Strings Considered Harmful

ERIK POLL

Erik Poll is Associate Professor at Radboud University Nijmegen. His research focuses on the use of formal methods to analyze the security of systems, especially of the software involved. Application areas that provided case studies for his research include smart cards, security protocols, payment systems, and smart grids. erikpoll@cs.ru.nl

Buggy parsers are an important source of security vulnerabilities in software: many attacks use malicious inputs designed to exploit parser bugs. Some security flaws in input handling do not exploit parser bugs, but exploit correct—albeit unexpected—parsing of inputs caused by the forwarding of inputs between systems or components. This article, based on an earlier workshop paper [11], discusses anti-patterns and remedies for this type of flaw, including the anti-pattern mentioned in the title.

LangSec and Parsing Flaws

The LangSec paradigm [3, 8] gives good insights into the root causes behind the majority of security problems in software, which are problems in handling inputs. It recognizes that the input languages used play a central role. More particularly, it identifies the following root causes for security problems: the sheer number of input languages that a typical application handles; their complexity; their expressivity; the lack of clear, unambiguous specifications of these languages; and the handwritten parser code (which often mixes the parsing and subsequent processing, in so-called shotgun parsers, where input is parsed piecemeal and in various stages scattered throughout the code). All this leads to parser bugs, with buffer overflows in processing file formats such as Flash or network packets for protocols such as TLS as classic examples. It can also lead to differences between parsers that can be exploited, with, for example, variations in interpreting X509 certificates [6] as a result. In all cases, these bugs provide weird behavior—a so-called weird machine, in LangSec terminology—that attackers can try to abuse.

Much of the LangSec research therefore concentrates on preventing parsing flaws: by having simpler input languages; by having clearer, formal specs for them; and by generating parser code to replace handwritten parsers, using tools such as Hammer (https://github.com/UpstandingHackers/hammer), Nail [2], or protocol buffers (https://developers.google.com/protocol-buffers). For a more thorough discussion of LangSec anti-patterns and remedies, see [8].

Forwarding Flaws

However, not all input-related security flaws are due to buggy parsing. A large class of flaws involves the careless *forwarding* of malicious input by some front-end application to some back-end service or component where the input is correctly—but unexpectedly and unintentionally—parsed and processed (Figure 1). Classic examples are format string attacks, SQL injection, command injection, path traversal, and XSS.

In the case of a SQL injection attack, the web server is the front end and SQL database is the back end. In the case of a format string attack, the back end is not a separate system like a database but consists of the C system libraries. In an XSS attack, the web browser is the back end and the web server the front end; this can get more complex, e.g., in reflected XSS attacks, where malicious input is forwarded back and forth between browser and server before finally doing damage in the browser.

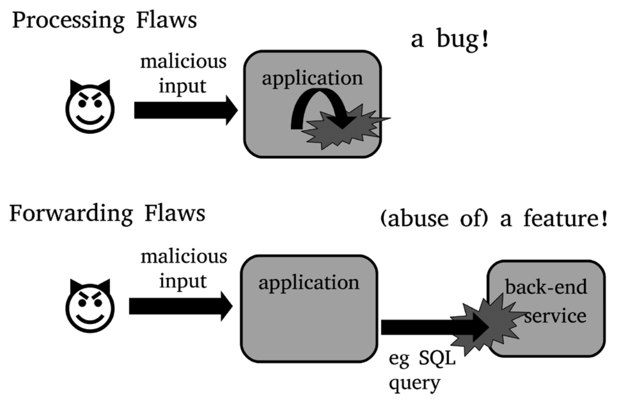

Figure 1: Processing vs. forwarding flaws

Forwarding attacks do not (necessarily) exploit parser bugs: the back-end service, say the SQL database, may well parse and process its inputs correctly. The problem is not that this SQL functionality is buggy but rather that it can be triggered by attackers that feed malicious input to the front end. Unlike attacks that exploit parsing bugs, where attackers abuse weird behavior introduced accidentally, attackers here abuse functionality that has been introduced deliberately but which is exposed accidentally.

Forwarding flaws are also called *injection flaws*, e.g., in the OWASP Top Ten, where they occupy the first spot. 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.

