Security in Programming Languages

Colin B Hamilton

December 15, 2015

Abstract

The most popular programming languages in use today were not designed for security. Programmers still, generally, must make up their own approaches to preventing security flaws and patching vulnerabilities and if, just once in thousands of lines, they forget to verify data before using it, there is no fallback.

This paper explores security as a property of programming languages. By understanding the sources of the most common security vulnerabilities, it may be possible for the environment of the language to prevent some altogether. In particular, it explores constructs and properties of languages that could prevent or mitigate injection attacks, one of the most prevalent security vulnerabilities in modern web applications.

1 Introduction

Software vulnerabilities have great variation among them, but in the end, nearly all come from preventable human error. Whether due to laziness, inattentiveness, or a simple lack of knowledge, small errors in writing systems can have enormous consequences when attackers find and exploit them. What's more, there are clear patterns in the types of mistakes that are made. The list of top security vulnerabilities hardly changes from year to year. While many companies work to patch security issues when they are found, preventing these issues from appearing in the first place is a task that gets more difficult the larger systems become.

This paper explores security at the programming language level, in the hope that more secure languages could prevent some of these issues. By including constructs in the tools and languages that create and organize systems, the hope is that more issues can be detected by the compiler, interpreter, or operating system running the code. While this cannot prevent all security issues, automation can provide an important first step in preventing common security issues in large systems.

OWASP, The Open Web Application Security Project, provides an online list of the top security vulnerabilities.[1] It lists injection as the most common security vulnerability. This paper focuses primarily on techniques for preventing injection in web applications. Similar techniques could potentially also work towards the prevention of cross-site scripting, a related vulnerability, and number three in 2013. This paper will then briefly address related work in secure programming language research, including preventing insecure uses of resources. This problem is related to "insecure direct object references" and "sensitive data exposure" in fourth and sixth place, respectively, in OWASP's ranking. Finally, this paper will note areas of security where programming language design is not likely to provide a solution.

2 To the Community

The state of computer security is disheartening. Vulnerabilities common twenty years ago are still rampant. While tools and techniques exist that can identify many common vulnerabilities, there is a cost: it takes extra time and effort, and detracts from the developer's typical workflow. For this reason, such tools have largely failed to make a significant impact on most applications.

The benefit of a language-based approach is that it is built in to the developer's environment, instead of being a separate piece that can be ignored. It gives assurances (to the programmer, team, and organization) that some vulnerabilities will not exist, and it does this automatically, by default. A secure programming language could be an invaluable tool in improving the state of security.

3 Preventing Injection

SQL injection is an incredibly common vulnerability, and it has been this way essentially since it was first discovered. A recent Vice article gives an overview of the history and basics of SQL injection, and remarks on the fact that it has been a common attack for over fifteen years.[2] The article credits Jeff Forristal with the first documentation of this vulnerability in 1998, when he explained that users could "possibly piggyback SQL commands." In their input they can include an unexpected value that "forces the database to do something it's not meant to do."[2]

General injection attacks are not limited to SQL injection, though that is a very common form. Injection can take place any time user input is used to create a command meant to be run by an external application. If input is not properly handled (which is frequently the case), users can insert special control characters. These can then change the meaning of the query from what was intended.

As an example, suppose a webpage validates users with the following code:

```
password = request.get('pass')
login = request.get('login')
query = "SELECT id FROM users WHERE password =
    SHA1('" + password + "') AND login = '" +
    login + "'"
```

When input is submitted normally, eg. `login=mchow01&pass=baseball01234`, the query becomes

```
"SELECT id FROM users WHERE password =
    SHA1('baseball01234') AND login = 'mchow01'"
```

which works as expected. But if a user submits a login of the form `pass=blah&login=none' OR '1' = '1`, the query becomes

```
"SELECT id FROM users WHERE password =
    SHA1('blah') AND login = 'none' OR '1' =
    '1'"
```

which, because `'1' = '1'` is always true (and AND has higher precedence than OR), will select all users in the database.

