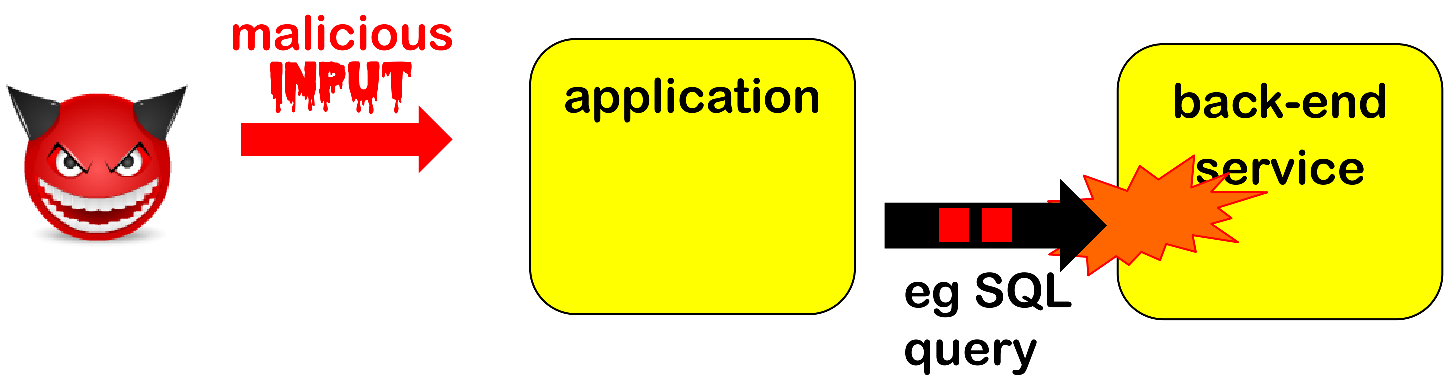
Two types of input problems: bugs vs features

1. Buggy, insecure parsing

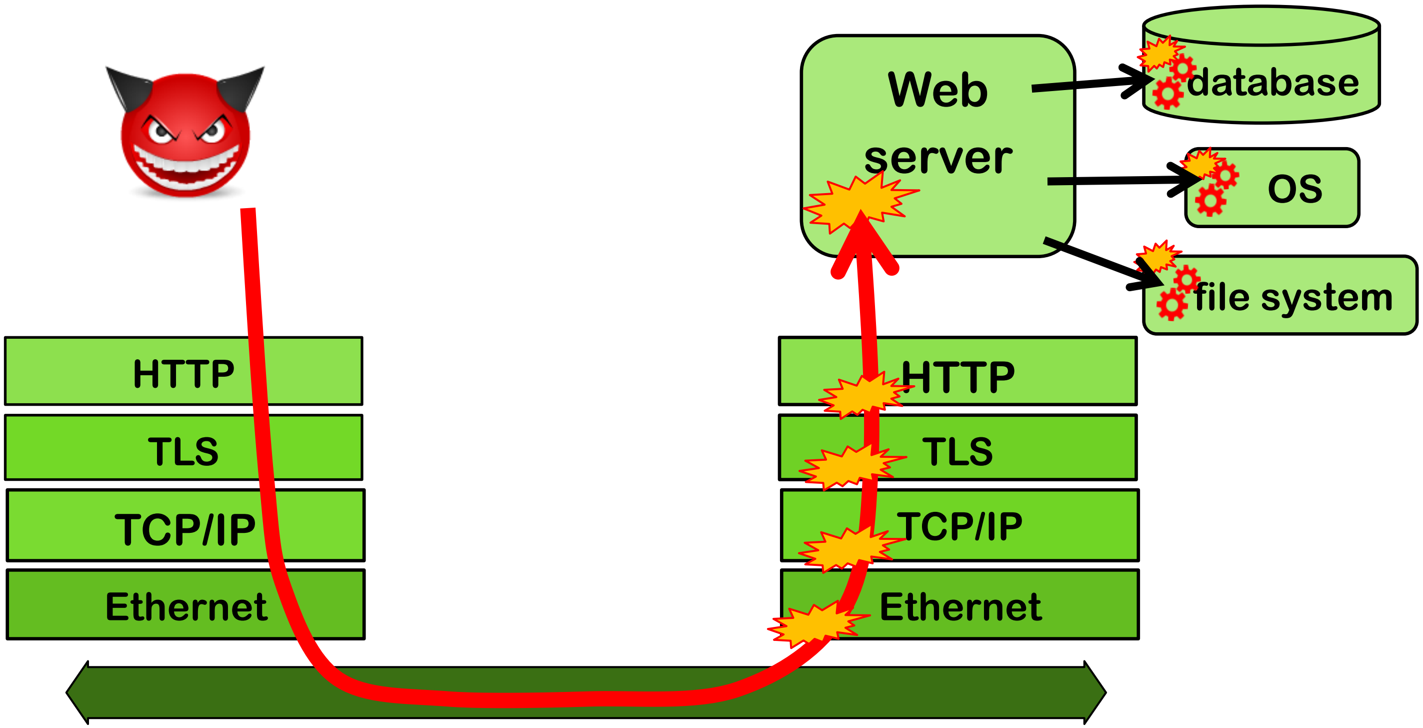


2. *Correct, but unintended parsing*

(abuse of)
a feature !



Why so many & so many different kinds?



Big attack surface in application, the underlying protocol stack, and external services.

Why so many & so many different kinds?

