Survey of Security in Programming Language Design

Conor O'Brien
csobrien@wm.edu
College of William & Mary
Williamsburg, Virginia, USA

Abstract

Previous surveys of security vulnerabilities have predominately focused on static identification. This paper seeks to fill the relatively unexamined niche in security survey literature which examines the design of programming languages themselves. It examines the vulnerability features of particular programming languages, as well as general features of programming languages, and potential methods used to mitigate the issues arising thereof. I express the language design maxims of Security-convenience alignment, Security consciousness, and Programmer-first safety.

CCS Concepts: • **General and reference → Surveys and overviews**; • **Security and privacy**; • **Software and its engineering**;

Keywords: programming languages, language design, vulnerabilities

1 Introduction

Dowd et al. [15] give a definition of vulnerabilities:

> In the context of software security, vulnerabilities are specific flaws or oversights in a piece of software that allow attackers to do something malicious—expose or alter sensitive information, disrupt or destroy a system, or take control of a computer system or program.

They further distinguish between the general classes of design, implementation, and operational software vulnerabilities. Herein, I will focus primarily on implementation software vulnerabilities, those that depend more so on the specific code fragments used, rather than conceptual flaws, or flaws that appear when the code is used and integrated into its environment. Implementation software vulnerabilities arise within particular a particular code fragment which is "generally doing what it should" [15], but nonetheless a mistake is present.

Previous survey approaches [17, 23] focus on vulnerability analysis methods for the discovery of such software vulnerabilities. This is particularly useful for ensuring the security of existing code, or code already in development.

Beyond analyzing individual software fragments, however, lies a broader topic: Programming language features (PLF) themselves can be assigned a security value, and particularly insecure PLF can promote insecurity for programmers of that language. One may risk writing more insecure software by choosing one language over enough, especially given the task at hand.

Given the variety of experiences programmers may be equipped with, it is unreasonable to expect every programmer and developer to be well-versed enough in how secure the various features in their programming language are. The USDoL Occupational Outlook Handbook [4] regards a Bachelor's degree as merely often required for entry-level corporate programming, although for more advanced software development positions, a master's degree may also be a requirement [9]. There is also the matter of independent developers, who not only are not necessarily vetted for experience by nature of their contract, but also must balance competing professional and market incentives [25].

In this paper, I will survey popular programming languages and analyze their strengths and weaknesses at a PLF level. I will also examine classes of programming language features which are open to security discourse. After that, I will discuss my findings in general and propose general advice for programming language designers and software developers.

2 Theoretical Background

There are widely-adopted secure PLF that one may take for granted at this mature stage of programming language design, such as static typing [16] in most compiled languages (ML, C, Java, etc.). As most software vulnerabilities are also software bugs [15], an essential component of designing a secure language are designing one which naturally leads to less bugs, and consequently, less software vulnerabilities.

Secure language designed can be achieved in a variety of ways, but some common approaches are by implementing comprehensive error and warning reporting, excluding insecure functions from the language specification, and having a well-defined software life cycle that allows for rapid changes responsive to vulnerability discovery.

A language's implementation is therefore of equal importance to its specification, if not of greater importance. Although one can specify a standard with almost mathematical precision, the exact implementation details are still liable to bugs, specification misinterpretations, and even implementation design choices that the specifications may not or do not account for. Even intentional design features may be harmful. In the next few subsections, I wish to survey some instances, historical and modern, where programming languages have or had notable security flaws within the language itself.

3 Language Survey

No discussion of programming language design would be complete without examining how existing programming languages and PLFs perform. I will instantiate the concepts outlined in the theoretical section above to particular languages, showing where they fail and succeed.

3.1 C and C++

C is a popular language with some ancient and insecure features, most often used for low-level programming, but also used significantly for generic programming (even including the Python reference implementation [27]). Some C PLFs can be directly identified with common and dangerous classes of vulnerabilities, such as the generic buffer overflow issue, as well as the language-specific format string attack. C++ is vulnerable to these same vulnerabilities in that most insecure C code fragments which fall in one of the below categories still compile in C++ without warning, although such programs are hardly reminiscent of idiomatic C++.

