LangSec revisited: 
input security flaws of the second kind

Erik Poll
erikpoll@cs.ru.nl
Digital Security group, Radboud University, Nijmegen, the Netherlands

Abstract—We consider a simple classification of input flaws in two categories: (1) flaws in processing input, with buffer overflows in parsers as the classic example and (2) flaws in forwarding input to some other system, aka injection flaws, with SQL injection and XSS as classic examples. The LangSec paradigm identifies common root causes for both categories of flaws, but much of the LangSec literature and efforts focus on the first category of flaws, esp. on techniques to eliminate parser bugs. Therefore we take a look at some existing approaches to tackling the second category of flaws, to identify (anti)patterns and place these in the LangSec perspective.

I. INTRODUCTION

LangSec gives excellent insights in the security problems in input handling that plague our software: into the root causes behind these problems, anti-patterns that are likely to result in security flaws, and patterns that can help to prevent them.

Broadly speaking, two categories of input problems can be distinguished: processing flaws and forwarding flaws (aka injection attacks). Some of same root causes are at play but some of ways forward to tackle these two categories of flaws are different. Much of the LangSec literature and efforts focus on the first category, with the aim to root out parser bugs and parser differentials.

To redress the balance, this paper considers the second category and looks at ideas and (anti)patterns in tackling injection flaws. One approach here is to address the problem at the level of the programming language, as proposed in Wyvern [1], with a so-called type-specific programming language that offers native support for input and output formats handled by programs. Another approach is the ongoing evolution in mechanisms to tackle XSS in applications [2]. For the sake of completeness, but at the risk of boring some readers, we also include infamous anti-patterns such as PHP magicquotes and dynamic SQL queries built using string concatenation.

The underlying (anti)patterns we observe also come up in software security design guidelines [3] and fit very naturally in the LangSec paradigm, as they confirm once again the central role of input languages and the associated parsing in input security flaws. Moreover, the simple classification we use and the (anti)patterns we observe suggest an extension (or refinement?) of the taxonomy of LangSec errors proposed by Momot et al. [4] and additions to the list of remedies to expunge them.

II. PROCESSING VS FORWARDING FLAWS

In a typical attack on an application, the attacker crafts some malicious input that causes the software to go off the rails, with all sorts of nasty consequences. When faced with a creative attacker

‘Garbage In, Garbage Out’ quickly descends into

‘Malicious Garbage In, Security Incident Out’.

We can distinguish two kinds of flaws in input handling:

**Buggy processing.** As LangSec points out, many input problems arise due to buggy parsing of input. Classic examples here are buffer overflows in parsers for complex input formats such as Flash or PDF. The program containing this buggy parser then provides some weird behaviour – a weird machine, in LangSec terminology [5] – when fed malformed input (or sometimes even when fed correctly formed input), and the attacker tries to (ab)use this weird functionality in interesting ways.

Buffer overflows and other memory-related bugs make up for the large share of these attacks, but any kind of logical flaw in the parsing or subsequent processing of input, may provide weird functionality for an attacker to exploit. Differences between parsers for the same language, so-called parser differentials [6], can also provide wriggling room for an attacker.

Note that the weird functionality that the attacker abuses here has typically been introduced by accident in the code.

**Careless forwarding.** In other input attacks, the problem is not so much buggy processing of input, but rather careless forwarding of input to some external system or back-end service or API, so that malicious input can propagate to do damage there. Classic examples are SQL injection, command injection, path traversal, and XSS (Cross Site Scripting). These flaws are also known as injection flaws\(^1\). We prefer the term forwarding flaws because in some sense all input attacks are injection attacks; the forwarding aspect is what sets these input attacks apart from the others.

The external system or service that is abused could be a separate application, some OS service, or an internal API for

---

\(^1\)The definition of injection flaws used in the OWASP Top 10 2017 [7], where injection flaws occupy the number 1 spot, and have done for many years, excludes XSS. The importance of scripting on the web, and hence the extra difficulties in rooting out XSS compared to say SQL injection, justify XSS getting its own spot in the OWASP Top 10, but it is fundamentally just another injection flaw like the others.
a component inside the application, but that does not make any difference for most of the discussion in this paper.

