Some Security by Construction thanks to LangSec

Erik Poll

Digital Security

Radboud University Nijmegen

My personal take on LangSec

Erik Poll

Digital Security

Radboud University Nijmegen

LangSec (Language-Theoretic Security, see langsec.org)

- Interesting look at root causes of large class of security problems, namely problems with input
- Useful suggestions for dos and don'ts

Sergey Bratus & Meredith Patterson
'The science of insecurity'
CCC 2012

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

‘Cybersecurity needs abstract thinking’- Giampaolo Bella

We face a never-ending stream of security vulnerabilities

CVE-1999-0001: remote attacker can cause a denial of service via crafted packets to ip_input.c in BSD-derived TCP/IP implementations

. . .

CVE-2021-44228 aka Log4Shell: remote attacker can execute arbitrary code by abusing JDNI features via LDAP injection in Apache Log4j2 <=2.14.1

These can be funny, interesting, clever, scary … or just depressing

How to see the wood for the trees?

Making sense of all these problems?

One approach: an ever-expanding list of *categories of flaws*

CWE-5: J2EE Misconfiguration: Data Transmission Without Encryption

. . .

CWE-121: Stack-based buffer overflow

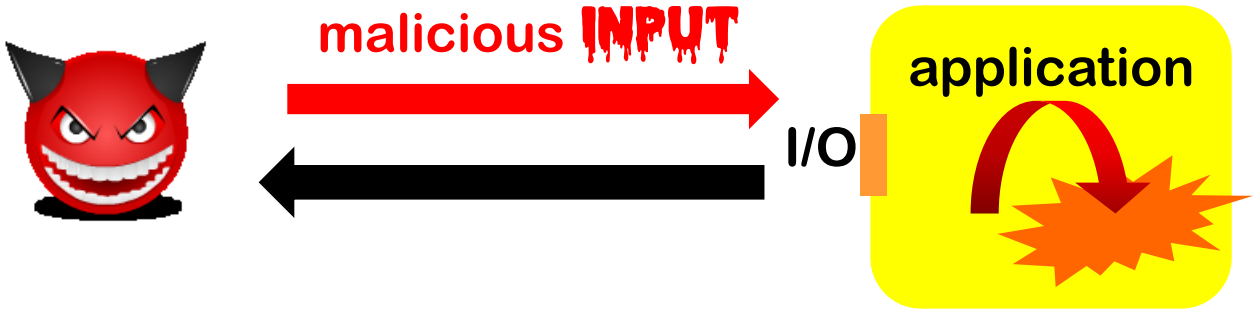
. . .

CWE-1351: Improper Handling of Hardware Behavior in Exceptionally Cold Environments.

The OWASP Top 10 and SANS/CWE Top 25 are useful,

but what about the other 1300+ (!) categories of flaws?

There's only ONE main problem: INPUT



- Aka the I/O attacker model

- Attacker's goal:

