Wyvern: Impacting Software Security via Programming Language Design

Darya Kurilova
Carnegie Mellon University
darya@cs.cmu.edu

Alex Potanin
Victoria University of Wellington
alex@ecs.vuw.ac.nz

Jonathan Aldrich
Carnegie Mellon University
aldrich@cs.cmu.edu

Abstract

Breaches of software security affect millions of people, and therefore it is crucial to strive for more secure software systems. However, the effect of programming language design on software security is not easily measured or studied. In the absence of scientific insight, opinions range from those that claim that programming language design has no effect on security of the system, to those that believe that programming language design is the only way to provide "high-assurance software." In this paper, we discuss how programming language design can impact software security by looking at a specific example: the Wyvern programming language. We report on how the design of the Wyvern programming language leverages security principles, together with hypotheses about how usability impacts security, in order to prevent command injection attacks. Furthermore, we discuss what security principles we considered in Wyvern's design.

1. Introduction

Software systems have become an integral part of our lives, and are responsible for managing important and confidential information, such as bank accounts, medical records, and the content of private communications. Having given such responsibilities to software systems, it is important to ensure that they can reliably keep the information secure. One would expect programming languages, as the primary tool of the programmer, to be a factor in software security, yet the extent by which programming languages can impact a software systems' security is a controversial topic.

Some suggest that designing a programming language to provide security is futile, arguing that system safety and security is a "completely programming language agnostic topic" [2, 7]. We disagree with this opinion, and argue that programming language design can have a direct influence on the security of software systems. To support our position, we describe how the design of the Wyvern programming language addresses a specific type of security vulnerability—command injections—and what hypotheses led us to make the design decisions we did.

2. Command Injection Defense in Wyvern

Wyvern [4] is a web programming language that aims to address the OWASP's Top 10 most critical web application security risks [6] and being *secure by default*. Wyvern is able to prevent command injection vulnerabilities [3], and we now illustrate how the design of the Wyvern programming language is able to do it by looking specifically at SQL injections.

Our investigation begins with a question: why are command injection vulnerabilities introduced? After all, preventing them is easy: developers simply need to use prepared SQL statements. However, developers often do not. We formed the following usability hypothesis as to why:

H1: *Command injection vulnerabilities are often introduced because it is easier to compose queries by pasting together strings than to use secure libraries, such as prepared SQL statements.*

This hypothesis is suggesting that command injection vulnerabilities are in part about psychological acceptability of secure defaults: "It is essential that the human interface [e.g., of a database access system] be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly" [8]. Strings are easy to use, while more secure means of constructing commands are more difficult. This leads us to the following interconnected hypotheses:

H2: *If SQL is embedded natively in the host language, it will be easier to use SQL directly than to construct queries with strings.*

H3: *If developers use a host language's natively embedded, safe version of SQL, they will be less likely to introduce SQL injection vulnerabilities.*

```
1  let authorName : String = user_input
2   let webpage : HTML = ~
3     <html>
4       <body>
5         <h1>Search results:</h1>
6         <ul id="results">
7           {query_results(db, ~)
8             SELECT author, bookTitle FROM books
9             WHERE author = {authorName}}
10        </ul></body></html>
```

Figure 1. A sample Wyvern program. Code in black is Wyvern (host language); code in purple is embedded HTML; and code in orange is embedded SQL.

With SQL embedded in the language, we can *provide a positive security model* for SQL (all queries are constructed using the SQL language), rather than by using program analysis to identify unsafe cases (i.e., a negative, or blacklist, model) of pasting commands together from strings. Furthermore, having programmers write queries in structured SQL *keeps security simple* in contrast to using program analysis, which may involve tracing the flow of user input through multiple functions into an SQL query.

Figure 1 shows how an SQL query looks in Wyvern: the code snippet presents a program that serves a simple webpage displaying a list of book titles for a given author name that is extracted from a database. The code looks similar to a concatenated strings version, but in fact it is not. Two embedded languages are used to produce this code snippet: embedded HTML (shown in purple) and embedded SQL (shown in orange). The value of the `authorName` variable is provided by the user and is inherently unsafe. If we were constructing the query by string concatenation, an SQL injection could occur on line 9. However, because the SQL language is embedded within Wyvern, the author name is integrated as data and cannot be misinterpreted as a command, precluding the command injection vulnerability [5] while at the same time enhancing usability—readability and understandability—of the code.

While prior languages have incorporated SQL as native syntax, we designed Wyvern as an extensible language, so that libraries can extend the base syntax of Wyvern [5]. Thus, any command language can be incorporated, guarding against any kind of command injection attack [3], not just SQL injections. In many prior extensible languages it was possible for language extensions to conflict. Wyvern eliminates these conflicts and the attendant usability problems by using indentation as a way to delimit an embedded language and using the expected type of an expression to determine which embedded language to use [5].

3. Security Principles in Wyvern's Design

In addition to specific programming language design decisions to defend against command injections, more generally, Wyvern has been designed to facilitate developers' adherence to security principles and software development practices that are said to decrease security risks.

For example, one common principle prescribes the use of *secure defaults*. To enforce this principle, Wyvern does not allow `null` as a default value. Furthermore, we specifically chose the syntax for option types to be as convenient as possible, supporting psychological acceptability for our decision to forbid `null` defaults.

In order to support the *principle of least privilege*, we are designing Wyvern's module system to provide *architectural control*, limiting the parts of the program that can access security-critical resources such as the network or file system [1]. The capability-based architectural-control mechanisms in Wyvern can be used, for example, to ensure that dynamically loaded code or third-party libraries are given no more trust than necessary.

Finally, the principles of *keeping security simple* and *using open design* were addressed by making Wyvern's syntax and semantics straightforward and Wyvern's type system statically checked and as simple as possible.

4. Conclusion

In this paper, we examined how the Wyvern programming language's design addresses command injection attacks suggesting that the programming language design can indeed impact the security of the software written in it. Although the arguments in this paper need further validation, they illustrate ways in which language design can potentially improve security.

References

[1] J. Aldrich, C. Omar, A. Potanin, and D. Li. Language-Based Architectural Control. In *IWACO*, 2014.

[2] K. Hickey. Most secure Web programming language? It depends., April 2014. URL `http://gcn.com/articles/2014/04/24/programming-language-security.aspx`.

[3] D. Kurilova, C. Omar, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Type-Specific Languages to Fight Injection Attacks. In *HotSoS*, 2014.

[4] L. Nistor, D. Kurilova, S. Balzer, B. Chung, A. Potanin, and J. Aldrich. Wyvern: A Simple, Typed, and Pure Object-oriented Language. In *MASPEGHI*, 2013.

[5] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Safely Composable Type-Specific Languages. In *ECOOP*, 2014.

[6] OWASP. Category:OWASP Top Ten Project, 2014. URL `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`.

[7] P. Perego. Which Is the Most Secure Programming Language Ever?, July 2012. URL `http://armoredcode.com/blog/which-is-the-most-secure-programming-language-ever/`.

[8] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Proc. IEEE*, 1975.