- **Many input languages**

  incl. data formats (URLs, filenames, email addresses, X509, ...)
  protocols (eg. in network stack: 4G, Bluetooth, TCP/IP, Wifi, HTTP(S), ...)
  file formats (Word, PDF, HTML, audio/video formats, JSON, XML, ...)
  script/programming languages (SQL, OS commands, JavaScript, ...)
  ...

- **Complex** input languages and formats

  eg. look at https://html.spec.whatwg.org for HTML or https://url.spec.whatwg.org *and* https://www.rfc-editor.org/rfc/rfc3987 for URLs

- **Sloppy definitions** of input languages and formats

- **Expressive** languages and formats

  eg. *macros* in Office formats, *SMB protocol* for Windows file names, *JavaScript* in HTML & PDF, `eval()` in programming languages, ...

Some of these factors also explain the success of fuzzing.

Audience poll

How should you defend against input problems?

Possibly by input *validation*

Probably NOT by input *sanitisation*

It's a common mistake to think that input validation and input sanitisation are the best or only defences !

It's also a common mistake to confuse sanitisation & validation!

Preventing input handling problems

I. Basic protection primitives:

Validation, Sanitisation, Canonicalisation

II. Tackling buggy parsing with LangSec

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

a) Input vs output sanitisation

b) Taint Tracking

c) Safe builders

Case study: XSS

I. The three basic protection mechanisms

a) Canonicalisation
b) Validation
c) Sanitisation

Canonicalisation, Validation, Sanitisation

1. Canonicalisation: *normalise* inputs to canonical form

   E.g. convert 10-31-2021 to 31/10/2021

   www.ru.nl/ to www.ru.nl

   J.Smith@Gmail.com_ to jsmith@gmail.com

2. Validation: *reject* 'invalid' inputs

   E.g. reject Nov 32nd 2024 or negative amounts

3. Sanitisation: *fix* 'dangerous' inputs

   E.g. convert <script> to <script>

Beware: Often confused ! Sometimes combined !

Many synonyms: escaping, encoding, filtering, neutralising, ...

Invalid inputs could be fixed instead of rejected as part of validation.

Which of these operations should be done first?

a) Canonicalisation (aka Normalisation)

There may be *many* ways to write the same thing, eg.

- upper or lowercase letters eg `s123456` vs `S123456`
- trailing spaces eg `s123456 ` vs `s123456`
- trailing `/` in a domain name, eg `www.ru.nl/`
- trailing `.` in a domain name, eg `www.ru.nl.`
- ignored characters or sub-strings, eg in email addresses:

  `name+redundantstring@bla.com`
- `..` `.` `~` in path names
- file URLs `file://127.0.0.1/c|WINDOWS/clock.avi`
- using either `/` or `\` in a URL on Windows
- Unicode encoding eg `/` encoded as `\u002f`

  Beware: some forms of encoding are not meant as form of sanitisation

a) Canonicalisation

- Data should always be put into canonical form *before* any further processing, esp.
  - *before* validation
  - *before* using the data in security decisions

- But: the canonicalisation operation itself may be abused, for instance to waste CPU cycles or memory
  - eg with a zip bomb of XML bomb

(Btw: a docx file is a zip file!)

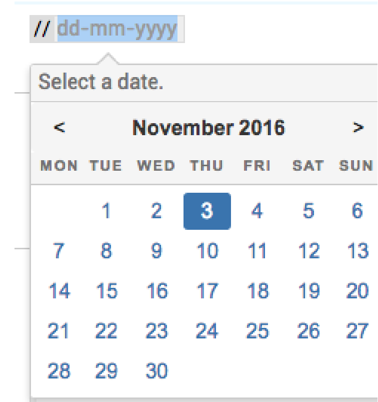
b) Validation

Many possible forms of patterns for validations

- **Eg. for numbers:**
  - **positive, negative, max. value, possible range?**
  - **Luhn mod 10 check for credit card numbers**
- **Eg. for strings:**
  - **(dis)allowed characters or words**
  - **More precise: regular expressions or context-free grammars**
    - **Eg for RU student number (s followed by 6 digits), valid email address, URL, …**

Unfortunately, regular expressions and context-free grammars are not expressive enough for many complex input formats (eg email address, JPG, PDF,…) ☹

b) Validation techniques



- Indirect selection
  - Let user choose from a set of legitimate inputs; User input never used directly by the application
  - Most secure, but cannot be used in all situations; also, attacker may be able to by-pass the user interface to still enter invalid data, eg by messing with HTTP traffic
- Allow-listing (aka white-listing)
  - List *valid* patterns; *accept* input if it matches
  - Instance of a positive security model
- Deny-listing (aka black-listing)
  - List *invalid* patterns; *reject* input if it matches
  - Least secure, given the big risk that some dangerous patterns are overlooked
  - Instance of a negative security model

c) Sanitisation aka encoding

Commonly applied to prevent injection attacks, eg.

- replacing " by \" to prevent SQL injection, aka escaping
- replacing < > by &lt &gt to prevent HTML injection & XSS
- replacing script by xxxx to prevent XSS
- putting quotes around an input, aka quoting
- removing dangerous characters or words, aka filtering

NB after sanitising, changed input may need to be *re-validated*

As for validation, we can use allow-lists or deny-lists for replacing or removing characters & keywords

Validation patterns can get COMPLEX

A regular expression to validate email adressess

```
\A(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*
 |  "(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]
      |  \\[\x01-\x09\x0b\x0c\x0e-\x7f])*")
@ (?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?
  |  \[(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
       (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:
          (?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]
          |  \\[\x01-\x09\x0b\x0c\x0e-\x7f])+)
     \])\z
```

See http://emailregex.com for code samples in various languages

Or read RFCs 821, 822, 1035, 1123, 2821, 2822, 3696, 4291, 5321, 5322, and 5952 and try yourself!