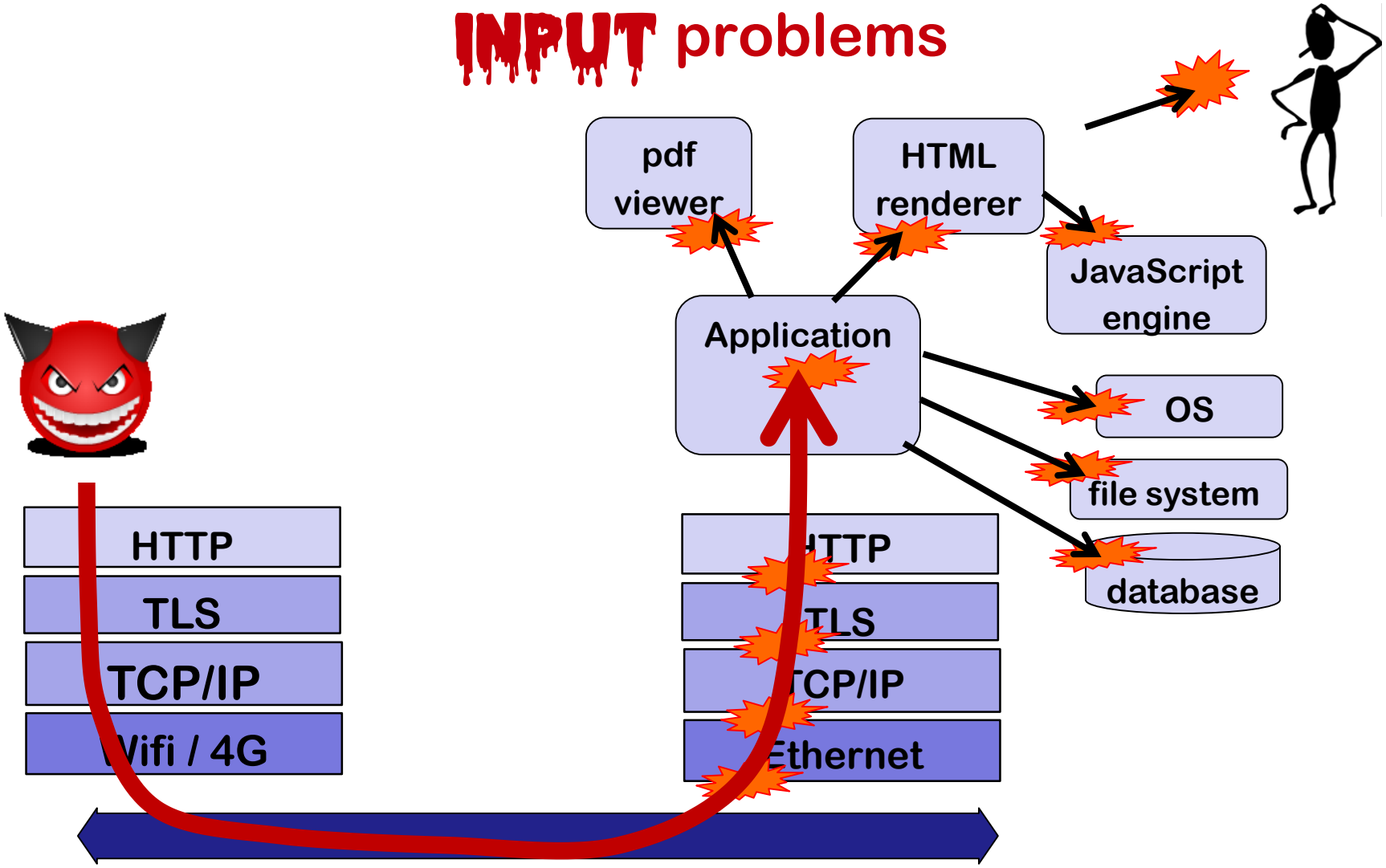
  Compromising integrity and/or availability of the application's behaviour *in any way* (aka 'hacking')

- Garbage In, Garbage Out

  quickly becomes *Malicious* Garbage In, *Security Incident* Out

  aka Garbage In, Evil Out

INPUT problems



Note that input is parsed aka decoded aka interpreted aka deserialised aka … as it moves up the technology stack

LangSec: root causes & remedies

Input languages (aka protocols, file formats, encodings, …) play a central role in causing security flaws In entire protocol stack,

> eg. IPv4/v6, Wifi, Ethernet, Bluetooth, USB, GSM/3G/4G/5G, TLS, SSH, OpenVPN, HTTP(S), X509, HTML5, URLs, email addresses, S/MIME, HTML, XML, XXE, LDAP, JDNI, PDF, JPG, MP3, mpeg, .docx, …

Root causes of security problems here:

- Overly complex, poorly specified, and overly expressive input languages, and too many of them
- Handwritten parser code then results in lots of weird behaviour – bugs or ambiguities - for I/O attacker to have 'fun' with

Remedies:

- simpler languages with clearer (formal!) specs
- generated parser code

Most INPUT problems are PARSING problems

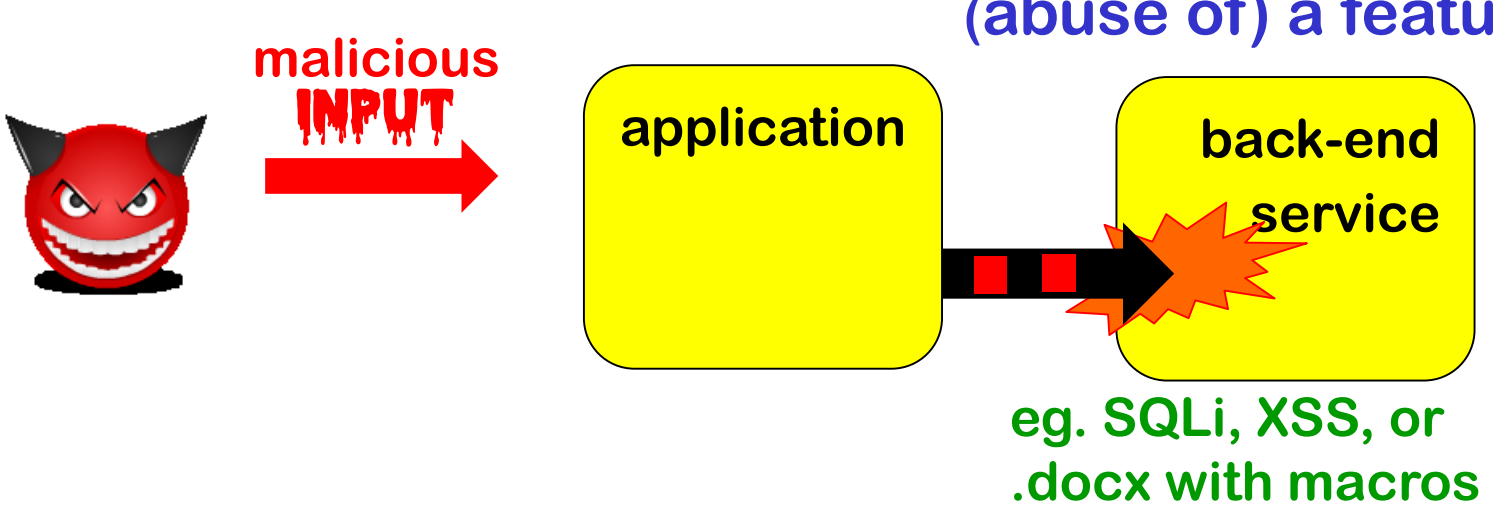
See just about all the CVEs from last week, eg

- CVE-2021-43533 When parsing internationalized domain names, high bits of the characters in the URLs were sometimes stripped, resulting in inconsistencies that could lead to user confusion or attacks such as phishing. This vulnerability affects Firefox < 9
- CVE-2021-44044 An out-of-bounds write vulnerability exists when a JPG file using Open Design Alliance Drawings SDK before 2022.11. The specific issue exists with parsing JPG files. Crafted data in a JPG (4 extraneous bytes before the marker 0xca) can trigger a write operation past the end of an allocated buffer. An attacker can leverage this vulnerability to execute code.

Two types of parsing flaws: processing vs forwarding

1. Processing Flaws

2. Forwarding/Injection Flaws



There are only two main types of INPUT problems

1. Buggy processing & parsing
   - Bug in processing input causes application to go of the rails
   - Classic example: buffer overflow in a PDF viewer, leading to remote code execution