Input or Output Problem?

Forwarding flaws involve two systems—a front-end application and a back-end service—and both input and output, since the malicious input to the front end ultimately ends up as output from the front end to the back end. This not only introduces the question of *how* to tackle this problem but also the question of *where* to tackle it. Should the front end prevent untrusted input from ending up in the back end, and if so, should it sanitize data at the program point where the data is output to the back end, or should it do that earlier, at the program point where it received the original malicious input? Or should the back end simply not provide such a dangerously powerful interface to the front end? We can recognize anti-patterns that can lead to forwarding flaws, or to bad solutions in tackling them, as well as some remedies to address them in a structural way.

Anti-Pattern: Input Sanitization

There are very different ways to treat invalid or dangerous input. It can be completely *rejected* or it can be *sanitized*. Sanitization can be done by escaping or encoding dangerous characters to make them harmless, typically by adding backslashes or quotes, or by stripping dangerous characters and keywords. A complication here is that ideally one would like to validate input at the point where the input enters an application, because at that program point it is clear whether such input is untrusted or not. However, at that point we may not yet know in which context the input will be used, and different contexts may require different forms of escaping. For example, the same input string could be used in a path name, a URL, an SQL query, and in HTML text, and these contexts may require different forms of escaping.

Because escaping is context-sensitive in this way, it is well known that using one generic operation to sanitize all input is highly suspect, as one generic operation is never going to provide the right escaping for a variety of back-end systems. This also means that *input* sanitization, i.e., sanitization at the point of input rather than at the point of output, is suspect since the context typically is not known there.

The classic example here is the infamous PHP magic quotes setting, which caused all incoming data to be automatically escaped. It took a while to reach consensus that this was a bad idea: magic quotes were deprecated in PHP 5.3.0 and finally removed in PHP 5.4.0 in 2012.

Anti-Pattern: String Concatenation

A well-known anti-pattern in forwarding attacks is the use of string concatenation. Concatenating several pieces of data, some of which are user input, and feeding the result to an API call, as is done in dynamic SQL queries, is the classic recipe for disaster.

Given that the LangSec approach highlights the importance of parsing, it is interesting to note that string concatenation is a form of *unparsing*. Indeed, the whole problem in forwarding attacks is that the back-end service parses strings in a different way than the front end intended.

Anti-Pattern: Strings

We would argue that a more general anti-pattern than the use of string concatenation for dynamic queries is the use of strings at all. There are several reasons why heavy use of strings can spell trouble:

- *Strings can be used for all sorts of data*: usernames, email addresses, file names, URLs, fragments of HTML, pieces of JavaScript, etc. This makes it a very useful and ubiquitous data type, but it also causes confusion: from a generic string type, we cannot tell what the intended use of the data is or, for instance, whether it has been escaped or validated.
- *Strings are by definition unparsed data*. So if a program uses strings, it typically has to do parsing at runtime. Much of this parsing could be avoided if more structured forms of data were used instead. The extra parsing creates a lot of room for trouble, especially in combination with the point above, which tells us that the same string might end up in different parsers.

The shotgun parsing that the LangSec literature warns against, where partial and piecemeal parsing is spread throughout an application, also inevitably involves the use of strings, namely for passing around unparsed fragments of input.

- *String parameters often bring unwanted expressivity.* Interfaces that take strings as a parameter often introduce a whole new language (e.g., HTML, SQL, the language of pathnames, OS shell commands, or format strings), 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 it is one generic data type, for completely unstructured data, and for many kinds of data, obscuring the fact that there are many different languages involved, possibly very expressive ones, each with its own interpretation. Of course, others have warned about the use of strings before, e.g., [1].

The disadvantages above apply equally to *char* pointers in C, *string* objects in C++, or *String* objects in Java. Of course, for security it is better to use memory-safe, type-safe, or immutable and hence thread-safe data types rather than more error-prone versions.

Remedy: Reducing Expressive Power

An obvious way to prevent forwarding flaws, or at least mitigate the potential impact, is to reduce the expressive power exposed by the interface between the front end and the back end.