Parse, don't validate!

If input validation requires parsing, then parse & don't just validate!

Eg instead of having a validation function

```
boolean isValidURL(String s)
```

we could have a parsing function

```
URL createURL(String s) throws InvalidURLException
```

which returns some datatype `URL` (eg. an object, record, or struct) that comes with relevant operations (eg. to extract domain, protocol).

Advantages of parsing? Disadvantages?

- You cannot forget validation, as then code won't type check ☺
- No duplication of parsing code ☺ - in validation & subsequent parsing.
- More work, at least initially, to define all these types such as `URL` ☹
  Though maintenance should be easier…

Spot the defect

```
char buf1[MAX_SIZE], buf2[MAX_SIZE];
// make sure url is valid URL and fits in buf1 and buf2:
    if (!isValid(url)) return;
    if (strlen(url) > MAX_SIZE – 1) return;
// copy url excluding spaces, up to first separator, ie. first ’/’, into buf1
    out = buf1;
    do { // skip spaces
        if (*url != ’  ’) *out++ = *url;
    } while (*url++ != ’/’);
    strcpy(buf2, buf1);
```

Loop fails to terminate flaw for **URLs without /**

Exploited by **Blaster worm**

[Code sample from presentation by Jon Pincus]

Parse, don't validate?

```
char buf1[MAX_SIZE], buf2[MAX_SIZE];
// make sure url is valid URL and fits in buf1 a
    if (!isValid(url)) return;
    if (strlen(url) > MAX_SIZE – 1) return;
// copy url excluding spaces, up to first separator, ie. first '/', into buf1
    out = buf1;
    do { // skip spaces
        if (*url != ' ') *out++ = *url;
    } while (*url++ != '/');
    strcpy(buf2, buf1);
```

Why not parse the url into some URL object/datatype as part of the isValid() method?

The (partial) parsing by this loop possibly repeats work done in isValid()

[Code sample from presentation by Jon Pincus]

Sanitisation nightmares: XSS

Many places to include Javascript and many ways to encode

Eg <script> alert('Hi'); </script> **can be injected as**

- <body onload=alert('Hi')>
- <b onmouseover=alert('Hi')>Click here!
- <img src="http://some.url.that/does/not/exist" onerror=alert('Hi');>
- <img src=j&#X41vascript:alert('Hi')>
- <META HTTP-EQUIV="refresh" CONTENT="0;url=data:text/html;base64,PHNjcmlwdD5hbGVydCgndGVzdDMnKTwvc2NyaXB0Pg">

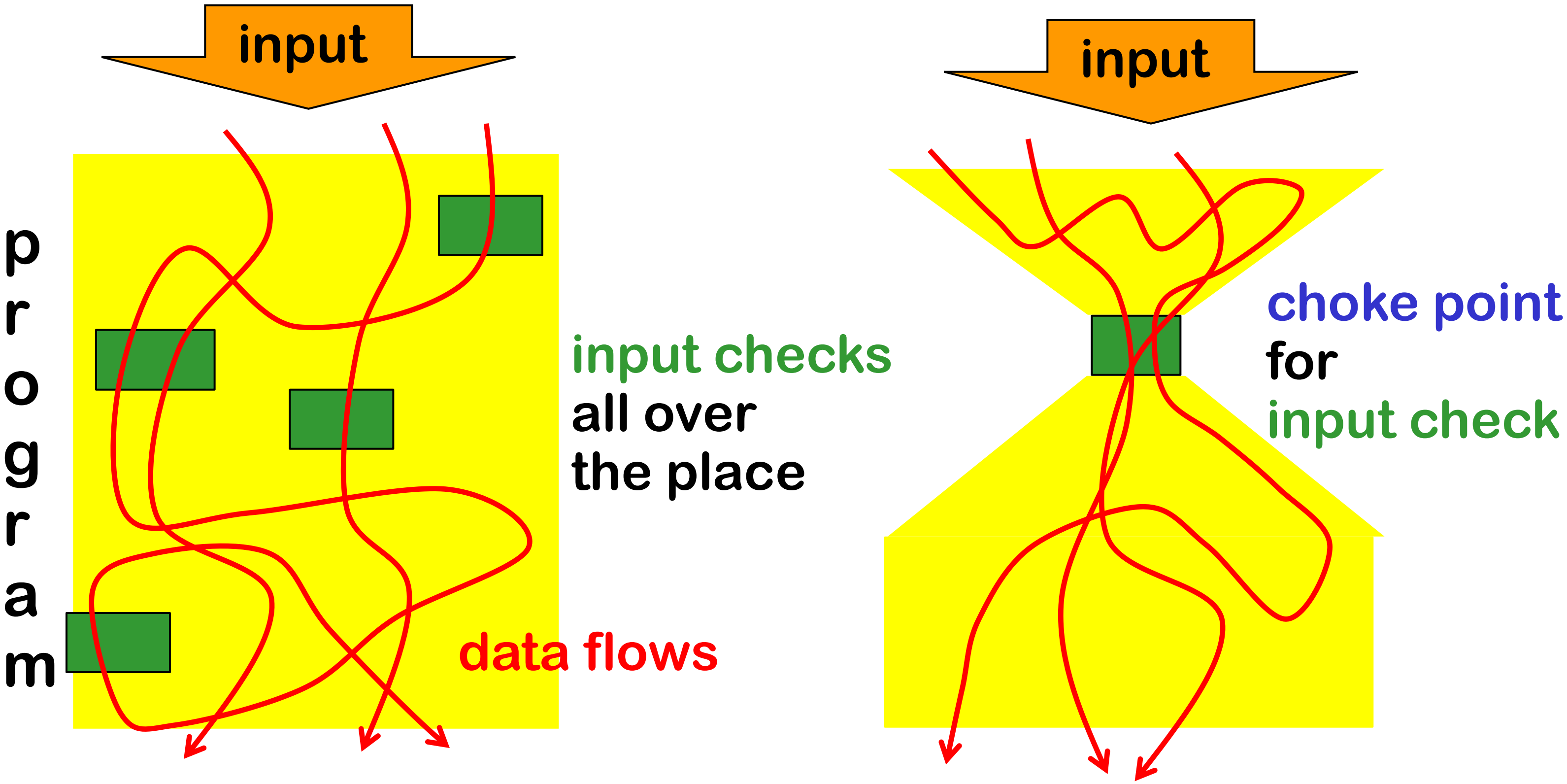
Root cause: COMPLEXITY of HTML format (https://html.spec.whatwg.org)