3.1.1 Buffer Overflow. The library functions `gets`, `scanf`, `sprintf`, and `strcpy` are each liable to buffer overflow vulnerabilities. Previous methods attempt to combat these vulnerabilities dynamically [22] or statically [17, 21]. The root issue, however, lies within both C as a language and as a specification. First, a positive: The C ISO/IEC 9899:201x standard [18] officially deprecated the `gets` function, and most C modern compilers [3, 6] will not compile code using the function. Yet, some C compilers, such as The Tiny C Compiler (tcc) [10] have not removed the function, and most compilers still offer ways to access older standards which do not fully remove `gets`, such as `gcc -std=iso9899:199x`. So long as there are methods to use the function, a sufficiently determined programmer, perhaps motivated by trying to update legacy code, can enable the usage of `gets` in most cases.

Though the current state of `gets` may not raise much concern, the remaining functions I mentioned have yet to be deprecated or addressed in any way, neither by language specification nor implementation. For instance, despite featuring an egregious buffer overflow vulnerability via unchecked `strcpy`, the following C program compiles without error or warning in major C and C++ compilers [3, 6], sometimes even without a runtime error [3], all while using debugging and warning compiler flags (e.g. `gcc -g3 -Wall`):

```
#include <stdio.h>
#include <string.h>

int main(int argc, char** argv) {
  char buf[10];
  strcpy(buf, argv[1]);
  return 0;
}
```

It was easy to deprecate `gets` because of its unequivocally dangerous behavior. Although these remaining functions do not have such strong arguments for removal, such as the legitimate usage of, say, `strcpy(units, "quarts");` [18], the more dangerous usages of this function, such as the above code snippet, are not guarded against, despite being somewhat predictable in nature.

3.1.2 Format String Attack. Format string attacks can be considered a subset of buffer overflow vulnerabilities, but the distinctions in attack method and mitigation are notable for this survey. Examine the `printf` family of functions, which take a format string as a first parameter, followed by any number of parameters which are consumed as part of parsing the format string. For example, the following C code prints a string followed by an integer according to the given format string:

```
#include <stdio.h>

int main(int argc, char** argv) {
  printf("%s / %i\n", argv[1], argc);
  return 0;
}
```

The intention for this family of functions is for the format string to be known at compile time, or at least be generated with predictable behavior. Given the low-level nature of C, there is no error checking involved at runtime, but instead there are warnings given for a mismatch between format string types and supplied types, as well as missing or surplus parameters, when compiling with warnings enabled.

Now, consider the "simplest" method to write a string to STDOUT without a trailing newline. The uninitiated in this particular attack may well say `printf(string)` instead of the safer `printf("%s", string);`, which is where the vulnerability arises. In such a case, the supplied string is interpreted as a format string, and when an external agent has control over what the contents of that string are (e.g., it comes from STDIN, a command line argument, or a remote request), then they can perform the format string attack.

As mentioned, the `printf` family of functions do not perform runtime checks on their arguments. Due to C's implementation of variadic functions (i.e., functions without a fixed number of arguments), the length is unknowable, and must either be supplied (inconvenient) or inferred (as in `printf`). The vulnerability emerges with the `va_arg` macro, responsible for incrementing the argument pointer and accessing the next argument supplied to the function. Since the arguments to `printf` exist on the stack, `va_arg` will read successive entries from the stack. This means it can leak information and even lead to stack corruption.

While practical format string attacks able to redirect program execution and leak potentially sensitive data exist [22], modern compiler technologies can at least partially mitigate these vulnerabilities, although not insurmountably [26].

3.1.3 Discussion. One could argue the aforementioned proposal for warnings about insecure behavior in these C functions cannot be reliably generated statically. By nature of C's design, there still remains the more general issue that there are no simultaneously safe and convenient alternatives to functions like `strcpy`; to be "extra safe", one must use `strncpy` and supply the number of bytes to read manually, thereby sacrificing the convenient syntax of `strcpy`. Were C's strings necessarily coupled with the length of their contents, then the concepts of safe and convenient could be merged into a safe variant of `strcpy` which incorporates the usual security checks one would need to do if using the function with untrusted source input. Of course, given the low-level objectives of C, as well as the lack of inclusion of a standard string type, this solution is hardly tenable for C.

