Software Security

Typical security problems, esp. INPUT problems

Erik Poll

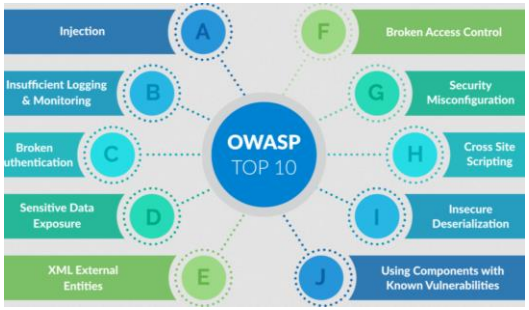
Digital Security

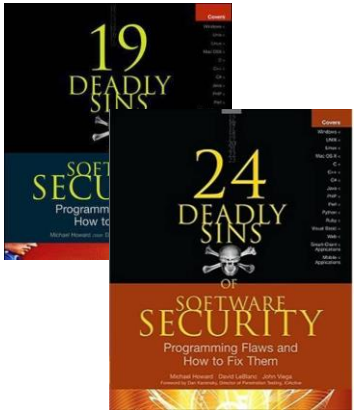
Radboud University Nijmegen

Classifications & rankings of security flaws

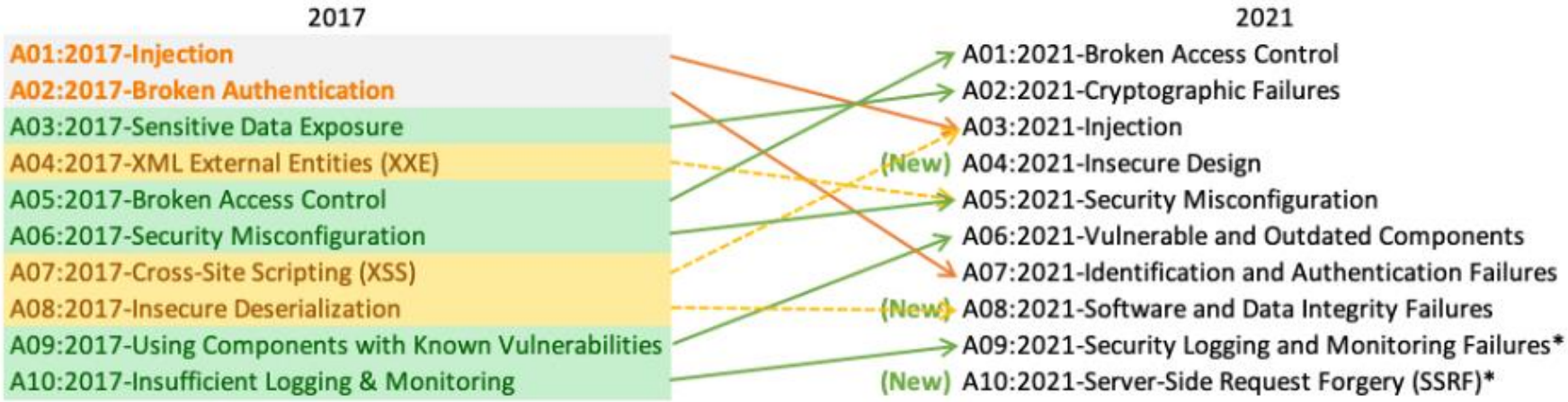
Many proposals to categorise & rank common security vulnerabilities in bug classes

- **OWASP Top 10**
- **SANS CWE Top 25**
- **24 Deadly Sins of Software Security**
- …
- …





OWASP Top Ten



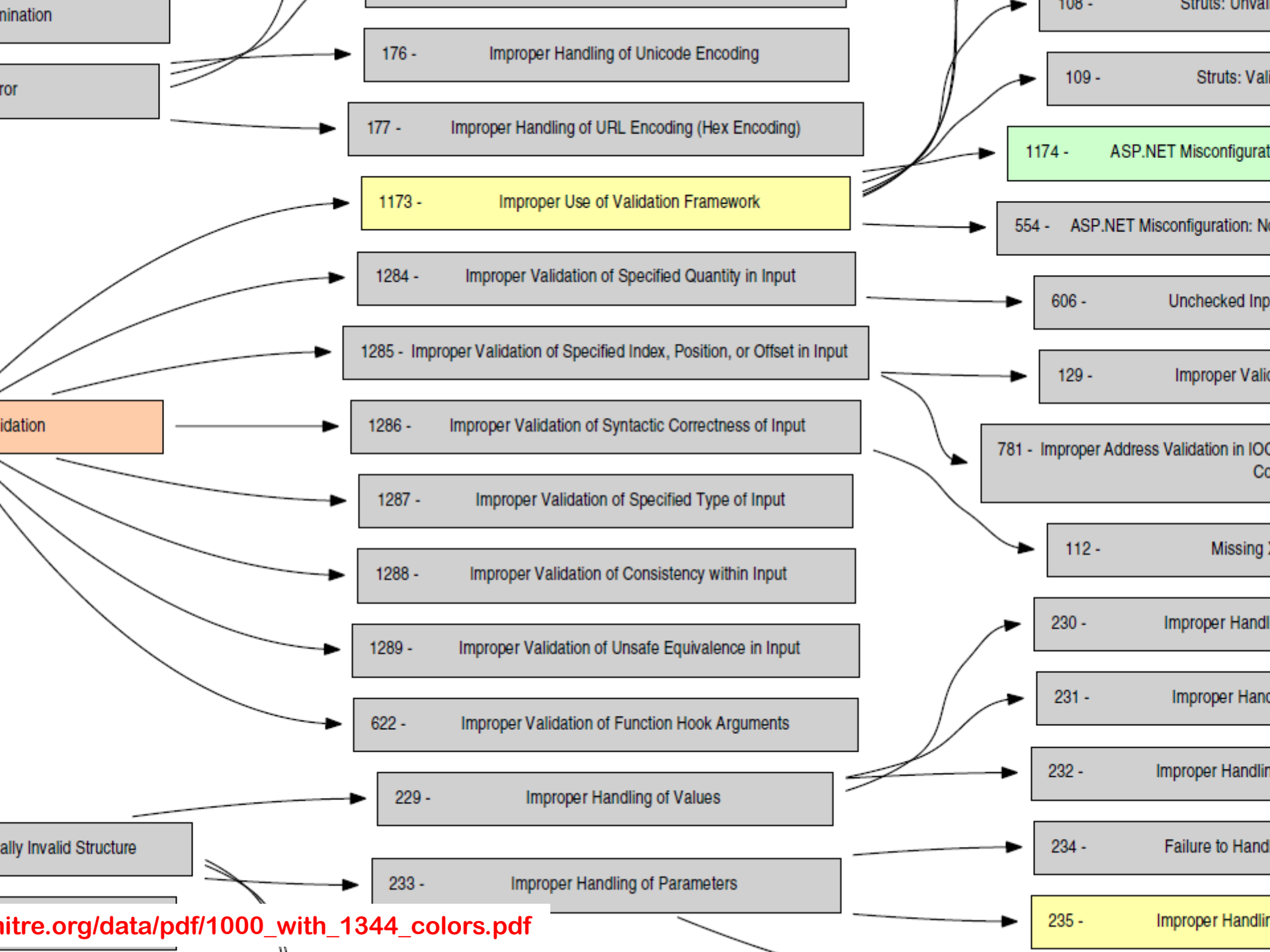
SANS CWE Top 25 [2021]