Forwarding attacks do not rely on any parser bugs: the back-end service, say the SQL database, parses and processes its inputs correctly. Of course, there could also be parser bugs in this service for an attacker to abuse, but we ignore that possibility not to confuse the discussion. So the problem is not that this functionality is buggy, but rather that it is exposed to attacker, without proper constraints. Consequently, the weird machine that the attacker can abuse with forwarding attacks is often not quite so weird, as it provides normal functionality of say a SQL database or the underlying OS. The attacker abuses functionality has been introduced deliberately, but that is exposed accidentally.

An interesting class of flaws to compare the two categories are attacks with Microsoft Office documents that contain malicious macros. This is a (deservedly!) popular form of attack and frequently used in phishing campaigns. It is different from many other input attacks in that it requires a human user to click on some attachment, which is probably why it is (undeservedly!) missing in many lists of standard input attacks. However, it is just another forwarding attacks: a Word document with a PowerShell macro is just another way of doing OS command injection.

Attackers can also exploit parser flaws in phishing attacks, e.g. using malicious PDF attachments that exploit some buffer overflow in the parser of a PDF viewer. But that has the disadvantage of depending on a specific flaw in a specific PDF viewer and it is typically harder to craft payloads to exploit buffer overflows than to write macros: so exploiting a feature of Microsoft Office can be much more attractive than exploiting a bug in a PDF viewer.

III. COMMON ROOT CAUSES

Some of the same root causes highlighted by LangSec are at play in both parsing and forwarding attacks. One root cause is the expressivity of input languages used by back-end services. E.g. one can question the wisdom in having such a powerful feature as macros in a document input format, and indeed LangSec warns us about the expressive power of input languages. A second root cause is the sheer number of such languages, which may include SQL, OS commands, path names, LDAP, XML, \ldots .

IV. COMMON ANTI-PATTERN: SHOTGUN PARSING

The well-known LangSec anti-pattern of shotgun parsing is present in forwarding flaws, as noted in [4]: some of the parsing is not done in the main application but in the external service it relies on. However, it is not so clear that this anti-pattern is really avoidable here: after all, the back-end service is meant to process some data, and doing some parsing for that may be unavoidable.

V. INPUT OR OUTPUT PROBLEM?

A fundamental complication with forwarding flaws is that they involve two systems -- the front-end application and a back-end service -- and that they involve both input and output, as the input language of the back-end is the output language of the front-end. This raises the question of who is to blame, and who can or should prevent the problem: is the application at fault for being careless in invoking the back-end service, or the back-end service at fault for expressing a too powerful interface? It also introduces a well-known dilemma in where and how do to do validation of user input, esp. when it comes to sanitisation, discussed in more detail below.

VI. ANTI-PATTERN: INPUT SANITISATION

There are different kinds of operations that can be done as part of input validation. A validation routine can simply filter out the invalid inputs from valid ones, rejecting the invalid ones, but it can also try to sanitise data, also called encoding or escaping. The typical example is escaping dangerous characters that have a special meaning in the back-end, to prevent forwarding flaws.

To explicitly distinguish these two aspects, the first can be called filtering and the second sanitisation, but beware that the terms input validation and input sanitisation are often treated as synonyms.

A complication with forwarding flaws is that ideally one would like validate input at the point where the input enters the application, because at that program point it is clear that it is untrusted input. However, at that point you may not yet know in which context the input will be used, and different contexts will may require different forms of escaping. E.g. the same input string could be used in a path name, a URL, an SQL query, and a piece of HTML text, and these contexts may need different encodings.

Because escaping is context-sensitive, it is well known that using one generic operation to sanitise all input is highly suspect, as one generic operation is never going to provide the right escaping for a variety of different back-end systems. Moreover, doing input sanitisation, i.e. sanitisation at the point of input rather than at the point of output, is suspect, as the context typically is not known there.

The classic example here is the PHP magicquotes setting, which caused all incoming data to be automatically escaped (by pre-pending certain characters with a backslash). It took a while for people to come to the agreement that this was a bad idea: magicquotes were depreciated in PHP 5.3 and finally removed in PHP 5.4 in 2012\(^2\).