For a longer lists of XSS evasion tricks, see

https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

Where to canonicalise, valididate or sanitise:

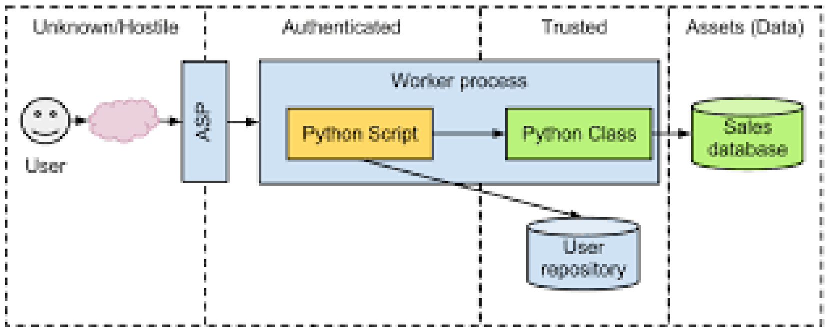
Best done at clear choke points in an application

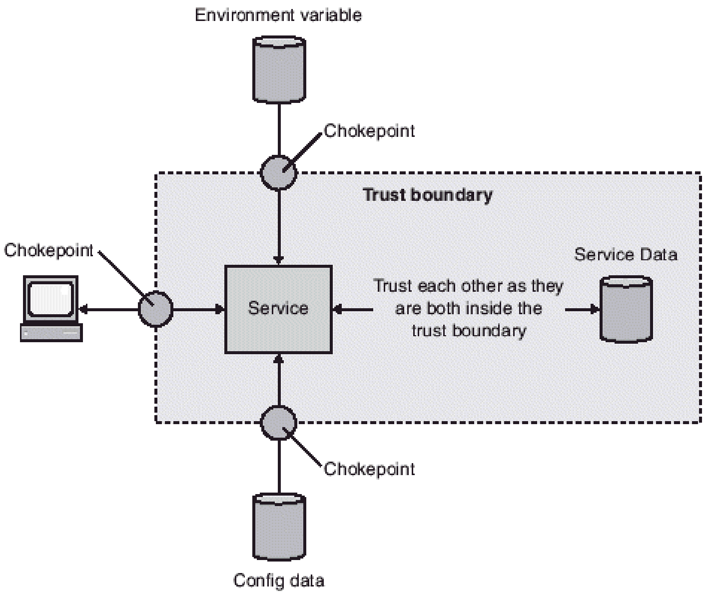


Trust boundaries & choke points

Identifying trust boundaries useful to decide *where* to have choke points

- **in a network, on a computer, or within an application**





II. Tackling insecure & incorrect parsing
-
using the LangSec approach

Buggy parsing – two different kinds

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'

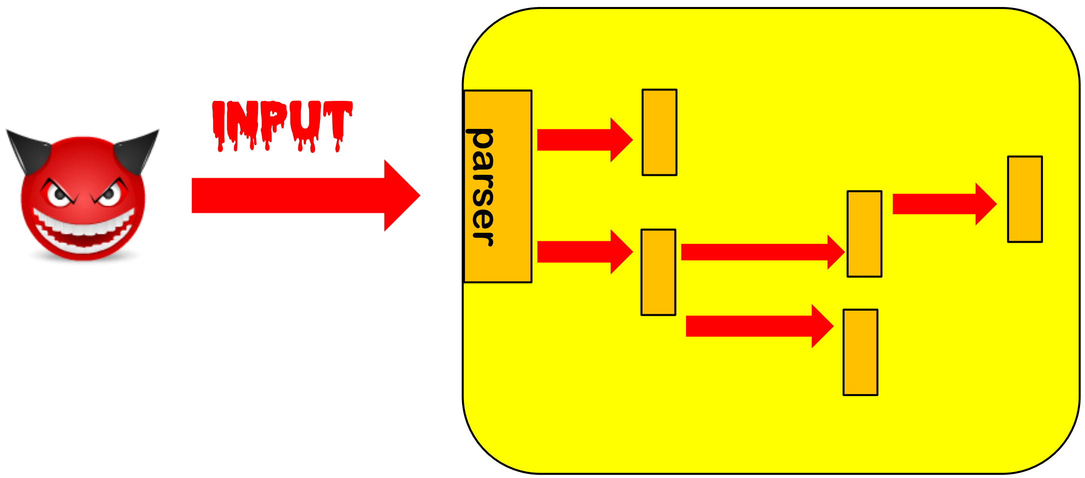
- The 'Lang' in 'LangSec' refers to *input* languages, not *programming* languages.