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

See https://cwe.mitre.org/top25/index.html

CWE Top 1357 [Nov 2023]

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



itre.org/data/pdf/1000_with_1344_colors.pdf

CVE, CWE, CRE

- CVE - Common *Vulnerability* Enumeration

https://cve.mitre.org

- CWE - Common *Weakness* Enumeration

https://cwe.mitre.org

Here weakness means 'bug class'
NB this is very non-standard use of the term!

- CRE - Common *Requirement* Enumeration

https://www.opencre.org

Recent initiative to standardise/relate requirements across (the many!) different security standards & guidelines

Top n lists of security flaws

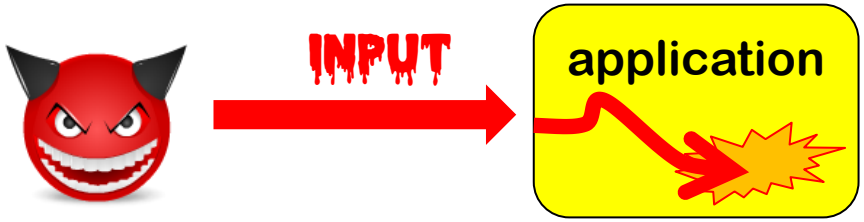
List and classifications of security flaws are

- very useful
  - for awareness & prevention – people keep making the same mistakes!
  - 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 are missing in generic taxonomies
- can be misleading & used incorrectly
  - e.g. 'lack of input validation' – more on that later

Tackling INPUT problems

High level observations

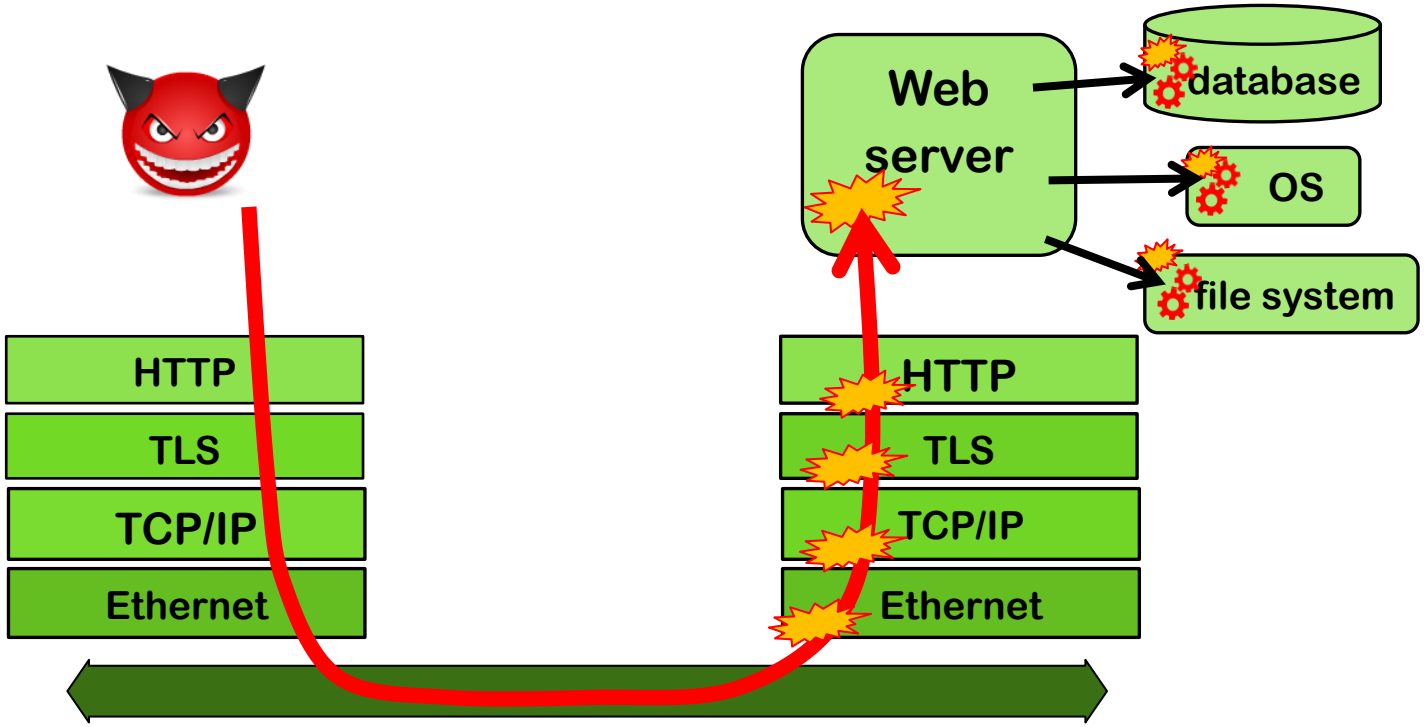
Most (all?) attacks involve malicious INPUT which ends up in a place where processing it causes software to 'go off the rails'