\(^2\)This has led to countermeasures, such as opening untrusted documents in a protected mode with macros disabled, aka ‘Protected View’, but a bit more social engineering can typically easily overcome that.

\(^3\)Typically this escaping uses a black-list approach rather than a white-list approach; this is known to be more dangerous, but we’ll ignore that in this paper.

\(^4\)Canonicalisation is a third aspect of validation, and an important one, but we ignore it here, as it is not relevant to our discussion.

VII. ANTI-PATTERN: STRING CONCATENATION FOR DYNAMIC QUERIES

Another well-known anti-pattern in forwarding attacks is the use of string concatenation to combine user input with other strings to construct a parameter that is fed to some API call, as is done in dynamic SQL queries.

Given that LangSec highlights the importance of parsing, it is interesting to note that string concatenation is a form of unparsing. Indeed, the whole problem with forwarding attacks is that the back-end service may parse query strings in a different way that intended.

An early effort investigating the essence of injection attacks proposed a runtime countermeasure which traces user input as it propagates through an application to then detects if it corrupts the way queries are parsed [8]. Here a query is deemed to be corrupted if the shape of the resulting parse tree has changed. This uses a negative security model: it aims to identify and stopping unsafe cases.

Of course, the better way to prevent SQL injection is to use parameterised queries. This uses a positive security model: it tries to prevent unsafe SQL calls, and at compile time, rather than weeding them out at runtime.

The use of parameterised queries reduces the expressive power of the interface to the back-end database and reduces the amount of runtime parsing. So clearly this mechanism involves key aspects highlighted in the LangSec paradigm, namely expressivity and parsing.

VIII. PATTERN: PROGRAMMING LANGUAGE SUPPORT FOR INPUT/OUTPUT LANGUAGES

Even if programmers are aware that the benefits of prepared statements, they might still use dynamic SQL queries because of the convenience. The programming language Wyvern, dubbed a type-specific language, proposed a more structural way to rule out forwarding problems, by natively embedding different (input or output) languages inside the programming language. The aim here is to provide a safe mechanism where the program always handles structured data rather than strings, but a way that is just as convenient for the programmer as using strings. Having a native embedding helps here by allowing notations that are just as convenient for the programmer as for strings. The idea is that type-specific programming language does not provide ad-hoc support for one output language like SQL, but allows any number of languages to be embedded. In the original use case, web programming, these embedded languages would include both SQL and HTML. These different languages then show up as different types in the programming languages.

IX. PATTERN: SECURITY TYPES FOR OUTPUT SANITISATION

Even if automatic input sanitisation built into the programming language construct with a global setting like magicquotes cannot work, in some circumstances automated output sanitisation can be made to work. A fundamental problem with sanitising outgoing data is that at that program point where data is passed on as output, it is typically not clear if this data stems from user input, but type systems can be used to remedy this.

An example where this approach has been successfully used is in web templating frameworks, where mechanisms have been added to existing frameworks to automatically perform sanitisation in a context-sensitive manner [11]. The approach uses type qualifiers, which for instance distinguish string constants from unsanitised input variables, so that type inference traces which sanitisations have been performed and, given a specific context in which a variable is used, decide which additional sanatisations need to be inserted.

The approach was demonstrated to be practicable with an implementation for Google’s Closure Templates.

This approach has since evolved to a wider systematic approach to combat XSS at Google [2]. In addition to the automatic sanitisation in template engines, the approach relies on safe APIs that acts as a wrapper around original API that suffers from injection problems. Security types play a central role in the approach. For example, it relies on a type SafeHtml for strings (or string-like objects) that will not cause untrusted script execution when evaluated as HTML in a browser, and which are therefore safe to use as HTML or as HTML parameter in calls to the DOM APIs. Only a limited set of constructions can be used to construct elements of this type, which guarantee the soundness of the assumptions captured by the type.

The ongoing struggle against XSS attacks is by no means finished. The latest forms of DOM-based XSS attacks using script gadgets [13] highlight the fundamental difficulties in rooting out XSS via the DOM. A recent proposal “Trusted Types for DOM Manipulation” aims to replace all string-based APIs of the DOM with typed APIs, in an effort to get rid of DOM-based XSS.