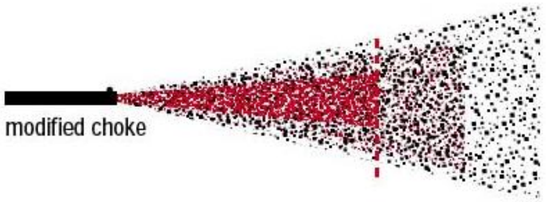
Root causes / anti-patterns

- 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





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

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

LangSec concepts

- **Shotgun parser:** scattershot 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**

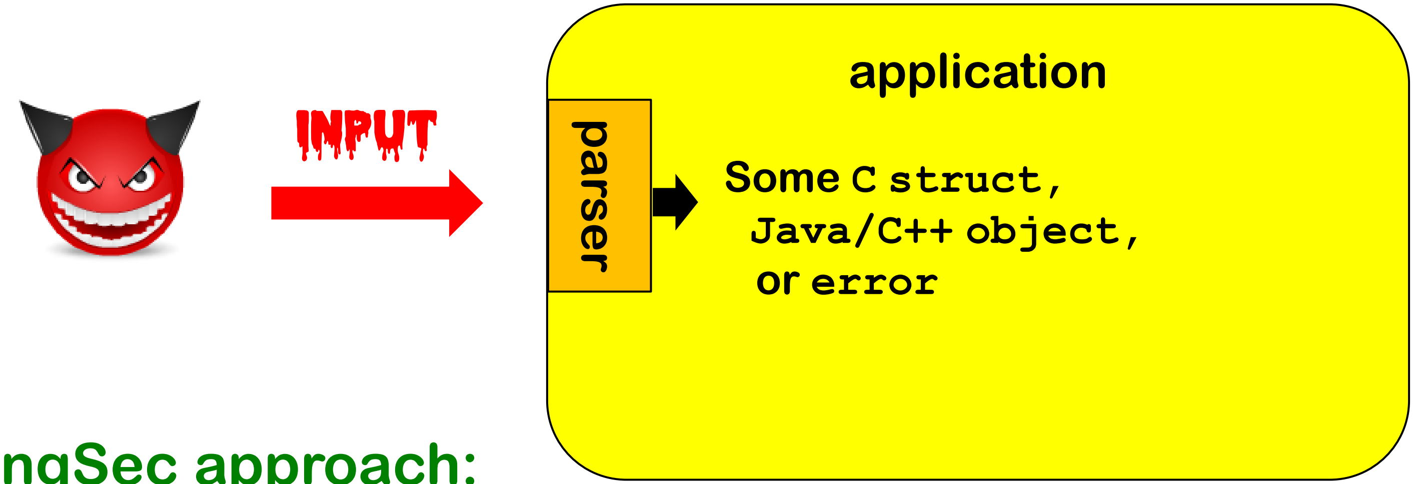
3. ***complete parsing before processing***

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 language spec
- Generated parser code
- Complete parsing before processing

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

LangSec in slogans

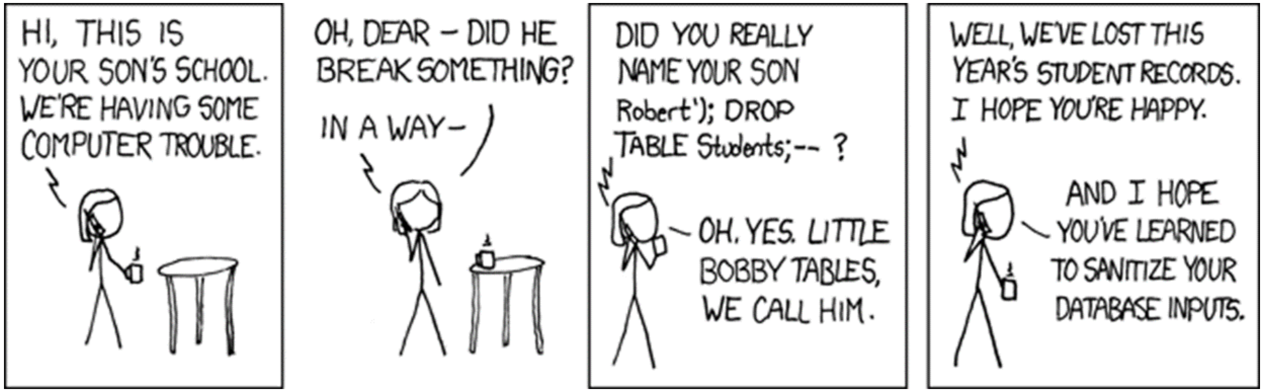

STOP
WEIRD
MACHINES
CONTEXT
FREE
OR
REGULAR

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

III. How (not) to prevent unintended parsing, i.e. injection attacks



How & *where* to prevent injection attacks?