Input may be forwarded between systems to reach place where it does damage

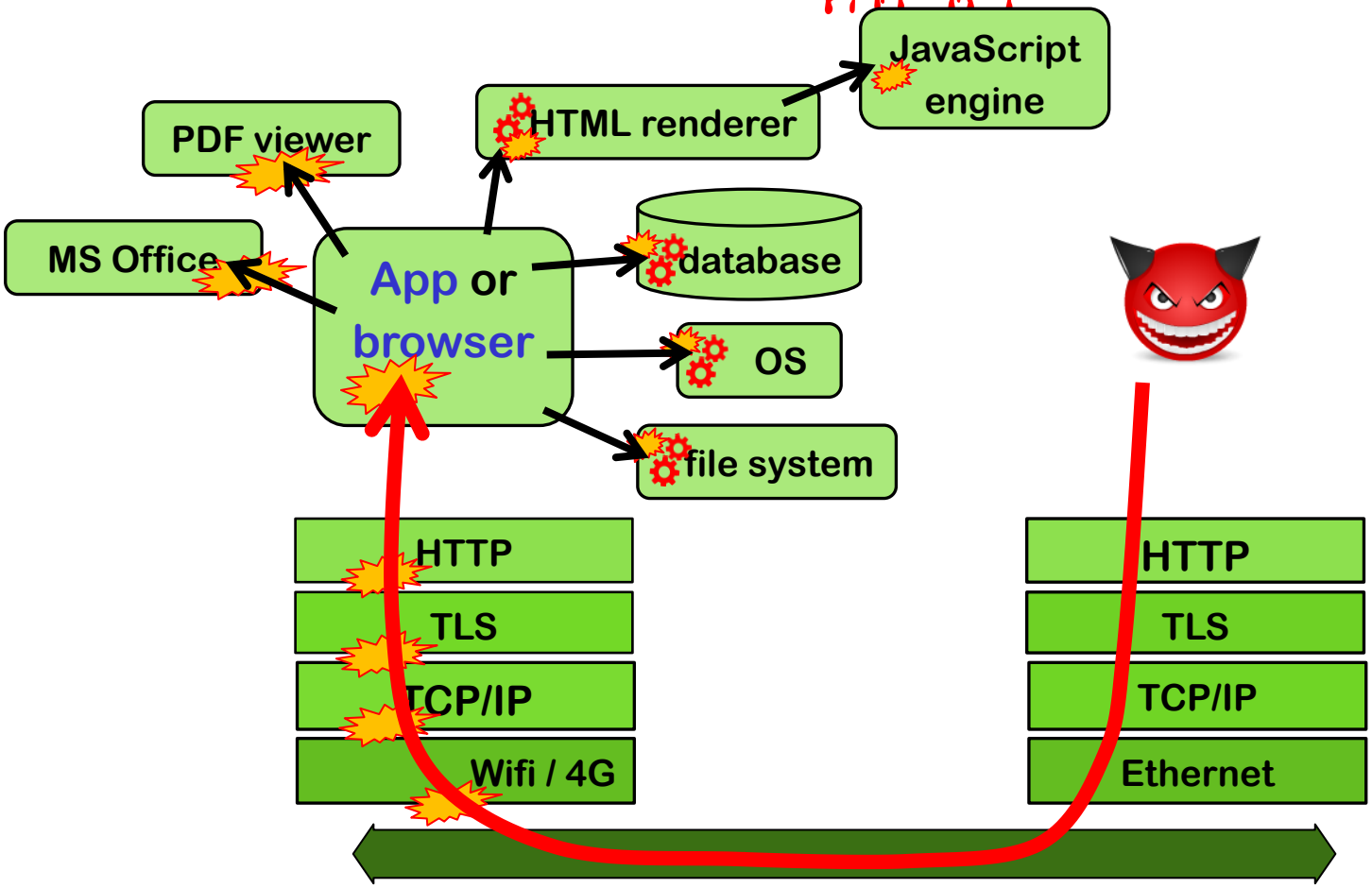
Are there structural approaches to combat these 100s of variants of input handling problems?

