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 approach identifies common root causes for both categories of flaws, but much of the LangSec literature and efforts focus on the first category of flaws, especially 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

The LangSec approach 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 remedies 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. Most of the efforts inspired by the LangSec approach, and indeed nearly all the work presented at the annual LangSec workshop, 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 existing ideas and (anti)patterns in tackling injection flaws.

This paper does not present any new results or implementation efforts. It is more of an attempt at Systemisation of Knowledge (SoK). The category of flaws we consider is hardly new; indeed, injection flaws go back to phone phreaking in the 1950s. Some of the patterns in tackling these flaws are also ancient; for instance, the use of types for information flow goes back to the 1970s [1]. For the sake of completeness, but at the risk of boring some readers, we also include infamous anti-patterns such as PHP's `magic_quotes` and the by now established countermeasure of parameterised queries.

Less familiar to a wider audience might be the countermeasures proposed in programming language design, notably in Wyvern [2], and in the ongoing efforts to root out XSS (especially DOM-based XSS) at Google with improved language-support and APIs [3], [4]. One motivation for writing this paper was the observation that these approaches fit very neatly in the language-theoretic view on the root causes of the security flaws, if you take a wider view to consider not just parsing bugs also forwarding flaws. Another motivation was the observation that many of the anti-patterns that cause forwarding flaws and remedies to prevent them are missing in the taxonomy of LangSec errors and remedies by Momot et al. [5]. Hopefully this paper can provide a step towards extending this taxonomy to also cover forwarding flaws, to provide a more comprehensive account of how we can tackle input flaws.

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 we are faced with a creative attacker,

'Garbage In, Garbage Out'

quickly descends into

'Garbage In, Evil Out' [6].

Here we can distinguish (at least) two kinds of flaws in input handling, as discussed below.

A. Buggy processing.

Many input problems arise from 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 [7] – when it is fed malformed input (or sometimes even when fed correctly formed input) and the attacker can try to (ab)use this weird functionality in interesting ways.

Buffer overflows and other memory-related bugs make up a 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 [8], can also provide wriggling room for an attacker.

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

B. 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 collectively 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 of some 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 for now to not confuse the discussion.) So the problem is not that this functionality is buggy, but rather that it is exposed to attackers, without proper constraints. Consequently, the weird machine that attackers 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 attackers abuse functionality has been *introduced deliberately*, but that is *exposed accidentally*.

Attacks due to *in-band signalling*, with Blue Boxes for phone phreaking as the classic example, are also injection attacks. But these are (generally) not 'forwarding' attacks, as they do not involve any forwarding of the malicious input to some external service; instead, the interface of the service being abused is directly accessible to the attacker. So one could argue that forwarding attacks are only a subset of all injection attacks.

C. Example: malicious email attachments

An interesting type of attack to compare the two categories of input problems above are phishing campaigns where attackers add a malicious attachment to emails. These attacks are different from other input attacks in that they require a human user to click on the attachment, which is probably why they are (undeservedly) missing in some lists of standard input attacks.

Microsoft Office documents with malicious macros are a deservedly popular choice here for attackers to use. 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. Note that such an attack is just another forwarding attack: a Word document with a PowerShell macro is just another way of doing OS command injection.

Attackers can also exploit parser bugs 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. Moreover, it is typically harder to craft payloads to exploit buffer overflows than it is to write macros: so exploiting a *feature* of Microsoft Office can be much more attractive than exploiting a *bug* in a PDF viewer.

[1]The definition of injection flaws used in the OWASP Top 10 [9], where injection flaws occupy the number 1 spot, and have done for many years, excludes XSS. The importance of scripting on the web, and 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.

D. Common root causes

Some of the same root causes 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 the LangSec literature warns 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, . . . , which creates a large attack surface.

E. Input or output problem?

A fundamental complication with a forwarding flaw is that it involves two systems – the front-end application and a back-end service – and that it involves both input and output, as the input language of the back-end is the output language of the front-end. In a SQL injection attack on a web application, the web server the front-end and SQL database is the back-end. In a XSS attack, the web browser is the back-end and the web server the front-end. (To make matters more complicated, in reflected XSS attacks the web browser is also acts as front-end to the server, namely in the first step of the attack.)