Suppose we are worried about SQL injection via a website

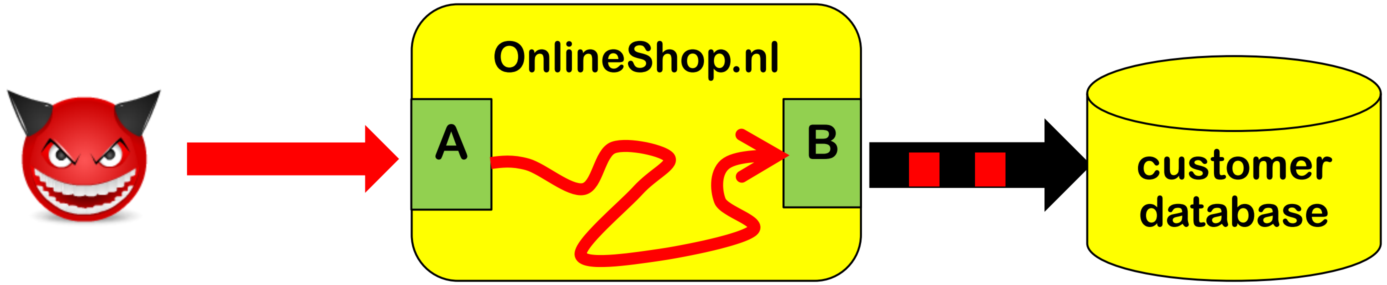
- Should we validate, sanitise, or both to prevent SLQi?
- if so, where? At point A or B?

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

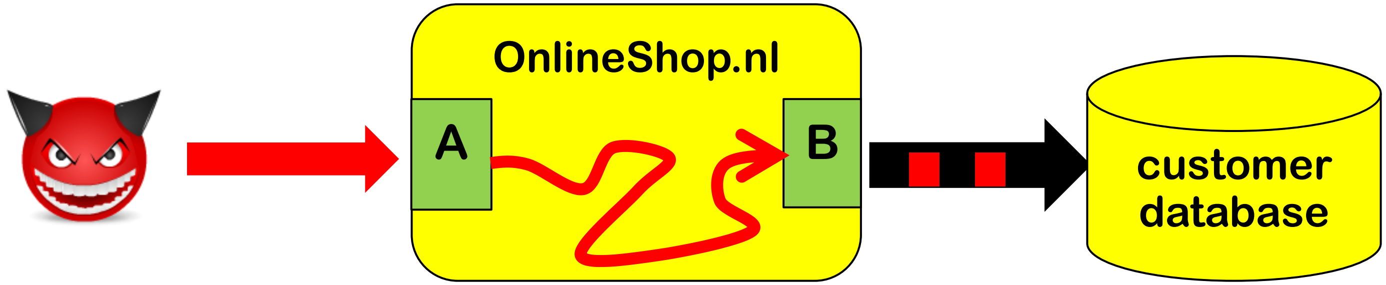
Pros?

- 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 `\'`

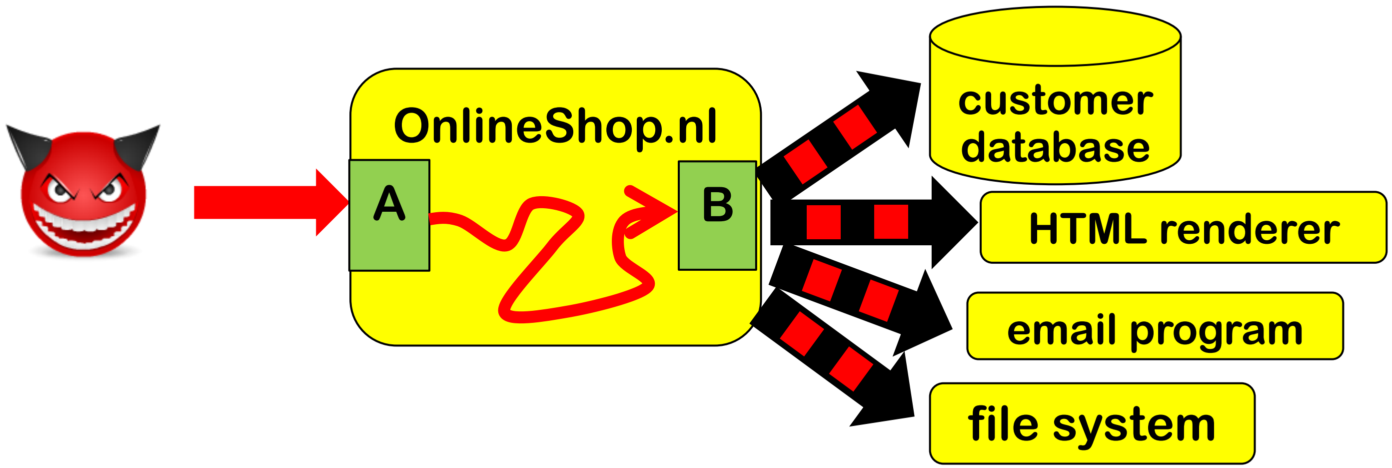
Pros?