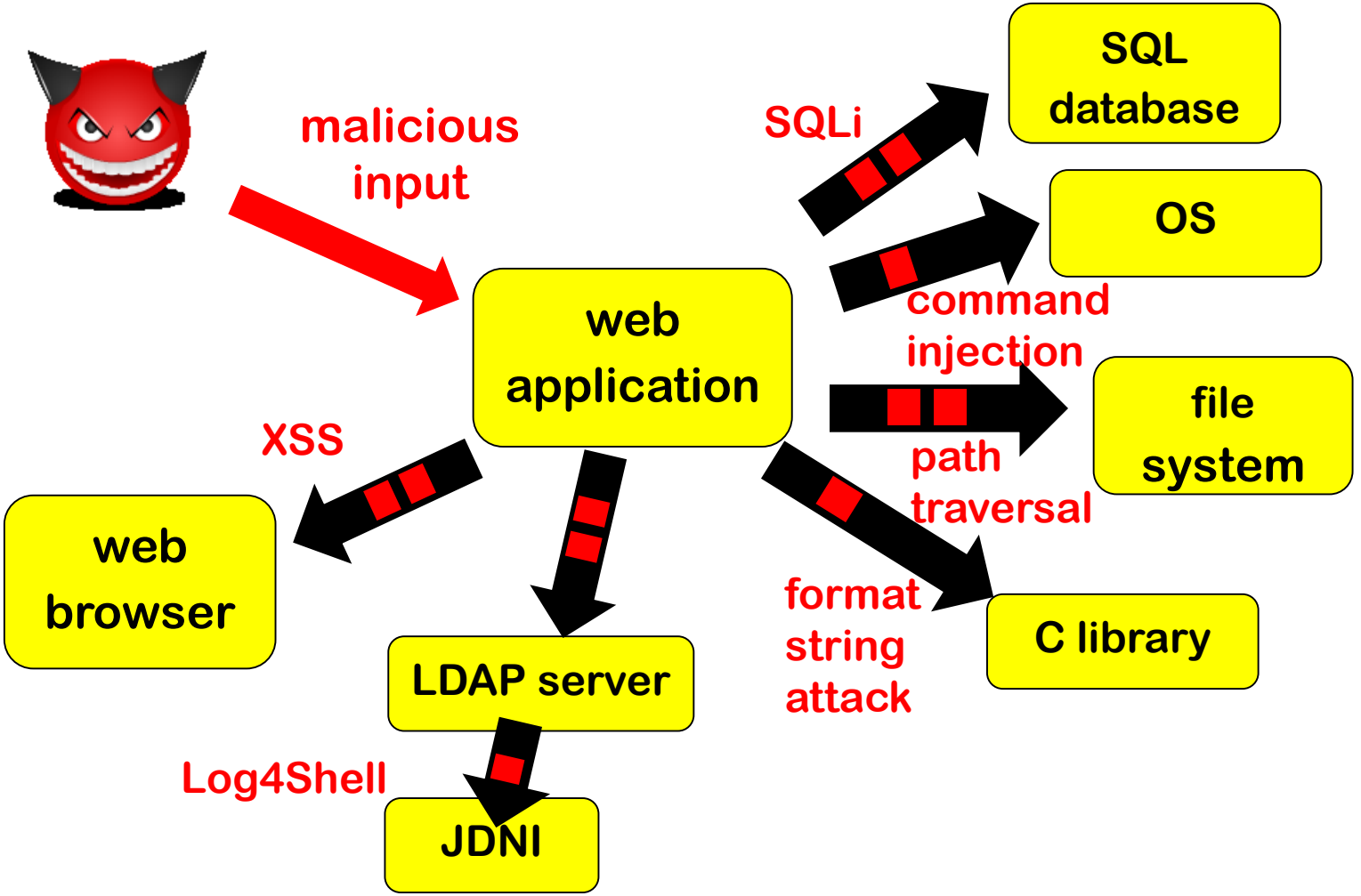
   This is *unintended* behaviour, introduced by *mistake*

2. Flawed forwarding (aka injection attacks)
   - Input is forwarded to *back-end* service/system/API, to cause damage there
   - Classic examples: * injection, XSS, format string attack, Word macros

   This is *intended* behaviour of the back-end, introduced *deliberately*, but *exposed by mistake*

   It is UNWANTED parsing, rather than BUGGY parsing

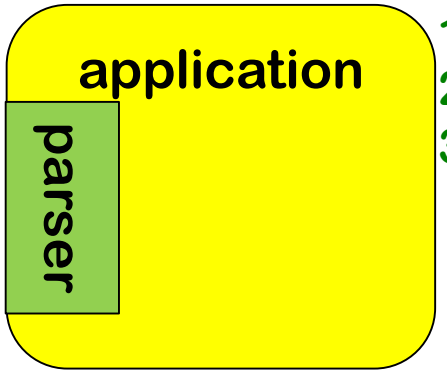
More back-ends, more languages, more problems with unintended parsing …



How & where to tackle input problems?

