LangSec revisited: input security flaws of the 2\textsuperscript{nd} kind

Erik Poll

Digital Security

Radboud University Nijmegen
Motivations

• Lots of (well-justified!) LangSec efforts to eliminate parser bugs, but *what about input problems that do not involve parser bugs?*

• (How) do existing efforts to tackle such input problems fit in with the LangSec paradigm?
  • Eg efforts at Google to combat XSS

• Can we extend the taxonomy of LangSec anti-patterns & remedies?

Caveats:

• Some answers are obvious, but took me some time to spot

• I’m only connecting some dots I happen to be aware of; there may well be others
(At least) two types of input problems

1. **Buggy processing**
   - Bug in processing input causes application to go of the rails
   - Eg buggy parsing, parser differentials, flaw in program logic
   - 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 example: SQL injection, XSS, Word macros

   This is *intended* behaviour of the back-end, introduced *deliberately*, but *exposed by mistake* by the front-end
Processing vs Forwarding Flaws

Processing Flaws

Forwarding Flaws

Erik Poll
More back-ends, more languages, more problems

- malicious input
- SQLi
- command injection
- LDAP injection
- XSS
- web browser
- web application
- SQL database
- OS
- LDAP server

Erik Poll
Familiar root causes of forwarding flaws

- **Input languages:**
  - too many, overly complex, ill-specified, and overly expressive
  - eg SQL, OS commands, path names, HTML (incl. CSS & javascript), …

- **Parsing:**
  - but unintended parsing, rather than buggy parsing.
  - Some shotgun parsing is unavoidable, as back-end will have to do some parsing
How & where to tackle input problems?

Tackling processing flaws 😊

Simple & clear language spec, generated parser code, complete parsing before any further processing

Tackling forwarding flaws? 😞

Which bits are input?

Where will this input end up?

validation and/or sanitisation (aka encoding aka escaping)?
Anti-patterns
in tackling forwarding flaws
Anti-pattern: **INPUT ESCAPING**

- *Input* escaping, e.g., processing *inputs* to escape dangerous meta-characters, is a bad idea
  - at the point of input, the context in which inputs will be used (e.g., as path name, in SQL query, or as HTML) is unclear, and different contexts require different solutions
  - classic anti-example: PHP magic-quotes
- *Output* escaping makes more sense, because there context is known
  - but there it can be unclear which data originates from input

Erik Poll
Anti-pattern: **STRING CONCATENATION**

- **Recipe for disaster:** *concatenate* several pieces of data, some of them user input, and pass this on to some API
  - Classic example: SQL injection
- **Note:** *string concatenation is inverse of parsing*
- **Forwarding flaws can be** parsing problems, namely if back-end parses data differently than the front-end serialised it
  - but, you can still have forwarding problems *without* any serialisation in the front-end, eg in format string attack like `printf(user_input);`
Anti-pattern: **STRINGS**

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

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

- **Strings are useful**, because you use them to represent many things:
  eg. name, file name, email address, URL, shell command, bit of SQL, HTML,…

- This also make strings **dangerous:**
  
  1. Strings are unstructured & unparsed data, and processing may involve some interpretation
     - If you have a shotgun parser, your code will use strings
  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
Remedy: Parameterised queries

- The best-known & most robust way to tackle SQL injection is to use parameterised queries (or stored procedures)
  - reduces the expressive power of the interface to the back-end
  - avoids unparsing in front-end & (hence) parsing in back-end
- Note: this replaces a generic API call that takes a single `STRING` as argument
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*

- Information flow types can be used to track the origins of data and/or 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 difficult to root out

• as attacks using script gadgets demonstrate
  [Lekies et al., Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets, CCS’17]

Trusted Types initiative [https://github.com/WICG/trusted-types] replaces string-based APIs with typed APIs

• using TrustedHtml, TrustedUrl, TrustedScriptUrl, TrustedJavaScript,…
• ‘safe’ APIs for back-ends that auto-escape untrusted inputs

[Sebastian Lekies’ talk at OWASP Benelux 2017: Don’t trust the DOM]
[Christoph Kern, Securing the Tangled Web, CACM 2014]
Beyond types: extending programming language

**Wyvern** programming language by Jonathan Aldrich et al. allows domain-specific extensions, eg

```javascript
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 ‘built-in’ types of the programming language

Added advantage over types: more convenient syntax

Conclusions

- **Forwarding flaws vs processing flaws** is a useful taxonomy to analyse input problems & LangSec solutions.

- 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)

- Or even extend the programming language for this

These do’s are (programming) **language-based** security as much as (input) **language-theoretic** security.

*Are there more forwarding anti-patterns & remedies, or more good examples of these?*
Thanks for your attention