Attack surface for INPUT problems



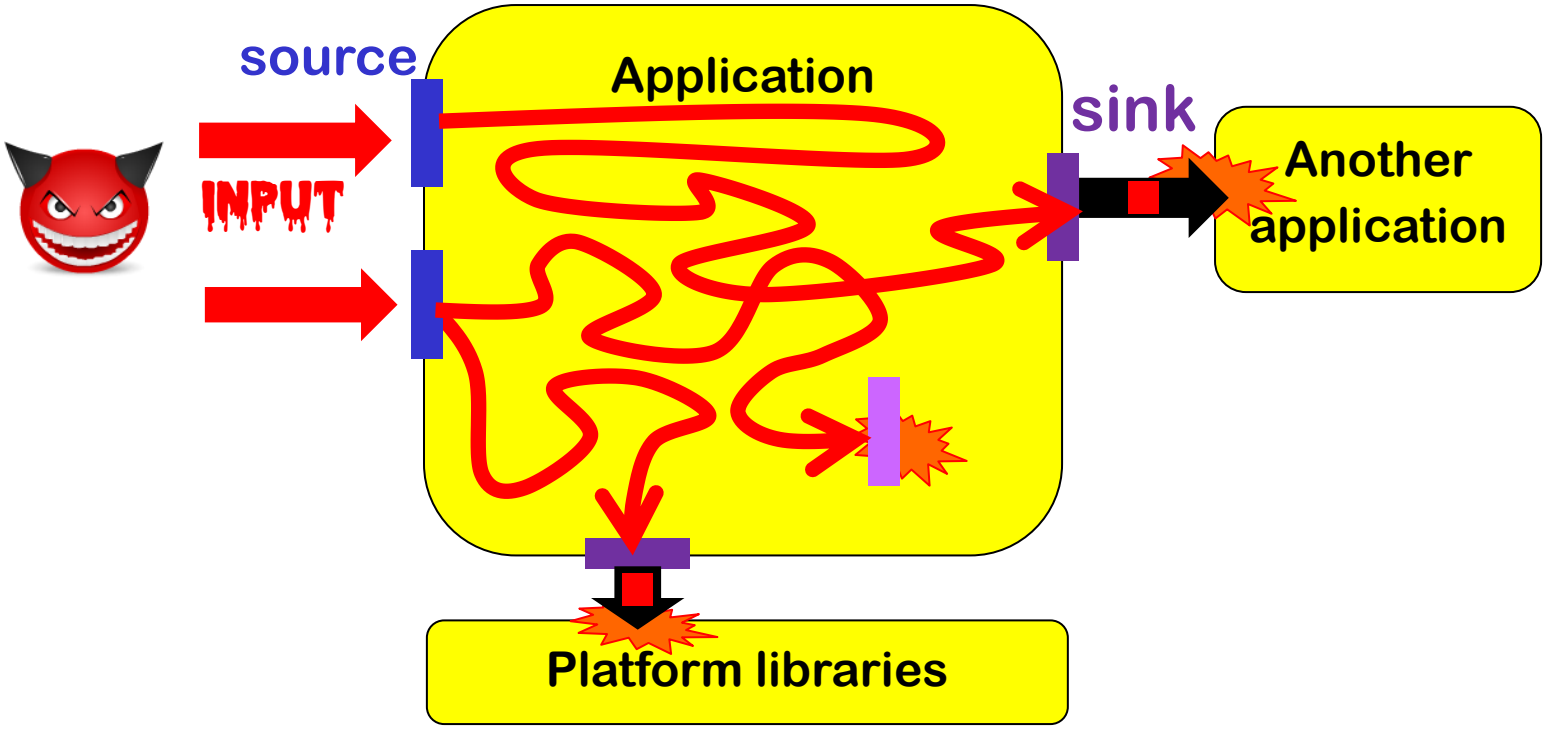
Big attack surface: inside application, in underlying protocol stack, and in external services.

Attack surface for INPUT problems



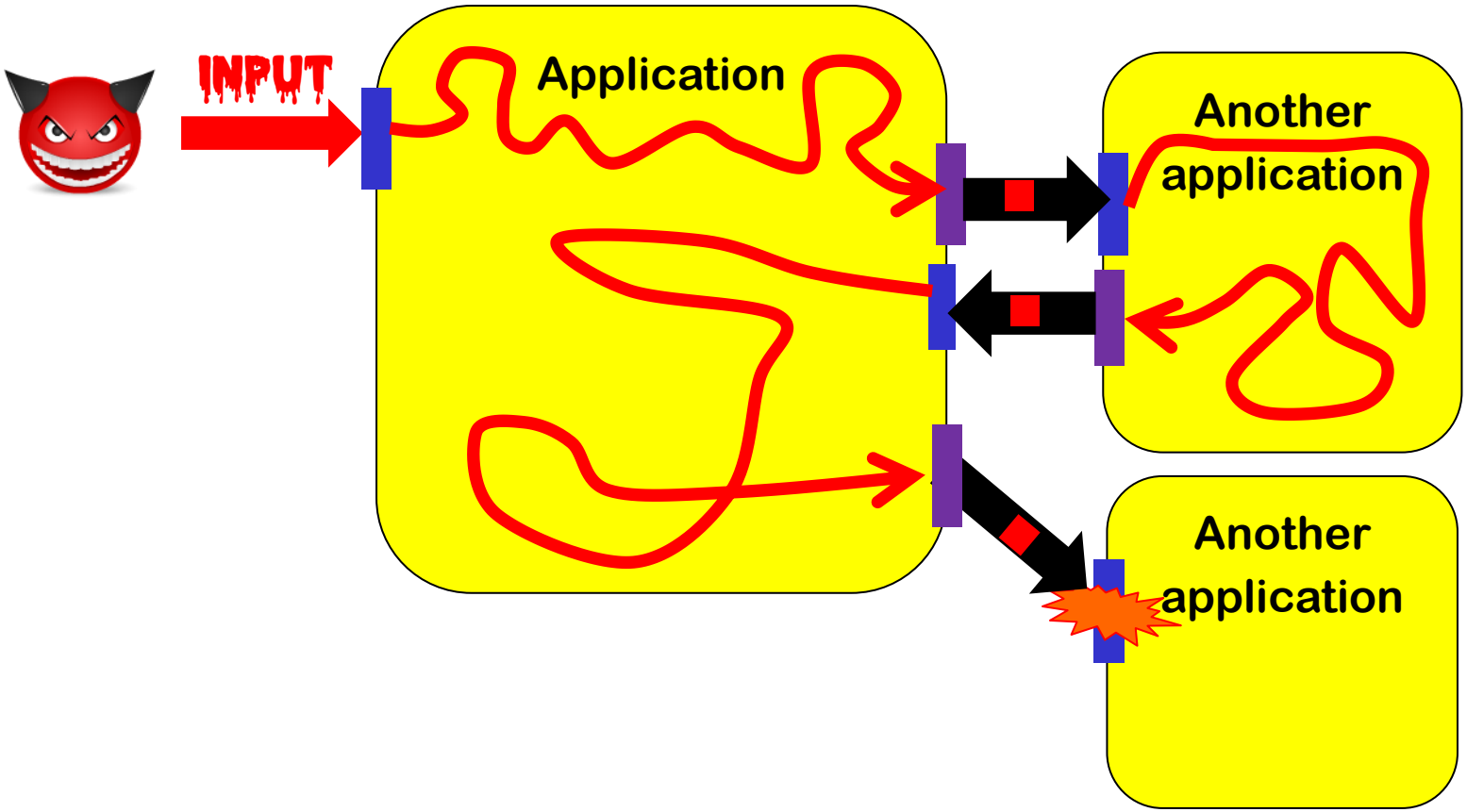
Terminology

Untrusted input travels as tainted data from source to sink



Sinks can be external APIs or internal functions / bugs

2-nd order attacks



Example: 2nd order SQL injection

Suppose I want to access Lejla's account

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

```
UPDATE users
   SET password='abcd1234'
 WHERE username='lejla' -- '
```