X. ANTI-PATTERN: STRINGS CONSIDERED HARMFUL

One recurring anti-pattern – the elephant in the room – is the use of strings. There are several reasons why the use of strings can lead to problems, and heavy use of strings is a sign of trouble:

- Strings can be used for all sorts of data: email addresses, file names, URLs, fragments of HTML, pieces of JavaScript, etc. This makes it a very useful and ubiquitous data type, but it the downside is that using the same type for different kind of data can cause confusion: from the type we cannot tell what the intended use of the data is, or indeed whether it has been validated.
- Strings are by definition unparsed data. So if a program uses strings, it will typically have to do parsing at runtime, incl. parsing that could have been avoided if

---

6or stored procedures, though these are not always safe, as discussed on https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet
7https://github.com/WICG/trusted-types. Also presented at OWASP Benelux 2017 [14].
more structured forms of data were used instead. The extra parsing creates a lot of room for trouble, esp. in combination with the point above, which means that the same string might end up in different parsers.

- **String parameters in interfaces often bring unwanted expressivity.** Interfaces that take strings as parameter often – implicitly or explicitly – introduce a whole new language, with all sorts of expressive power that may not be necessary, and which only provides a security risk.

In summary, the problem with strings is that they use one generic data type, for completely unstructured data, and for many kinds of data, hiding the fact that there are many different languages involved, possibly very expressive ones, each with their own interpretation.

(For these disadvantages it does not matter if the strings we use are type-safe, memory-safe and immutable String objects in a language such as Java; string objects in C++, without these nice guarantees; or C byte arrays or char pointers, which are even more error-prone. Of course, the more safety guarantees we can get from our programming language, and the less error-prone the data type or its, the better.)

**XI. CONCLUSION**

The distinction between processing flaws and forwarding flaws is a very natural and obvious one when considering security problems in input handling, but we have not aware of this distinction having been discussed from a LangSec perspective. The LangSec view is useful for both categories of flaws: input languages play a central role in both; there are common root causes, namely the large number of input languages and the expressivity of these languages; and shotgun parsers appear as anti-pattern for both, even if for forwarding flaws this anti-pattern is harder to avoid.

Many of the ways forward suggested by the LangSec paradigm focus on eradicating parser bugs (e.g. insisting on clear specifications of input languages, keeping these languages simple, generating parsers from formal specs instead of hand-rolling written parser code, and separating parsing and subsequent processing in an attempt to avoid shotgun parsers). However, these techniques are not sufficient to root out forwarding flaws. Even if we can get rid of all parser bugs we can still have forwarding flaws, and some form of shotgun parsing seems unavoidable with forwarding flaws.

Fortunately, there are ideas for ways forward, which already appear in the literature and in practice. Important recommendations here, which we feel deserve to be added to the list of LangSec remedies listed in [4], are

- avoid using strings
- use types offered by the programming language, not only to distinguish different input languages (e.g. distinguishing HTML from SQL) but also to distinguish different trust assumptions about the data (e.g. distinguishing tainted user input from sanitised values or constants).

Some the (anti)patterns we discussed also show up in the security design flaws discussed by Arce et al. [3], more in particular under ‘Strictly separate data and control instructions, and never process control instructions received from untrusted sources’ and ‘Define an approach that ensures all data are explicitly validated’. Both Wyvern and Google approach to combatting XSS can be seen as ways to put these design principles in practice, at the level of the programming language, or at least its type system. Note that these approaches belong to the paradigm of language-based security as much as to the paradigm of language-theoretic security. (Beware of the possible confusion here: in the term language-based security, the word ‘language’ refers to programming languages, whereas in the term language-theoretic security, it refers to input languages.)

The (anti-)patterns for injection flaws that we observe are related to familiar LangSec themes of parsing and the expressive power of input languages: the remedies try to reduce expressive power, try to reduce the potential for confusion and mistakes in (un)parsing, or try to avoid (un)parsing altogether.

**REFERENCES**