This example shows the key behind injection attacks: the attacker is trying to get their own input executed as part of a command. Techniques exist to prevent such attacks, including stripping control characters (like the single quote in the example) from user input strings. But what if a developer forgets to do such a thing? Ideally, the mistake could be caught when before the query is executed.

3.1 Tainted Variables

The idea of *tainted variables* has existed since early versions of the Perl programming language, and at least as early 2001 in Ruby.[3, 4] The idea behind tainting comes from the knowledge that user input can never be trusted. Therefore, any values that have come from a user should be considered dangerous. They carry a *taint* that is passed on to any values derived from them. Specifically, if a tainted string is concatenated with another string, the resulting string will also be tainted.

This results in taint propegating throughout the program. Operations that must be secure can inspect parameters to see if they are tainted – and the operation, perhaps, could report an error if an attempt was made to run it on tainted data.

The system of tainted variables can help developers analyze a system that is already in place. Because the path of tainted data is clear, they can find the operations that might be fed dangerous data, and work to secure them.

Using taint for anything more than analysis, however, is not likely to help. The reason is that, for innumerable types of queries made by web applica-

tions, the query *requires* user data. For example, the application may need to verify a login or search for a particular term. In these cases, use of the user's input in the query would mark the entire query as tainted, despite that it may be prefectly safe. There must, then, be some way of removing taint. It would make sense to remove the taint from a variable if it passes an analysis.

This brings us back to validation of input strings; but, if we still require developers to do this manually, what help was the language in keeping track of taint? Ultimately, injection attacks work because they can masquerade as a normal command, so validation of the query string cannot happen *after* it has been constructed.

This, in addition to other, more specific concerns (such as user inputs being used in multiple types of commands), suggest that taint checking, while a good tool for analysis, will not work as a general solution to the problem of injection.

3.2 Embedded Languages

Many efforts, especially recently, have focused on embedding the language of concern into the host language. Commands, instead of being constructed with strings, would be special literal values that the language environment can interpret. That way, the language is responsible for building and executing the query, so it can properly escape strings.

One approach to this issue is that by the Wyvern programming language, currently in development.[5] Instead of strings, Wyvern introduces what its authors call *type-specific languages* – literals that allow one to easily construct an object.[6] For example, a library could define a `SQLQuery` type that allows the following syntax to construct a query:

```
let query : SQLQuery = SELECT id FROM users
    WHERE password = SHA1({password}) and login
    = {login}
```

This literal would be constructed into a `SQLQuery` object, and the inputs would be properly and safely integrated into the query. Note that this is *not a string*; instead, everything after the assignment operator is parsed according to the `SQLQuery` type's specification. The input strings are placed into this data structure, which unambiguously marks them as literals, so that injection is impossible.

The extensible nature of this system also means that it does not rely on the language designers' foresight. This system can be applied to any kind of structured data, including HTML, XML, JSON, and command line scripts. When future languages are created, this system allows for them to be integrated too.[6]

Though research and development on Wyvern is still underway, mainstream languages have attempted to adopt similar features. The most common approach is through templating, which allows programmers to place delimiters within their query string that are safely filled in by the API. This is the approach taken by Python's `MySQLdb` module, and Java's `java.sql`.

And yet, injection vulnerabilities are still common. Template strings are still strings, and can still be constructed through concatenation. Templating and embedded languages could eventually help, but it seems for the time being that programmers are more comfortable sticking to what they know, namely string concatenation. As long as these concatenated strings can still be executed as queries, with no oversight, these vulnerabilities will likely remain.