So long as the insecure versions of these functions exist within C, programmers, particularly more novice programmers, will use them in production code, especially since they also appear to be the most convenient expressions of certain code patterns. Although there exist countermeasures to such insecurities creeping in within the software development life cycle, such as code review, the mere existence of such functions necessitate that, as imperfect agents, programmers will invariably introduce these insecure functions to their code despite the countermeasures.

3.2 Python 2 and 3

Let's turn to a much more beginner-friendly family of languages: Python. Although this example is primarily historical, as opposed to the previous example, which showcases existing insecure PLF in C/C++, it still serves as a useful point of discussion.

In Python 2, there are two functions which obtain a line of input from STDIN: `input` and `raw_input`. From a language design perspective, `input` is to be used to allow the user to input Python objects for manipulation of the program without having to force the Python programmer to do parsing on their own. It might be desirable for easy parsing of a requested input of an array of integers, for example. In practice, the input is parsed by evaluating it as Python code. In a Python program which imports `os`, for example, we can supply the program that uses `input` with the payload `os.system("curl https://evil.domain/hack.sh | bash")` to execute any bash file remotely via the shell, elevated to the process privileges.

Similar to the process of deprecating `gets` in C, the functionality of Python 2's `input` was superseded in Python 3 by the functionality of `raw_input`, which simply reads a line from STDIN as a string and returns said string.

3.3 JavaScript

While this section could have alternatively been written about other interpreted languages like Python or Ruby, I choose JavaScript specifically because it is the language of browsers. The internet has a wonderful capacity to connect individuals and allow them to share their experiences. In turn, programmers must be diligent in how they enable users to communicate and share information with each other. I will return to this point in a moment.

JavaScript, like other interpreted languages, has a function which allows you to evaluate a string as code in the same language: `eval`. In a way, it poses a problem similar to the problem Python 2's `input` did, insofar as Python 2 implicitly calls `exec raw_input()` when evaluating `input()` (where `exec` is Python 2's equivalent to `eval` in JavaScript). However, because you must make an explicit call to `eval`, the differences in how the programmer uses it differ slightly. Some programmers may simply be unaware of the implications of calling `eval` on a user-supplied input. Others may think they are safe in performing input sanitation/validation on the string they pass to eval.

As a motivating example of the allure and consequent perils of `eval`, imagine designing a calculator web application, where users can share their input expressions to other users as links. Naturally, one might wish to leverage JavaScript's `eval` command to do the brunt of the work. Keeping input validation in mind, one might reject all strings which contain alphabetic characters before evaluating the input string. The resulting function may look like this:

```
const safeEvaluate = userInput => {
  if(/[a-z]/i.test(userInput)) {
    notifyUser("Invalid input: " + userInput);
    return;
  }
  return eval(userInput);
};
```

Although this may seem adequate validation, surprisingly, JavaScript (among other languages) are Turing Complete using a non-alphabetic subset of characters. For example, we could provide the JSFuck [19] program found in Appendix A as input to have the application share the user's `document.cookie`. It bypasses this filter, since it only uses combinations of the 6 characters `[]()!+`. This is possible due to JavaScript's exceptionally quirky type conversion system. This function now constitutes a cross-site scripting attack.

I will mention two methods for mitigating this issue. First, one could exclude all characters besides numeric characters and the basic mathematical operators. This, while inelegant and inextensible, does work. More preferably, a more sophisticated algorithm should be used, such as implementing a shunting yard algorithm, or using existing expression evaluation software that better fits the application design [14, 24].

Nonetheless, the existence of `eval` in JavaScript and similar languages makes it an appealing tool to handle user expressions, especially for those less familiar with existing libraries for the subject and with the potential pitfalls of evaluating user input. In this case, there is probably not much

that can be done on the language design front without being annoying in, say, emitting a warning the first/every time `eval` is invoked.

3.4 Discussion

These language-specific surveys demonstrate a few core concepts language designers should keep in mind. I will summarize my findings in the form of a list of maxims for the language designer, as well as developers wishing to be mindful of and have language to assess potential programming languages.