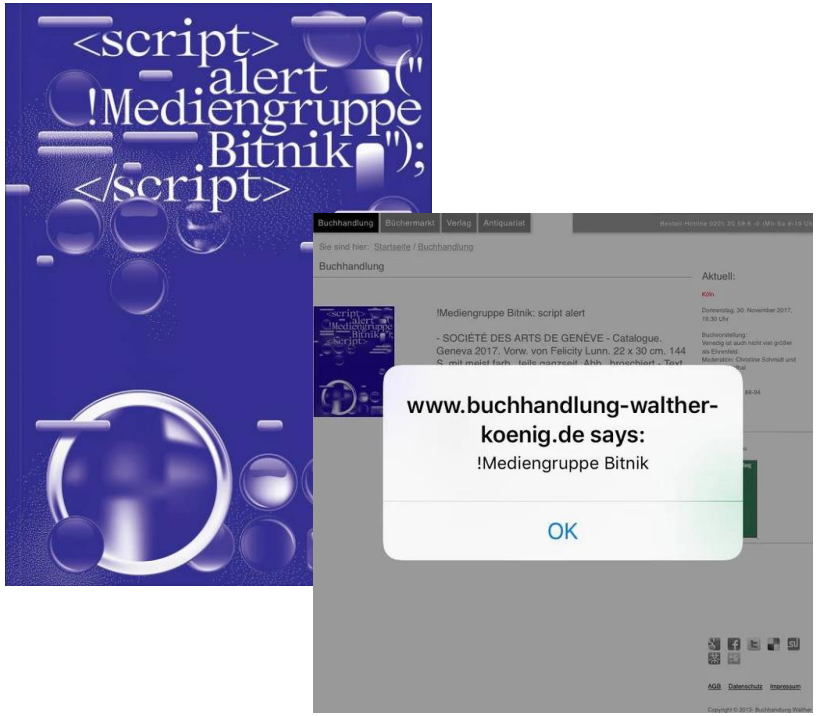
then I have reset Lejla's password

- 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

Expect the unexpected!

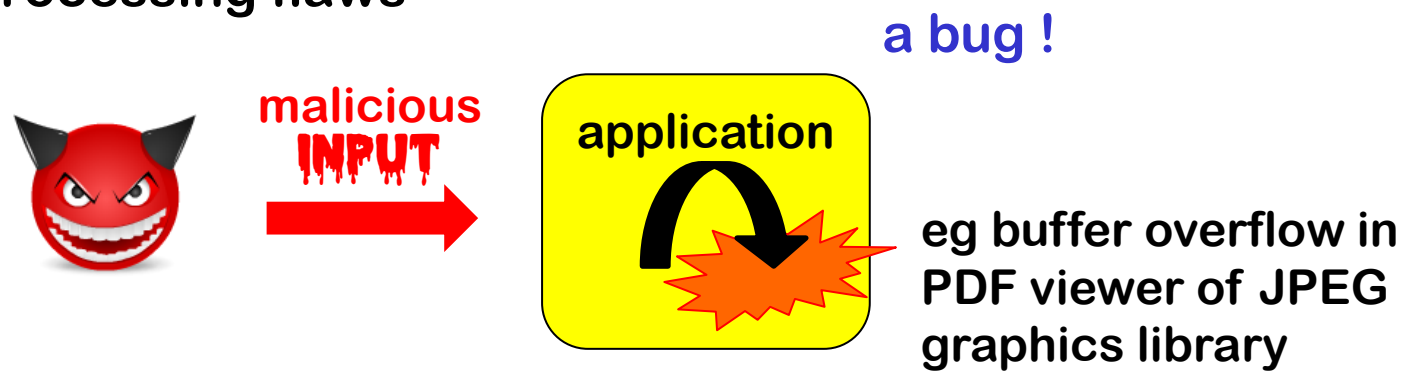
Malicious input can come from unexpected, ‘trusted’ sources



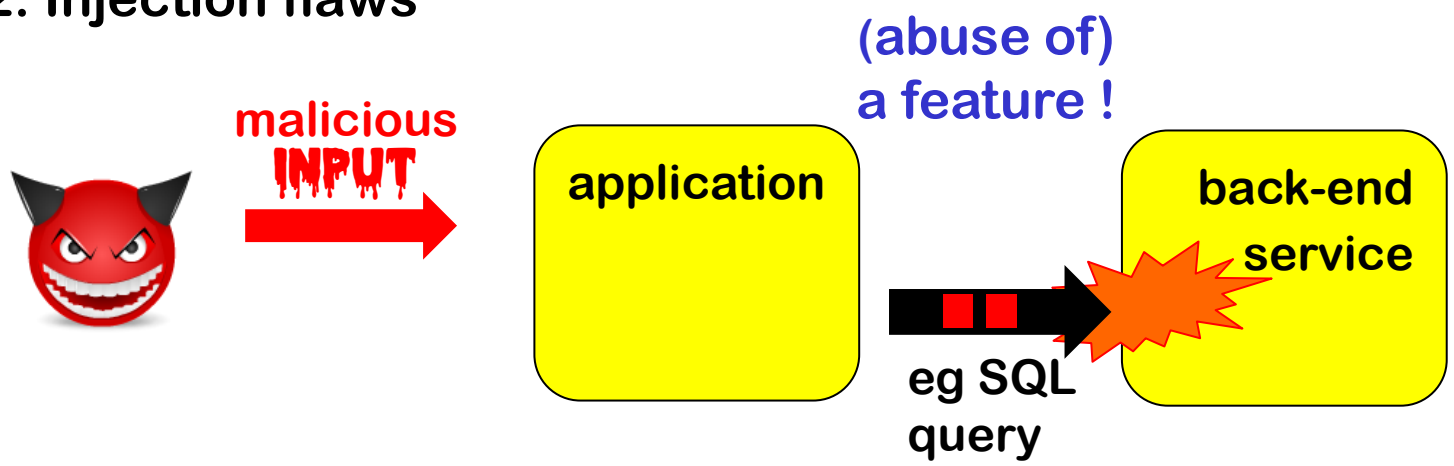
Talking about trusted vs untrusted (user) inputs can be misleading!