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 is the back-end service at fault for expressing a too powerful interface? Rather than a matter of blame for either party it is more a matter of not understanding the ramifications of a design choice in the interface between them: if one chooses to use a very powerful generic interface here, say for arbitrary SQL queries, then it is the responsibility of the front-end to ensure that malicious inputs cannot subvert queries to express something beyond what was intended.

The fact that there is a front-end and a back-end also introduces a well-known dilemma in *where* do to do input validation, especially when it comes to sanitisation, discussed in more detail in Section III-B below.

III. Anti-patterns

Several anti-patterns can be recognised that cause or contribute to forwarding flaws.

A. Anti-pattern: shotgun parsing

The well-known LangSec anti-pattern of shotgun parsing is present in forwarding flaws, as noted in [5]: some of the parsing is not done in the main application but in the external back-end that 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.

B. 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 *escaping* or *encoding*. The typical example is escaping dangerous characters that have a special meaning in the back-end, by adding backslashes or quotes, to prevent forwarding flaws.

To explicitly distinguish these two options, 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[2].

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 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 forms of escaping.

Because escaping is context-sensitive in this way, 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 infamous PHP `magic_quotes` 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: `magic_quotes` were depreciated in PHP 5.3.0 and finally removed in PHP 5.4.0 in 2012[3].

C. 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 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 with forwarding attacks is that the back-end service may parse query strings in a different way than 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 detect if it corrupts the way queries are parsed [10]. 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, as discussed in Section IV-A. Note that this uses a *positive* security model: it tries to prevent unsafe SQL calls, and at compile time, rather than weeding them out at runtime.

D. Anti-pattern: Strings considered harmful

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 the use of strings can lead to problems and heavy use of strings can be 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 the downside is that using the same type for different kinds 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 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.
- *String parameters in interfaces often bring unwanted expressivity.* Interfaces that take strings as parameter often – implicitly or explicitly – introduce a whole new language (e.g. HTML, SQL, the language of path names or of OS shell commands), with all sorts of expressive power that may not be necessary, and which only provides a security risk.

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

The Top 10 Security Software Design Flaws by Arce et al. [11] also warn about the use of strings as an anti-pattern. However, there the warning is more narrowly focused on the use of strings in APIs if these strings mingle data and control information – i.e. the case discussed in the last bullet point above. We would go one step further and argue that using structured types instead of strings is preferable *everywhere*.

(For the disadvantages above 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 and `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, the better.)

IV. Remedies

Measures to structurally avoid forwarding flaws can be taken at the level of API design, at the level of the type

[2]Canonicalisation is a third aspect of validation, and an important one, but we ignore it here, as it is not relevant to our discussion.

[3]See http://php.net/manual/en/security.magicquotes.php.

system, or – more ambitiously – at level of the programming language. Below we discuss the remedies we are aware of; there may well be more, or more interesting examples of them that deserve to be mentioned here. Many of the remedies involve avoiding the use of strings.

A. 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 parameterised queries (or with stored procedures, provided that these are safe[4]). The use of parameterised 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.

This pattern also comes up in the Top 10 Security Software Design Flaws proposed by Arce et al. [11], namely as the principle to 'strictly separate data and control instructions' in the design of APIs where strings are used as parameters.

B. Remedy: Using types to distinguish different languages

Types in the programming language (or more generally, different forms of structured data) can be used to distinguish the different input and output languages – or formats – that an application has to handle. This reduces ambiguity, both about the intended use of data and about whether or not it has been parsed and validated. It also reduces the scope for unintended interactions. For example, different types could be used to distinguish fragments of HTML from other string-like data, or to distinguish remote URLs and file URLs. The expressivity and flexibility of the type system (e.g. support for subtyping or polymorphism) may limit what is practical here.

C. Remedy: Using types for trust levels

Types (or so-called type qualifiers [12]) 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 here is to use different types for trusted string constants hard-coded in the application and untrusted aka tainted strings that stem from user input.

The use of different trust levels for security goes back to Denning's seminal work on information flow [1]. It has been used in many static and dynamic analyses over the years, incl. many security type systems and source code analysers. Within the field of language-based security [13], it has given rise to a whole sub-field of language-based information-flow security [14].

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 complication with forwarding flaws discussed in Section II-E, as it can track whether data has been or should be validated or sanitised. So the type system can enforce the security design principles to 'never process control instructions received from untrusted sources' and 'define an approach that ensures all data are explicitly validated' [11].