3.3 Record-Keeping Strings

The earlier description of injection provides intuition, but for a true foolproof prevention technique, we need a more precise definition of such an attack. In their paper "The Essence of Command Injection Attacks in Web Applications," Zhendong Su and Gary Wasserman provide the first definition of an injection attack. They formulate the definition based on how user input strings map to the abstract syntax tree of the resulting query.[7]

To facilitate their definition, they consider the query string not as a single, unbroken string, but rather as a concatenation of substrings. This is, after all, how the query is built. Some of these substrings came from the user, others did not. Parts that came from a user are enclosed in delimiters, so the *specific*

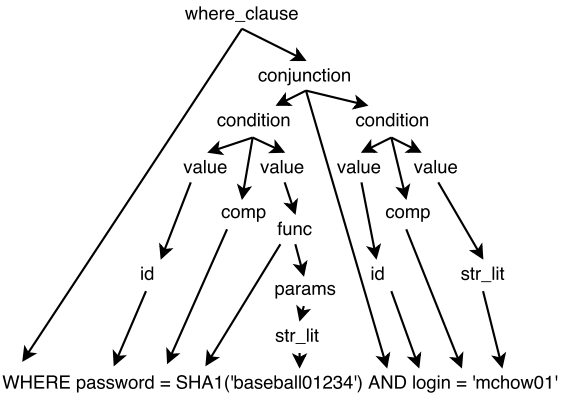

Figure 1: A subtree of the AST for a valid SQL query. Note that the user input strings correspond to specific subtrees (`str_lit` nodes, in this case).

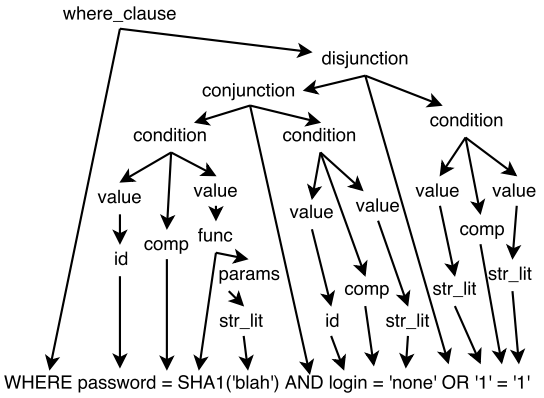

Figure 2: A subtree of the AST in an SQL injection attack. Note that with the addition of the disjunction node and its right subtree, the user input string crosses subtrees. No single subtree encompasses the user's input.

portions of a query can be identified that might be dangerous.

To execute a query, it must be parsed and turned into an abstract syntax tree (AST), representing the meaning of the query. The AST has branches based on the different parts of the query. Figure 1 shows an example of the AST of a normal SQL query. An injection attack could potentially result in a malformed query, but such an attack would fail anyways and so is of no interest. Serious potential attacks are those that do map to a valid AST. Figure 2 gives an example of an AST for a successful injection attack.

The difference between the two ASTs, Su and Wassermann argue, is that in the first, the substrings from the user correspond to their own subtrees in the AST. In the injection attack, however, they cross subtree boundaries, essentially escaping the part of the AST they were intended to be confined to. The authors use this observation to formulate a definition of a SQL command injection attack. Essentially, their definition states that a query is an injection attack if and only if there exists a substring s that came from a user, for which there is no node in the AST whose descendant leaves comprise s.[7]

Discovering such an attack requires checking the substrings that came from the user against all nodes of the AST. If there is a substring with no matches, then this is an injection attack. Importantly, this approach requires tracking user input throughout the program. The paper's solution is more specific, however, than taint checking. The latter can only identify a binary state: a given string value is either tainted, or is not. Their solution is instead to have each string keep track of any substrings that originally came from users by enclosing them in delimiters.

This kind of behavior is difficult to integrate with existing languages, because it causes interference with existing algorithms that operate on strings. Inserting delimiters around inputs only truly works if the delimiters 1) cannot be removed or modified by programmers, 2) cannot be part of the original input string , but 3) can be observed by code checking for injection.