Two types of problems: bugs vs features

1. Processing flaws



2. Injection flaws



Recurring themes: parsing & languages

- Processing an input begins with parsing

- This depends on input language / format / protocol

  Eg TCP/IP packets, HTTP, HTML, SQL, X509, mp3, JPEG, webp, PDF, URL, email address, Word, Excel, …

- Input handling bugs often come down to parsing bugs
  - buggy parsing (eg buffer overflow in PDF parsing)
  - unintended parsing (eg parsing user input as SQL command)

Buggy parsing (1)

Buggy – insecure - parsing can cause security bugs:

- esp. if parser is written in memory unsafe language: memory corruption can lead to memory leaks, RCE, …
- Parsers written in memory safe language can still crash

High risk for COMPLEX input formats: TCP/IP, 2/3/4/5G, Bluetooth, Wifi, JPEG, PDF, HTML, Word, …

Recall examples from the fuzzing lecture

Buggy parsing (2)

Buggy – incorrect - parsing can also cause misinterpretation

For example:

- Domain `www.paypal.com\0.mafia.com` in X.509 certificate
- Name `paypal.com,mafia.com` in X.509 certificate
- For which domain is this JDNI loop-up?
  `${jndi:ldap://127.0.0.1#.evilhost.com:1389/a}`

Parser differentials: two applications parse the same data differently, leading to exploitable misunderstandings

High risk for COMPLEX or POORLY SPECIFIED data formats

Unintended parsing

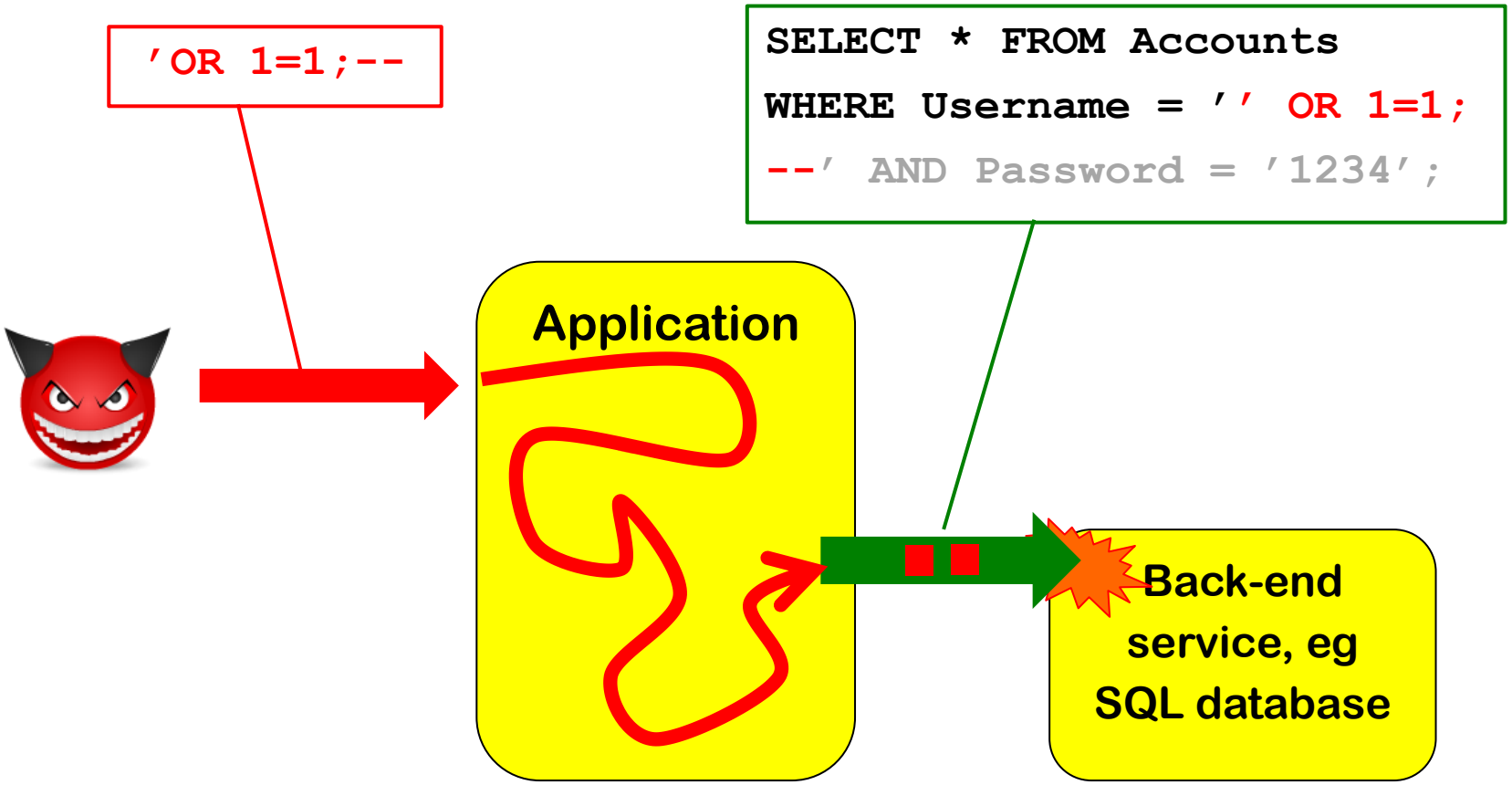
Correct but unintended parsing can also cause security problems, namely injection attacks

Eg parsing (and processing) of user input

- **as SQL command**
- **as file path**
- **as OS command**
- **as HTML or JavaScript**
- **....**

High risk for COMPLEX or EXPRESSIVE data formats/language

Typical injection attack, eg SQLi



Is this an input problem or an output problem?