- 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

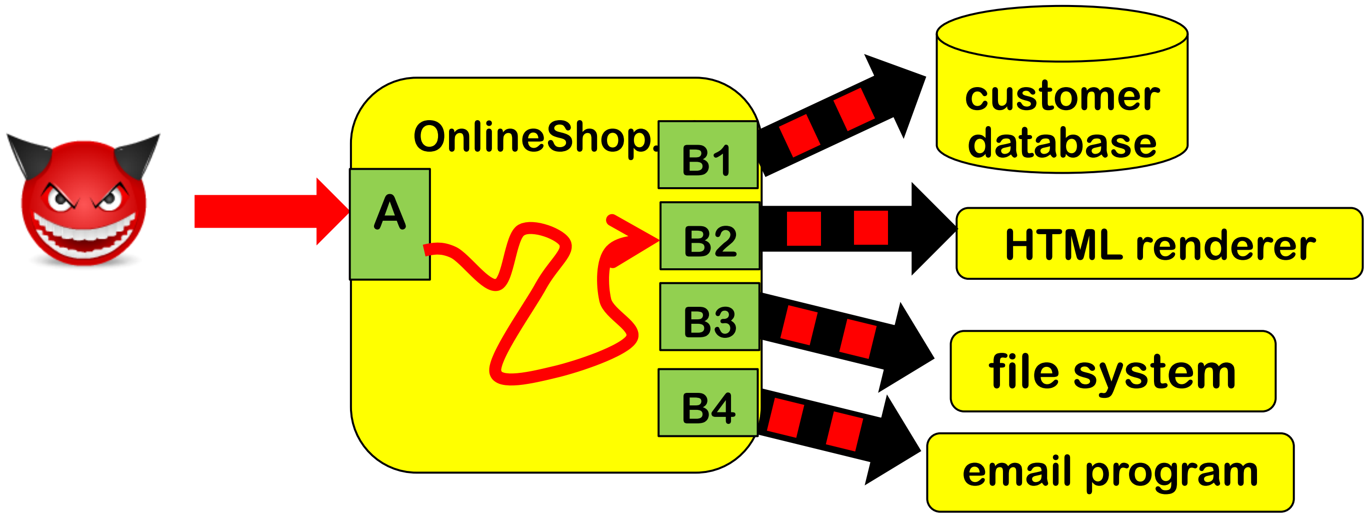


Different **escaping needed to prevent SQLi, XSS, path traversal, OS command injection, ...**

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

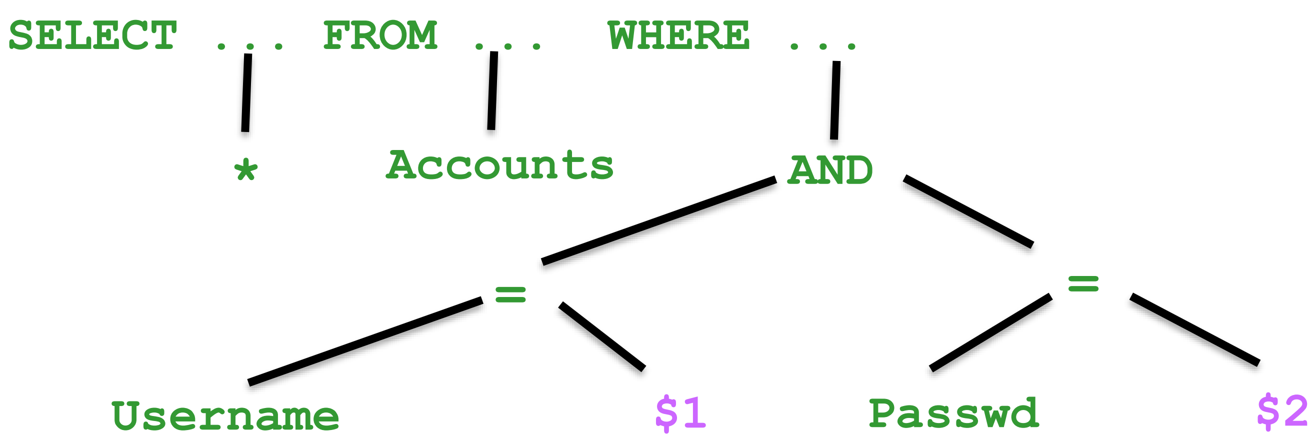
```
String updateString =
  "SELECT * FROM Account WHERE Username"
   + username + "AND Password =" + password;
stmt.executeUpdate(updateString);
```

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

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
  - Taint mode was removed in Perl 6
- Microsoft Office uses taint tracking of documents using the Mark of the Web to block or warn about macros in tainted documents

  Rules tightened March 2022; Visual Basic depreciated May 2024

  https://techcommunity.microsoft.com/blog/windows-itpro-blog/vbscript-deprecation-timelines-and-next-steps/4148301
- Most SAST tools (incl. Fortify, discussed in SIO lecture, semgrep and CodeQL) use 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 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 have to 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