Tackling processing flaws

malicious input

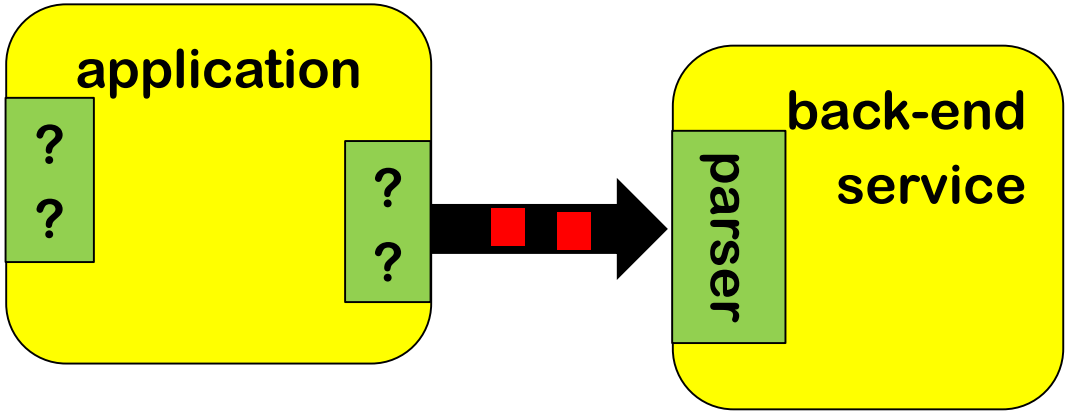


LangSec remedies:

1. Simple & clear language spec;
2. generated parser code;
3. complete parsing before any further processing

Tackling forwarding flaws?

malicious input



validation, sanitisation,
filtering, escaping, encoding?

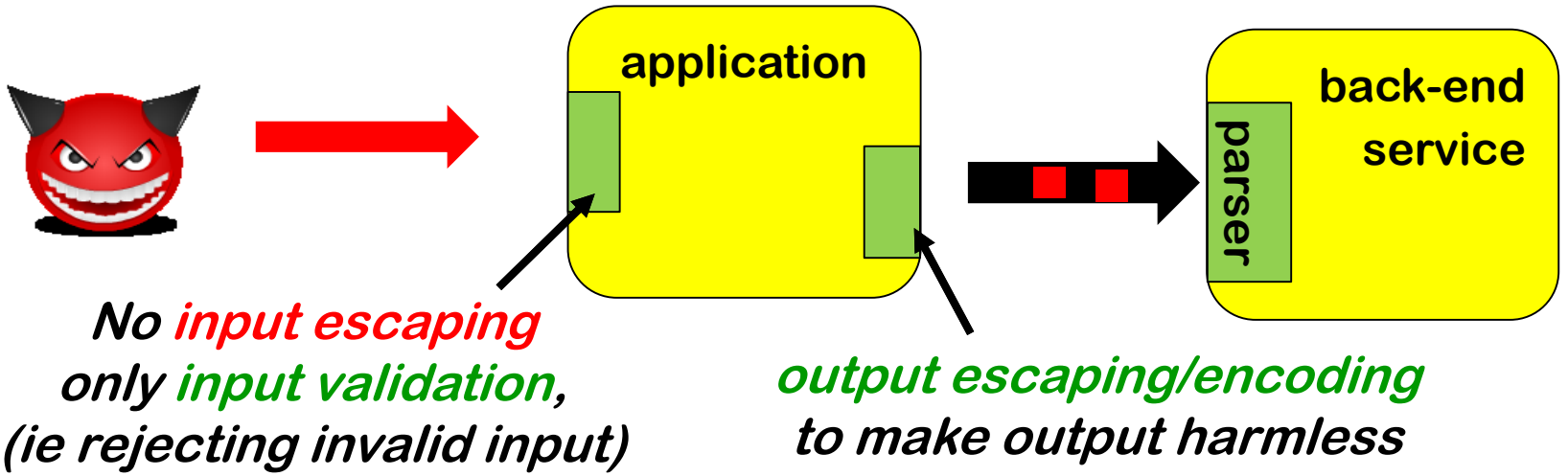
Anti-patterns
in tackling forwarding flaws