[4]As discussed on https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

Many language extensions to support some form of information flow have been proposed. A well-known example is Jif[5], which extends Java with information flow types. A more recent example is SPARTA [15], which uses the Checker framework[6] for adding pluggable type systems to Java [16], an approach enabled by the added expressivity in Java's type system since Java 7 by annotations on types.

Many approaches to support information flow target Java or Java-like languages, but not all do: e.g. Microsoft's SAL[7] (Standard Annotation Language) provides annotations to add information flow information to C/C++ code, as the SPARK/Ada approach does for Ada [17].

Of course, the two ways to use types – to distinguish different kinds of data or different trust levels – are orthogonal and can be combined, as discussed in the example in Section IV-E below.

D. Beyond types: Programming language support

Instead of using the type system of a programming language to distinguish the different input and output languages that an application has to handle, one can go one step further and provide native support for these languages in the programming language. This approach is taken in the programming language Wyvern[8] [18], [19], called a *type-specific language* by the designers.

Even if programmers are aware that the benefits of prepared statements, they may still use dynamic SQL queries because of the convenience. One of the design goals of Wyvern is to provide a safe alternative that is just as convenient as the unsafe dynamic SQL queries. By natively embedding the input and output formats in the programming language is possible to provide a safe mechanism where the program always handles structured data rather than strings, but in a way that is just as convenient for the programmer as using strings: the native embedding allows all the notational convenience and syntactic sugar that the programmer is used to (e.g. simple infix notation for concatenation).

The idea is that a 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, the embedded languages would include SQL and HTML. These two languages then show up as different types in the programming languages, with all the convenient syntax support.

[5]https://www.cs.cornell.edu/jif
[6]https://checkerframework.org
[7]https://msdn.microsoft.com/library/ms182032.aspx
[8]https://github.com/wyvernlang/wyvern

E. Example: Security types for web applications

The ongoing efforts to prevent forwarding flaws in web applications at Google, which have resulted in a recent proposal for 'Trusted Types for DOM Manipulation' [4], [20], provide interesting examples of the use of types for both aspects discussed above.

Even if automatic *input* sanitisation built into the programming language with a construct like PHP's `magic_quotes` cannot work, in some circumstances automated *output* sanitisation can be made to work, by making use of type information. An example where this approach has been successfully used is in web templating frameworks, where existing frameworks have been adapted to automatically perform sanitisation in a context-sensitive manner [21]. The approach was demonstrated to be practicable with an implementation for Google's Closure Templates[9]. It uses type qualifiers [12], which for instance distinguish string constants from unsanitised input variables, so that type inference can trace which sanitisations have been performed and, given a specific context in which a variable is used, decide which additional sanitisations need to be inserted.

This approach has since evolved to a wider approach to systematically combat XSS [3]. In addition to automatic sanitisation in the template engine, the approach relies on inherently safe APIs that acts as a wrapper around original APIs that suffers from injection problems. Security types that distinguish different formats and trust levels play a central role here. For example, it uses a type `SafeHtml` for strings that will not cause untrusted script execution when evaluated as HTML in a browser, and that 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 guarantees 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 [22] highlight the fundamental difficulties in rooting out XSS. (Of course, script gadgets are an excellent example of yet another weird machine.) The recent proposal 'Trusted Types for DOM Manipulation' [4] aims to replace all string-based APIs of the DOM with typed APIs in an effort to get rid of DOM-based XSS. So this takes the pattern to get rid of strings even further.

V. 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 are not aware of this distinction having been discussed from a LangSec perspective before. The LangSec view is useful for both categories of flaws: 1) input languages play a central role in both; 2) there are common root causes, namely the large number of input languages and the expressivity of these languages; and 3) shotgun parsers appear as anti-pattern for both, even if for forwarding flaws this anti-pattern seems harder to avoid.

[9]https://developers.google.com/closure/templates

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 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, there may still be forwarding flaws, and some form of shotgun parsing seems unavoidable with forwarding flaws.

Fortunately, there are ideas to remedy this, which already appear in the literature and in practice. Important remedies here, which we feel deserve to be added to those listed in [5], are

- avoid using strings, and use more structured forms of data instead;
- use types, 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 and constants).