Injection attacks

General recipe: USER INPUT is combined with other data and forwarded to & processed by some back-end API

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

Tell-tale sign 2: use of STRINGS

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`

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'
```

Blind injection attacks

SQL injection attack with

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

(Lack of) an error response reveals if username starts with 'a'

In a blind injection attack, we're only interested in leakage of information *about* the database, not in the effect of the query on the database (to corrupt data in the database) or the actual response (to leak data from database).

More injection attacks

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

- format string attacks
  - special processing of %n, %s, …
- deserialisation attacks
  - special processing of serialised data representation
- macros: Word & Excel containing Visual Basic (VBA)
  - or other weird Office 'features'!
- PDFs containing malicious JavaScript or ActionScript
- XML bombs & Zip bombs
- SMB relay attacks with bizarre file names
- …

More obscure injection attacks on Microsoft Office

Attackers can trigger RCE in Office without normal Visual Basic 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

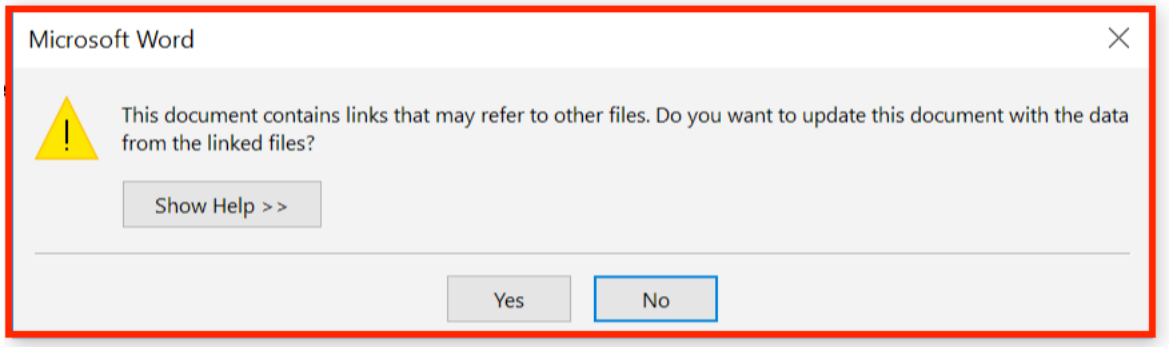
- Excel 4.0 macros
- Archaic legacy features that predate VBA

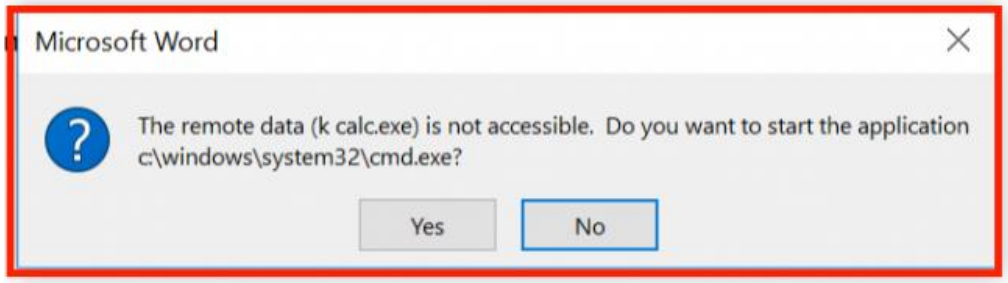
  http://www.irongeek.com/i.php?page=videos/derbycon8/track-3-18-the-ms-office-magic-show-stan-hegt-pieter-ceelen

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

Recall: COMPLEXITY in data formats is bad

DDE warnings in Office





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

SMB relays: Injection attacks via Windows file names

Windows supports *many notations* for file names

- classic MS-DOS notation C:\MyData\file.txt
- file URLs file:///C|/MyData/file.txt
- UNC (Uniform Naming Convention) \\192.1.1.1\MyData\file.txt

which can be combined in fun ways, eg file://///192.1.1.1/MyData/file.txt

Some cause *unexpected behaviour* by involving other *protocols*, eg

- UNC paths to remote servers are handled by SMB protocol;
  SMB sends password hash to remote server to authenticate,
  aka pass the hash

  This can be exploited by SMB relay attacks
  - CVE-2000-0834 in Windows telnet
  - CVE-2008-4037 in Windows XP/Server/Vista
  - CVE-2016-5166 in Chromium
  - CVE-2017-3085 & CVE-2016-4271 in Adobe Flash
  - ZDI-16-395 in Foxit PDF viewer

COMPLEXITY and (unexpected) EXPRESSIVITY is bad

[Example thanks to Björn Ruytenberg, https://blog.bjornweb.nl]

`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**

Why so many & such tricky input problems?

- **Many** input languages and formats

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

- **Complex** input languages and formats

  e.g. 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 misunderstanding to think that input validation and input sanitisation are the best or only defences !

It's an even more 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 May 32nd 2024 or negative amounts

3. Sanitisation: *fix* 'dangerous' inputs

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

Beware: validation & sanitisation are often confused !

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!)

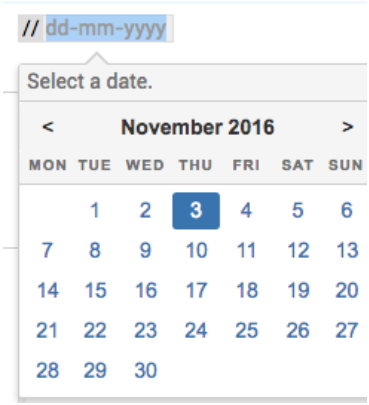
b) Validation

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 or 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
```