[LangSec revisited: input security flaws of the second kind, LangSec'2018]

[Strings considered harmful , USENIX :login;]

Anti-pattern: INPUT ESCAPING

- *Input* escaping, eg. processing *inputs* to escape dangerous meta-characters, is a bad idea
  - at the point of input, the context in which inputs will be used (eg as path name, in SQL query, or as HTML) is unclear, and different contexts require different sanitisations
  - classic anti-example: PHP magicquotes
- *Output* escaping makes more sense, because there context is known



Anti-pattern: STRING CONCATENATION

- Standard recipe for security disaster: concatenating several pieces of data, some of them user input, and passing the result on to some API
  - Classic example: SQL injection
- Note: string concatenation is inverse of parsing

Anti-pattern: STRINGS

More generally, the use of strings in itself is already troublesome

incl. `char*`, `char[]`, `String`, `string`, `StringBuilder`, ...

- Strings are *useful*, because you use them to represent many things:
  eg. name, file name, email address, URL, shell command, snippet of SQL, HTML,...
- This also make strings *dangerous:*

1. Strings are unstructured & unparsed data, and processing often involve some interpretation (incl. parsing)
2. The same string may be handled & interpreted in many – possibly unexpected – ways
3. A string parameter in an API call can – and often does – hide a very expressive & powerful language

Remedies
to tackle forwarding flaws

Recall: Avoiding SQL injection with Prepared Statements

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

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

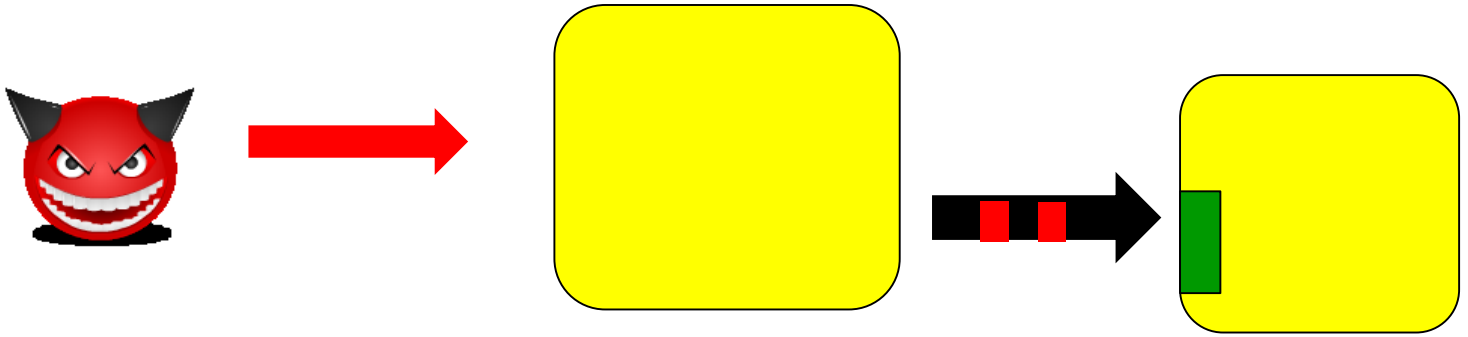
give a string with placeholders and the parameters as separate inputs

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

Recall: Avoiding SQL injection with Prepared Statements

Note that prepared statements (aka parametrised queries)