Su and Wasserman's solution is to choose delimiters as a random sequence of alphabetic characters that are not English words. This satisfies the third requirement, and the authors reason that, in the second requirement, collision is unlikely. For the first requirement, they say that they suspect that programmers will probably not filter out alphabetic characters from a user input string.[7]

Their solution is based on heuristics, which is per-

haps the best that can be done if the only hope is to retrofit less extensible programming languages with this functionality. But it is possible to do better in general, by building a string class that recalls the pieces that were used to build it, while supporting normal string operations, and allowing inspection of the string to determine the pieces that came from users.

A proof of this concept is presented in Python, which was chosen because of its extensibility.

4 Application

As a supplement to this paper, I provide a link to a set of three Python modules that can be used to write a web server, available at `https://github.com/cbh66/safeserver`. The modules rely on a common definition for a `SafeString` class. This class supports all of the same operations as a normal Python string. In addition, `SafeString` objects keep a record of the components used to build the string, specifically those considered *unsafe*. But importantly, developers can interact with such objects without knowing this; they can use and build them like a normal string.

The other two modules lie between the developer's code, the server code, and the database code. The first module, `safeserver`, is a "go-between" for the developer and the server framework. It simply modifies the built-in methods for retrieving user input for get and post parameters, returning a `SafeString` instead of a normal string. The second module, `safesql`, is for interacting with SQL databases. It handles `SafeString` parameters specially, which it can do because `SafeString`s have a record of which pieces came from users.

Importantly, use of these modules guarantees that if all user input is obtained through the server API methods, and if all queries are made through its SQL methods, then injection is impossible. Notably, clients of the code can treat inputs as if they were strings. Unfortunately, because the implementation is an addition to an existing language, instead of a built-in feature, it is possible to get around it, specifically by modifying the private variables of the object or by casting it to a raw string. Barring these kinds of abnormal interactions, security from SQL injection is guaranteed.

This is primarily a proof of concept, and has not been optimized for time or space efficiency. The `safeserver` module works only with the `webapp2` server configuration, although the "go-between" code is so short that it can easily apply to other frameworks as well. Further, processing of `SafeString`s cannot yet be applied to other queries or code fragments that could be subject to injection attacks (cross-site scripting, directory traversal, etc). Again, however, the same techniques for writing the SQL module could be applied to these other areas as well. All that is needed is a parser for the format of interest.

Python was chosen for this implementation because it allows for operator overloading with dynamic dispatch, and does not treat built-in strings any differently from user-defined classes. The same cannot be said for many other languages, where this solution would not work. However, this proof of concept is not about modifying existing languages. It is to demonstrate that such strings could exist as a built-in feature of a language.

5 Other Vulnerabilities

Although this paper has primarily focused on preventing injection, there are a number of other vulnerabilities that are common in software.

5.1 Secure Data Flow

While it may be less of a direct threat than injection attacks, information leakage can be a risk for a system that is meant to restrict access to certain categories of data. Manual checking of credentials is easy for a developer to forget, and verifying that an application properly restricts information flow is very difficult - any tool that would analyze it would need to know how the data is *meant* to be restricted.

The easiest way to restrict access to data is by giving labels to data that describe who can read and

write to it. From there, it is easy to automate checking of labels when access to labeled data is attempted.

One implementation of data labeling is suggested in the design of a language called Laminar. The authors place security of resources into the language's type system. They make the distinction between secrecy labels (which restrict who can read), and integrity labels (which restrict who can write).[8]

Actors (ie. functions, threads, and processes) are restricted to reading only data for which they have the secrecy label, and writing to resources for which they have the integrity label. Some actors may be given the ability to classify or declassify information by copying it with a higher or lower set of secrecy labels.

The important part is that these labels must be made explicitly by the programmer. It is fairly easy to do so: programmers enclose secure blocks with a `secure` keyword, which specifies the secrecy and integrity levels they are requesting. It is assumed that this will happen only infrequently, and so will not be a large burden on the programmer. Moreover, it requires the author to make explicit which sections of the code are critical.[8]

A downside of the approach presented is that, to be feasible in a system where processes interact with system resources, the operating system itself must take charge of enforcing labels. This requires the introduction of system calls to establish labels of processes and shared resources (like files and pipes). The paper was able to accomplish this with about 1500 lines of code added to the Linux kernel.[8] While this is not much, it is perhaps too much to expect organizations to change their operating system.