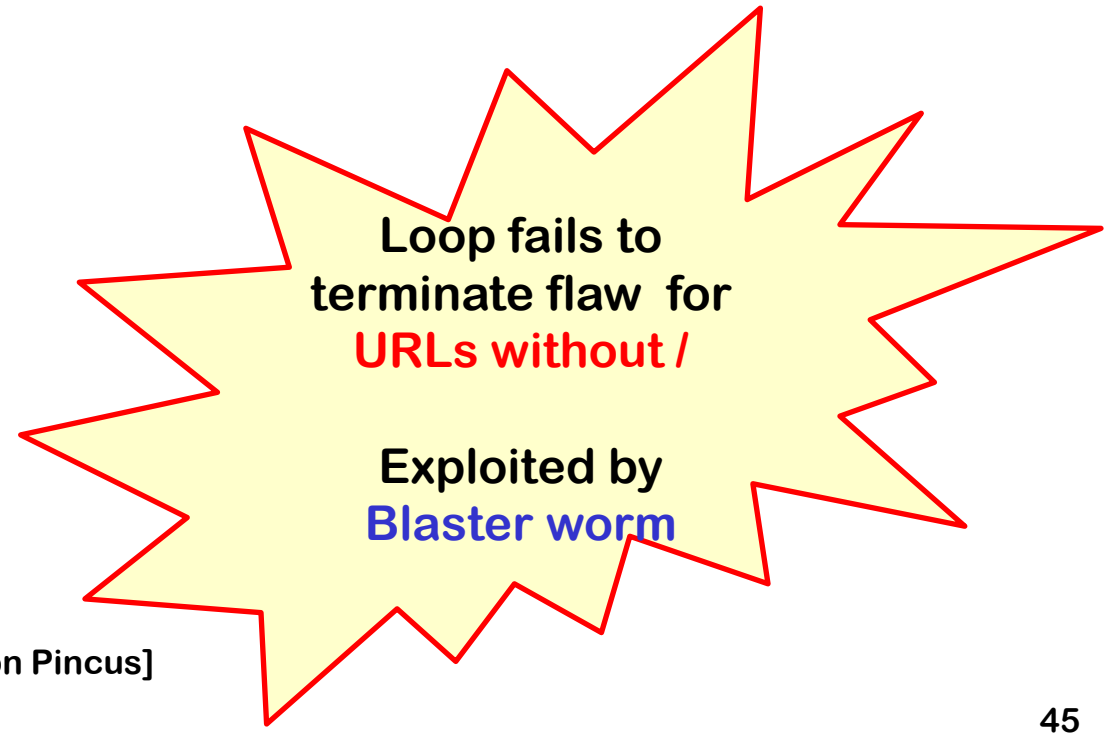
which returns some datatype `URL` (e.g. an object, record, or struct) that comes with relevant operations, eg to extract domain, protocol.

Advantages? 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);
```



[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 language="javascript"> alert('Hi'); </script>` can be injected 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">`

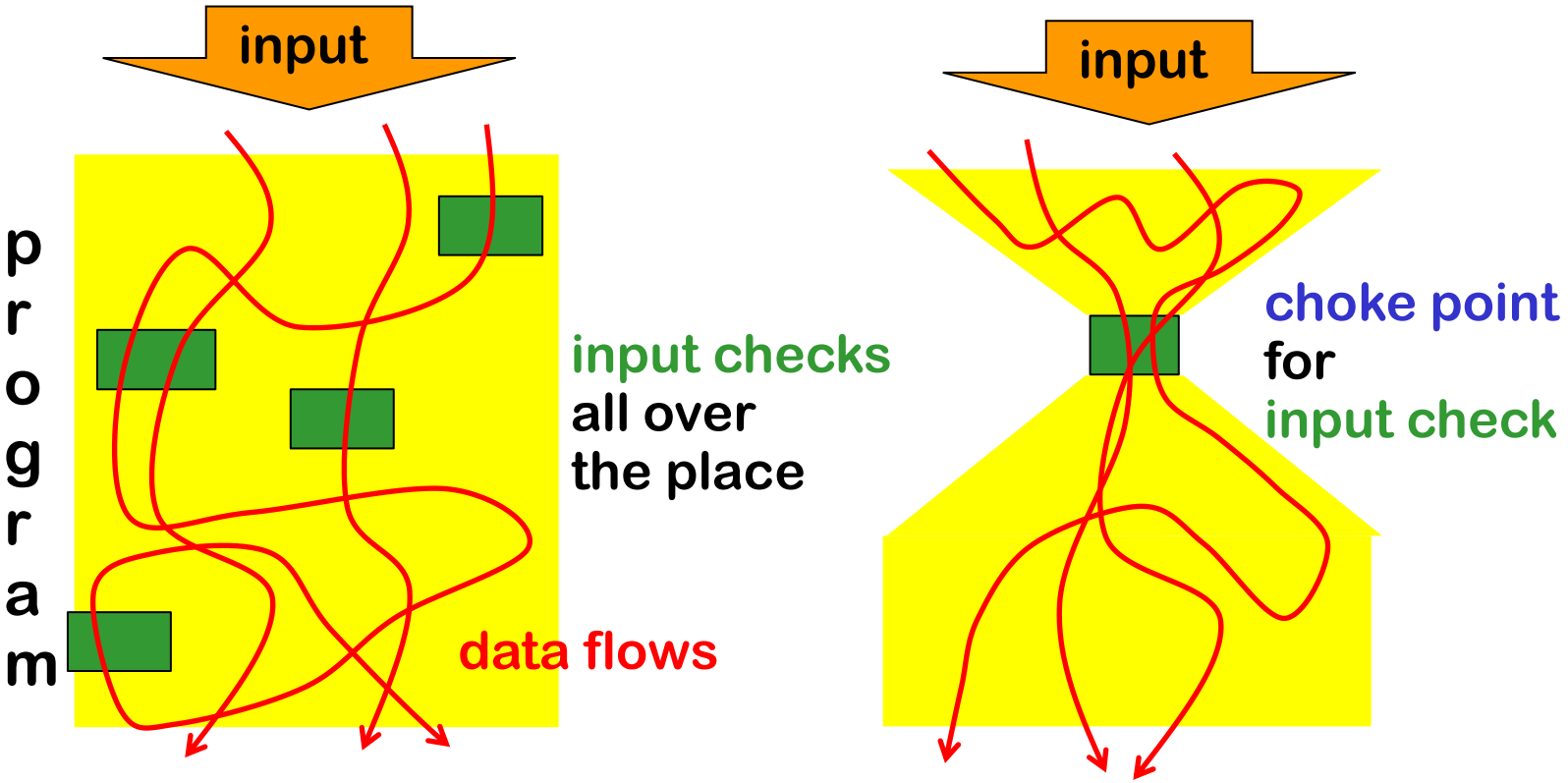
Root cause: COMPLEXITY of HTML format (https://html.spec.whatwg.org)

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**