For SQL injections this can be done with parameterized queries (or with stored procedures, provided that these are safe). The use of parameterized queries reduces the expressive power of the interface to the back-end database, and it reduces the amount of runtime parsing. So clearly this mechanism involves key aspects highlighted in the LangSec approach, namely expressivity and parsing.

Remedy: Types to Distinguish Languages and Formats

Different types in the programming language can be used to distinguish the different languages or data formats that an application handles. These types reduce ambiguity: ambiguity about the intended use of data and ambiguity about whether or not it has been parsed and validated. This then also reduces the scope for unintended interactions.

Note that standard security flaws such as double decoding bugs or problems with null terminator characters in strings also indicate confusion about data representations that use of a type system could—and should—prevent.

For example, an application could use different types for URLs, usernames, email addresses, file names, and fragments of HTML. The type checker can then complain when a username is included inside HTML and force the programmer to add an escaping function to turn a username into something that is safe to render as HTML.

For data that is really just a string, like a username, one might use a struct or object with a single string field. (Type annotations, as exist in Java for example, could also be used to distinguish different kinds of strings [10].) However, for structured data, say a URL, the type would ideally not just be a wrapper for the unparsed string but, instead, an object or struct with fields and/or methods for the different components, such as the protocol, domain, path, etc., to reduce the amount of code that handles data in unparsed form.

When data is forwarded between components inside an application or between applications written in the same programming language, data can be forwarded "as is," with all type information preserved and without the need for any (un)parsing. However, when data is exchanged with external systems, it may have to be serialized and deserialized. Here the risk of parsing bugs re-emerges, and the classic LangSec strategies to avoid these should be followed by, ideally, generating the code for (de) serialization from a formal spec.

Remedy: Types to Distinguish Trust Levels

Types can also be used for different *trust levels*. This then allows information flows from untrusted sources in the code to be traced and restricted. An example would be to use different types for trusted string constants hard coded in the application and for untrusted (aka tainted) strings that stem from user input to then only allow the former to be used as parameters to certain security-sensitive operations.

Efforts at Google to prevent XSS in web applications [7] use types in this way (https://github.com/google/safe-html-types/blob/master/doc/index.md). For instance, it uses different types to distinguish

- URLs that can be used in HTML documents or as arguments to DOM APIs, but not in contexts where this would lead to the referred resource being executed as code, and
- more trusted URLs that can also be used to fetch JavaScript code (e.g., by using them as `scr` of a `script` element).

A more recent proposal to combat XSS, called Trusted Types (https://github.com/WICG/trusted-types), extends Google's approach to fighting XSS using types by replacing all string-based APIs of the DOM with typed APIs. This approach tackles the root cause that makes it so hard to deal with the more complicated forms of (DOM-based) XSS: the ubiquitous use of string parameters in the DOM APIs.

The two ways to use types—to distinguish different kinds of data or different trust levels—are of course orthogonal and can be combined. Using trust levels for security goes back to work on information flow in the 1970s [4]. It has been used in many static and dynamic analyses over the years, including many security type systems and source code analyzers, and has given rise to a whole research field of language-based information-flow security [12].

Clearly, the notion of information flow goes to the heart of what forwarding flaws are about. A type system for information flow is precisely what can solve the fundamental problem of keeping track of whether data has been or should be validated or sanitized. Instead of just tracking untrusted data to prevent malicious input from being forwarded to places where it can do damage, type systems for information flow can also be used to track confidential information to prevent information leaks (see, e.g., [5]).

Beyond Types: Programming Language Support

Instead of using the type system of a programming language to distinguish the different languages and data formats that an application has to handle, one can go one step further and provide native support for them in the programming language. This approach is taken in Wyvern [9], called a type-specific programming language by the designers.

An added advantage is that the programming language can provide more convenient syntax to tempt programmers away from convenient but insecure coding styles. For example, it can provide syntax for safe parameterized SQL queries that is just as convenient as the unsafe dynamic SQL queries, with the nice infix notation for string concatenation that programmers like. The idea is that a type-specific programming language allows any number of input and output languages to be embedded. In the original use case of web programming, the embedded languages would include SQL and HTML. These languages then show up as different types in the programming languages, with all the convenient syntax support.