5.2 Problems That Can't Be Solved

A developer's programming language can be a powerful tool to prevent her from making mistakes. It is worth noting, however, that there are security flaws that are unlikely to be solved by programming language design.

5.2.1 Insecure Authorization and Weak Passwords

The second vulnerability in OWASP's top ten ranking is "Broken Authentication and Session Management."[1] This does not originate in a specific program, and so do not fall under the jurisdiction of a programming language. Instead, it is based on the design of the system. Examples of these flaws are exposing authentication tokens, improperly hashing passwords in a database, and sending authorization information over insecure channels.

Related to this flaw is the use of weak passwords by users. Users with weak passwords are exponentially more likely to have their accounts compromised, so it is in their interest that they not be allowed to make passwords that are easily guessed or found with a password-cracker. Building a verification system into the language environment is impractical, in part because of the performance penalty, and in part because there is not likely to be a one-size-fits-all solution.

5.2.2 Security Misconfiguration

The fifth vulnerability in OWASP's top ten ranking is "Security Misconfiguration."[1] These weaknesses can come from forgetfulness (leaving a setting enabled) or miscommunication between developers in different areas. A language would be unlikely to catch many of these issues (eg. open ports or web pages, extra privleges) because it could not know what was intended.

5.2.3 Social Engineering

The nature of social engineering is manipulation and deceit – using human nature to get access to a system or its information. This can take many forms, including email phishing, dropping a dangerous USB outside an office, impersonation, etc. There is not likely to be a good technological defence against these attacks until our machines are intelligent enough to recognize them (and that might take some time). As it is, the best defence is the establishment of procedures for these types of situations, along with proper training of employees.

6 Conclusion

Programming language design has introduced powerful concepts that help users design, built, and debug systems. Integrating security into the language is an extension of that premise. For those languages that lend themselves to extensibility, adding modules could achieve this goal, as shown in the proof of concept supplement. For the more general problem, however, such issues should be taken into account when new languages are designed, as a core part of the language. Security concerns too often come as an afterthought in building systems, leading to the state of computer security today. If security is ever going to be improved, this must change, and what better place to start than in improving the tools we use?

References

[1] OWASP. "OWASP Top Ten Project", (Columbia, MD: Open Web Application Security Project). https://www.owasp.org/index.php/Top_10_2013-Top_10

[2] Joseph Cox. "The History of SQL Injection, the Hack That Will Never Go Away", Vice.com, November 20, 2015. http://motherboard.vice.com/read/the-history-of-sql-injection-the-hack-that-will-never-go-away

[3] "perlsec – Perl Security". Official Perl Documentation. http://perldoc.perl.org/perlsec.html#Taint-mode

[4] Andy Hunt and Dave Thomas. Programming Ruby – the Pragmatic Programmer's Guide. Addison Wesley Longman, Inc, 2001. http://ruby-doc.com/docs/ProgrammingRuby/html/taint.html

[5] Darya Kurilova, Alex Potanin, and Jonathan Aldrich. "Wyvern: Impacting Software Security via Programming Language Design", October 2014. http://www.cs.cmu.edu/~aldrich/papers/plateau14-wyvern.pdf

[6] Cyrus Omar et al. "Safely Composable Type-Specific Languages", http://www.cs.cmu.edu/~aldrich/papers/ecoop14-tsls.pdf.

[7] Zhendong Su and Gary Wasserman. "The Essence of Command Injection Attacks in Web Applications", January 2006. http://web.cs.ucdavis.edu/~su/publications/popl06.pdf

[8] Indrajit Roy et al. "Laminar: Practical Fine-Grained Decentralized Information Flow Control", June 2009. http://www.cs.utexas.edu/users/witchel/pubs/roy09pldi.pdf