The anti-patterns and remedies discussed are not new and are related to established security design patterns for software [11]. The Wyvern programming language [2], Google's approach to combatting XSS [3], as well as efforts to support information flow in Jif [23] all put these remedies into practice. One can argue that these approaches belong to the paradigm of language-*based* security as much as to the paradigm of language-*theoretic* security. (Beware of possible confusion here: in the term language-based security, the word 'language' refers to the *programming* language, whereas in the term language-theoretic security, it refers to the *input* languages.)

The anti-patterns and remedies for forwarding flaws we observed all centre around the 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.

Acknowledgement

Thanks to the reviewers for their comments and providing concrete suggestions to structure this paper.

References

[1] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, Jul. 1977.

[2] D. Kurilova, C. Omar, L. Nistor, B. Chung, A. Potanin, and J. Aldrich, "Type-specific languages to fight injection attacks," in *Symposium and Bootcamp on the Science of Security (HotSos'14)*. ACM, 2014.

[3] C. Kern, "Securing the tangled web," *Communications of the ACM*, vol. 57, no. 9, pp. 38–47, 2014.

[4] "Trusted Types for DOM Manipulation," 2017, see https://github.com/WICG/trusted-types. The rationale and design goals are discussed on https://discourse.wicg.io/t/proposal-trusted-types-for-dom-manipulation/2360.

[5] 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 *Cybersecurity Development (SecDev)*. IEEE, 2016, pp. 45–52.

[6] D. McIlroy, "Buy/by the book or bye-bye the game," in *Third workshop on Language-Theoretic Security (LangSec'16)*, 2016, keynote talk. Available at http://spw16.langsec.org/papers.html.

[7] 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:*, pp. 13–21, 2011.

[8] D. Kaminsky, M. L. Patterson, and L. Sassaman, "PKI layer cake: New collision attacks against the global X.509 infrastructure," in *International Conference on Financial Cryptography and Data Security*, ser. LNCS, vol. 6054. Springer, 2010, pp. 289–303.

[9] OWASP, "OWASP Top 10 List," 2017, available at https://www.owasp.org/index.php/Category:OWASP_Top_Ten_2017_Project.

[10] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *POPL'06*. ACM, 2006, pp. 372–382.

[11] 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," IEEE Computer Society Center for Secure Design (CSD), Tech. Rep., 2014.

[12] J. S. Foster, T. Terauchi, and A. Aiken, "Flow-sensitive type qualifiers," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*. ACM, 2002, pp. 1–12.

[13] D. Kozen, "Language-based security," in *Mathematical Foundations of Computer Science (MFCS'99)*, ser. LNCS, vol. 1672. Springer, 1999, pp. 284–298.

[14] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan. 2003.

[15] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. B. Barros, R. Bhoraskar, S. Han *et al.*, "Collaborative verification of information flow for a high-assurance app store," in *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2014, pp. 1092–1104.

[16] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst, "Practical pluggable types for Java," in *International Symposium on Software Testing and Analysis (ISSTA'18)*. ACM, 2008, pp. 201–212.

[17] R. Chapman and A. Hilton, "Enforcing security and safety models with an information flow analysis tool," in *ACM SIGAda Ada Letters*, vol. 24, no. 4. ACM, 2004, pp. 39–46.

[18] D. Kurilova, A. Potanin, and J. Aldrich, "Wyvern: Impacting software security via programming language design," in *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM, 2014, pp. 57–58.

[19] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich, "Safely composable type-specific languages," in *European Conference on Object-Oriented Programming (ECOOP'14)*, ser. LNCS, vol. 8586. Springer, 2014, pp. 105–130.

[20] S. Lekies, "Don't trust the DOM: Bypassing XSS mitigations via script gadgets," 2017, presentation at OWASP BeNeLux. Slides and video available at https://www.owasp.org/index.php/OWASP_BeNeLux-Day_2017.

[21] M. Samuel, P. Saxena, and D. Song, "Context-sensitive auto-sanitization in web templating languages using type qualifiers," in *Computer and Communications Security (CCS'11)*. ACM, 2011, pp. 587–600.

[22] S. Lekies, K. Kotowicz, S. Groß, E. A. Vela Nava, and M. Johns, "Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets," in *ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. ACM, 2017, pp. 1709–1723.

[23] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng, "Secure web applications via automatic partitioning," in *ACM SIGOPS Operating Systems Review (SOPS'07)*, vol. 41, no. 6. ACM, 2007, pp. 31–44.