Conclusion

Many of the remedies 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 handrolling written parser code, and separating parsing and subsequent processing in an attempt to avoid shotgun parsers.

However, these remedies are not sufficient to root out forwarding flaws, which can exist even if our code does not contain any parser bugs. Fortunately, there *are* remedies to tackle forwarding flaws, as discussed above, which already appear in the literature and in practice:

- Using more structured forms of data than strings
- Using types, not only to distinguish different languages and formats that are manipulated (e.g., distinguishing HTML from SQL), but also to distinguish different trust assumptions about the data (e.g., distinguishing untrusted user input from sanitized values or constants)

The (anti-)patterns we discussed all center around the familiar LangSec themes of parsing and the expressive power of input languages; the remedies try to reduce expressive power, reduce the potential for confusion and mistakes in (un)parsing, or avoid (un)parsing altogether.

References

[1] I. Arce, K. Clark-Fisher, N. Daswani, J. DelGrosso, D. Dhillon, C. Kern, T. Kohno, C. Landwehr, G. McGraw, B. Schoenfield, M. Seltzer, D. Spinellis, I. Tarandach, and J. West, "Avoiding the Top 10 Software Security Design Flaws," Technical Report, IEEE Computer Society Center for Secure Design (CSD), 2014.

[2] J. Bangert and N. Zeldovich, "Nail: A Practical Tool for Parsing and Generating Data Formats," *;login:*, vol. 40, no. 1 (USENIX, 2015), pp. 24–30: https://www.usenix.org/system/files/login/articles/login_feb15_06_bangert.pdf.

[3] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, "Exploit Programming: From Buffer Overflows to Weird Machines and Theory of Computation," *;login:*, vol. 36, no. 6 (USENIX, 2011), pp. 13–21: https://www.usenix.org/system/files/login/articles/105516-Bratus.pdf.

[4] D. E. Denning and P. J. Denning, "Certification of Programs for Secure Information Flow," *Communications of the ACM*, vol. 20, no. 7, 1977, pp. 504–513: https://www.cs.utexas.edu/~shmat/courses/cs380s/denning.pdf.

[5] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. B. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu, "Collaborative Verification of Information Flow for a High-Assurance App Store," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS '14)*, pp. 1092–1104: https://homes.cs.washington.edu/~mernst/pubs/infoflow-ccs2014.pdf.

[6] D. Kaminsky, M .L. Patterson, and L. Sassaman, "PKI Layer Cake: New Collision Attacks against the Global X.509 Infrastructure," in *Financial Cryptography and Data Security*, vol. 6054 of LNCS (Springer, 2010), pp. 289–303: https://www.esat.kuleuven.be/cosic/publications/article-1432.pdf.

[7] C. Kern, "Securing the Tangled Web," *Communications of the ACM*, vol. 57, no. 9, 2014, pp. 38–47.

[8] F. Momot, S. Bratus, S. M. Hallberg, and M. L. Patterson, "The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them," in *Proceedings of the IEEE Conference on Cybersecurity Development (SecDev '16)*, pp. 45–52: http://langsec.org/papers/langsec-cwes-secdev2016.pdf.

[9] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich, "Safely Composable Type-Specific Languages," in *ECOOP 2014—Object-Oriented Programming*, vol. 8586 of LNCS (Springer, 2014), pp. 105–130: http://www.cs.cmu.edu/~aldrich/papers/ecoop14-tsls.pdf.

[10] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst, "Practical Pluggable Types for Java," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*, pp. 201–212: https://homes.cs.washington.edu/~mernst/pubs/pluggable-checkers-issta2008.pdf.

[11] E. Poll, "LangSec Revisited: Input Security Flaws of the Second Kind," in *Proceedings of the IEEE Symposium on Security and Privacy Workshops*, 2018, pp. 329–334: http://spw18.langsec.org/papers/Poll-Flaws-of-second-kind.pdf.

[12] A. Sabelfeld and A. C. Myers, "Language-Based Information-Flow Security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, 2003, pp. 5–19: https://www.cs.cornell.edu/andru/papers/jsac/sm-jsac03.pdf.