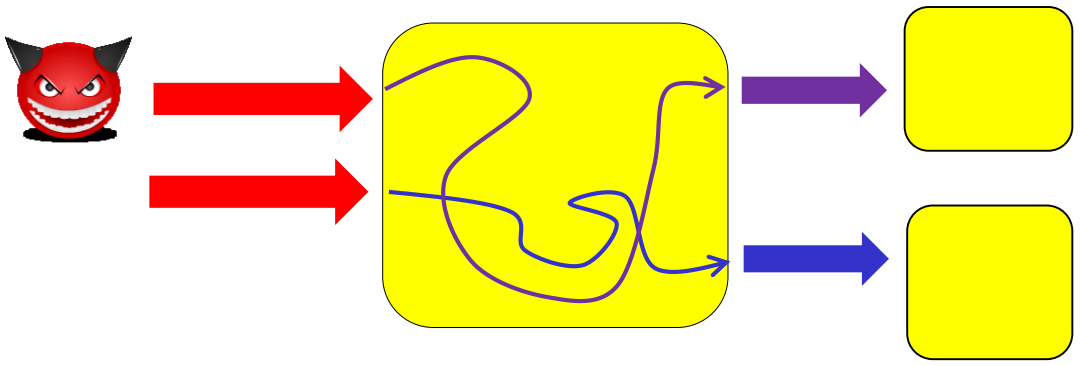
- reduce the expressive power of the interface to the back-end
- avoid unparsing in front-end
- (hence) avoid the need for parsing in the back-end



Remedy: Types (1) to distinguish *languages*

- Instead of using strings for everything,
  use different types to distinguish different kinds of data

  Eg different types for HTML, URLs, file names, user names, paths, …
- Advantages
  - Types provide structured data
  - No ambiguity about the intended use of data

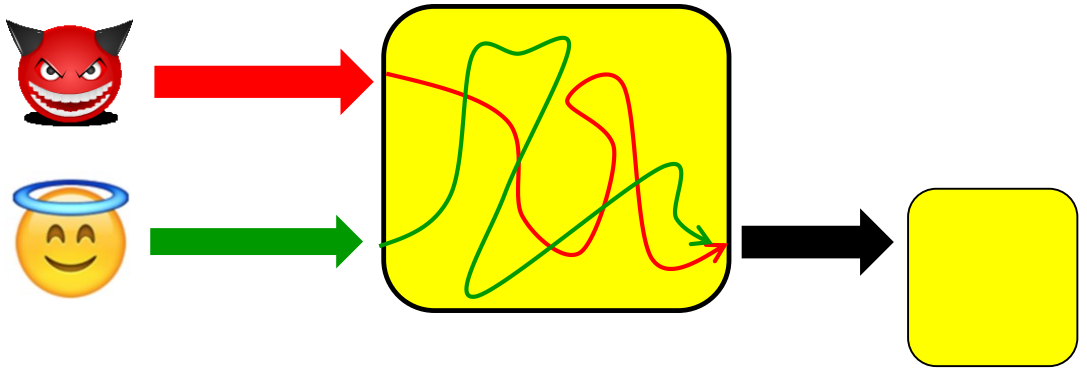


Remedy: Types (2) to distinguish *trust levels*

- Use information flow types to track the origins of data

  and/or to control destinations

  - Ancient idea, going back to [Denning 1976]
  - Eg untrusted user input vs compile-time constants



The two uses of types, to distinguish (1) languages or (2) trust levels, are orthogonal and can be combined.

Example: Trusted Types for DOM Manipulation

DOM-based XSS flaws are proving hard to get rid of, and hard to spot

Trusted Types initiative [https://github.com/WICG/trusted-types]

replaces string-based APIs with typed APIs to structurally root out XSS

- using `TrustedHtml, TrustedUrl, TrustedScriptUrl, TrustedJavaScript,…`
- 'safe' APIs for back-ends that auto-escape untrusted inputs

[Wang et al., If It's Not Secure, It Should Not Compile: Preventing DOM-Based XSS in Large-Scale Web Development with API Hardening, ICSE'2021]

Conclusions

- Most flaws are INPUT flaws and most input flaws involve PARSING
- Distinction between parsing vs forwarding flaws
  - ie. buggy parsing vs unwanted parsing -
  is a useful to analyse input problems
- Most of the LangSec efforts tackle *buggy* parsing,
  but what about *unwanted* parsing? Remedies include:
  - Don't use STRINGS
  - Do use types, to distinguish
    1) different languages, and/or
    2) different trust levels

  Output escaping then becomes safe(r) & sane(r)

These do's are (programming) language-*based* security
as much as (input) language-*theoretic* security (ie LangSec)

Thanks for your attention

Submit your papers to LangSec'21 !