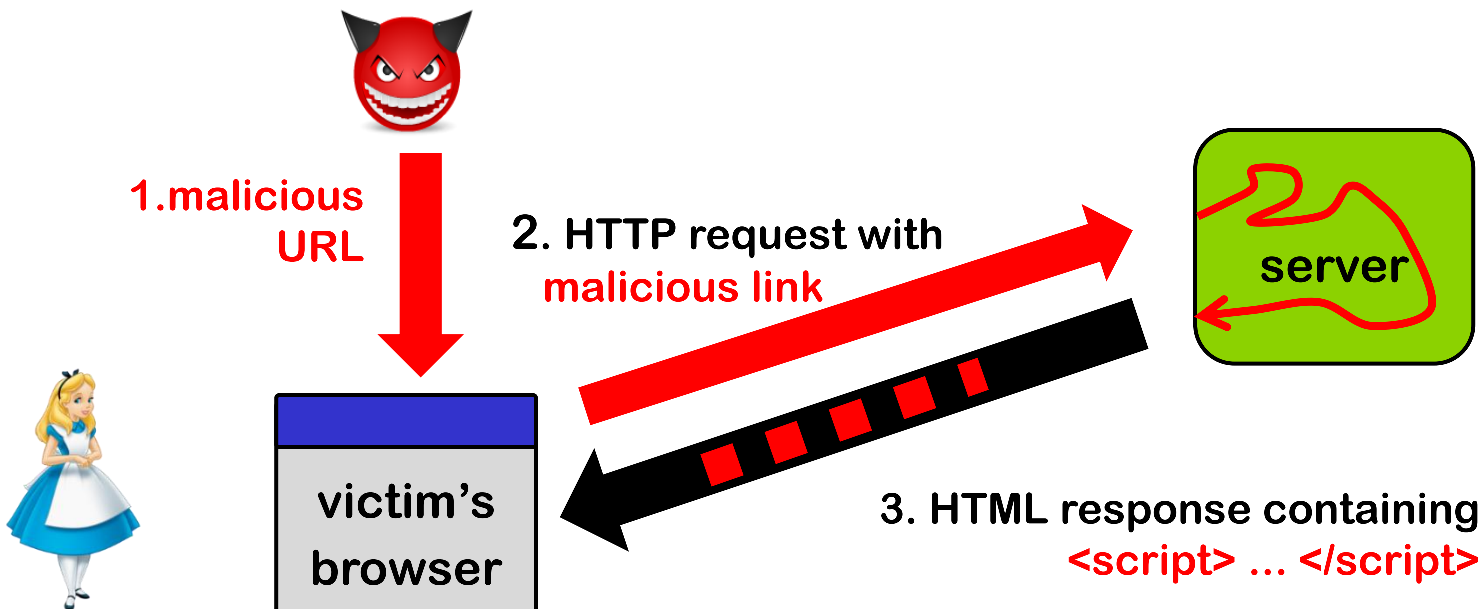
The messy business of preventing XSS

Reflected XSS attack

Attacker crafts malicious URL containing JavaScript

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

and tempts victim to click on this link

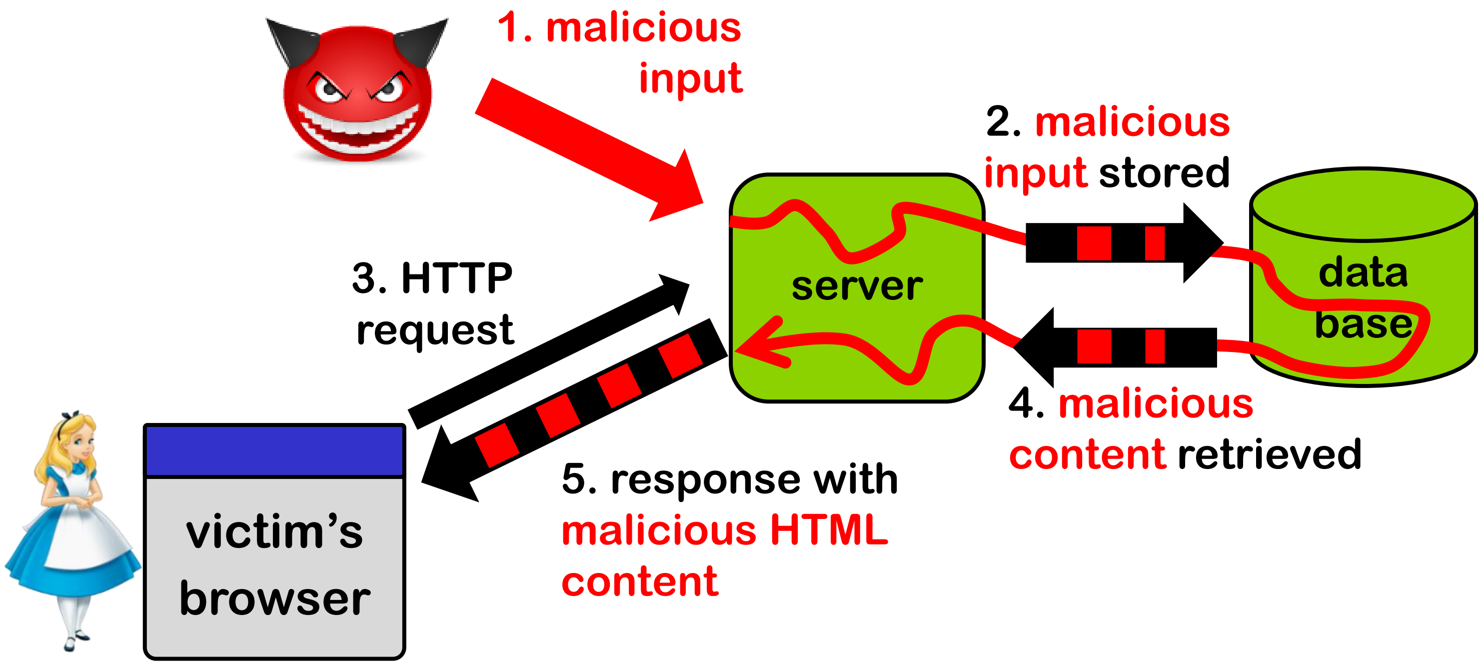


Could careful web server prevent this?

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 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 ${...}

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

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

How should the parameters be encoded here?

Encoding for the web - server-side

HTML encoding (eg of < and >)

URL encoding (eg of ?)

JavaString literal encoded (eg of ' and ")

```
'<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>'
```

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**
- ...