Security-convenience alignment. As best demonstrated by the deprecation of C `gets` and Python 2's `input`, when there is dissonance between convenience and security, programmers are more likely to introduce security vulnerability to their programs if they are not especially mindful. These vulnerabilities can be easily designed if the avenues for vulnerability are removed or fixed.

Security consciousness. As seen with the plethora of unsafe behavior still able to be invoked in C/C++ and JavaScript, when choosing or designing a programming language, one should prefer languages with fewer common vulnerable PLF, and more safe ones. If one does not need to use a language with more vulnerable PLFs, one should prefer other, safer languages over it, so as to minimize potential security vulnerabilities in advance of code writing.

Programmer-first safety. Even the most careful of programmers, as humans, are error prone. Giving programmers unilateral ability to succinctly make dangerous mistakes gives them the ability to unwittingly make those mistakes, as seen with the format string attack vulnerability in C. Therefore, for the programmer's benefit, including runtime checks and descriptive error messages can reduce risk of doing something disastrous unintentionally, thereby reducing the risk of security vulnerabilities.

Applying these three maxims can overlap and have interplay. For example, by adhering to Security-convenience alignment and making insecure behavior less convenient to achieve, one upholds both Security consciousness and Programmer-first safety. This upholds Security consciousness in that vulnerable PLFs are less accessible and consequently less likely to appear in normal code. It upholds Programmer-first safety in that the programmer has a smaller chance at making a mistake in using a vulnerable PLF.

4 Feature Survey

This paper now turns to a survey of some broad, general features and concepts present in a variety of languages.

4.1 Regular Expressions and Denial of Service

Regular expressions (regex) are a helpful feature found in almost any modern language which are used to validate input, perform text substitutions, and a variety of other tasks. They are succinct and easy ways to accomplish their tasks, and save manually writing tens to hundreds of lines of mundane parser code.

To support backreferencing, regex engines often transform regex into corresponding non-deterministic finite automata (NFA) [12]. Due to the nature of NFAs, interpreting certain NFAs can involve backtracking, which, in the worst-case scenario, can be very expensive, often with polynomial or exponential complexity [12]. This means that regexes of a particular form can cause regex engines to consume processing resources for a prolonged period, potentially constituting a denial of service (or DoS) attack. Davis et al. [12] identify this as particular problematic in Node.js, a popular server framework, where a single problematic regex can hang the entire server. A class of such regexes are termed *super-linear* (SL) regexes.

Programming languages using regex engines sometimes unconditionally use NFAs, even where conventional algorithms not bounded above by polynomial complexity could suffice [28]. Practically, swapping between these algorithms is harder than it would seem, and some engines opt to simply optimize NFA traversal by avoiding backtracking altogether when possible [11] or avoiding redundant state traversal [12].

These approaches are unsuccessful in entirely disarming the threat posed by SL regexes; regexes liable to search space explosion are commonplace and natural, especially in third-party software [12]. While innovations in regex engines continue to reduce the effect of ReDoS attacks, the attack method continues to pose a considerable threat [13]. The best mitigation remains to be revising affected regular expressions to avoid the use of SL regexes.

4.2 External Library Dependencies and Trust

Often considered one of the hallmarks of modern programming languages, external library integration is a powerful tool for any developer attempting to share their own code or reuse the code of others. The Node.js JavaScript implementation has npm [1], Python versions have pip [8], Ruby has RubyGems [2], etc.

While programmers often take these libraries for granted, tracing dependency chains reveals a complicated interpersonal lattice. A programmer may easily run `npm i node-ipc` on their command line, import that library in their program with, say, `import ipc from 'node-ipc';`, and use it without a second thought. Indeed, there is an expectation that popular open-source third-party libraries function correctly and safety. That is to say: We place an awful lot of trust in third parties.

The recently-coined concept of "protestware" classifies software which has the affect of distributing social, political, or personal stances through mutating code behavior [20]. Kula and Treude [20] describe both malignant and benign protestware. Malignant protestware is pertinent to the topic

at hand, and gestures to the broader issue with the amount of trust we place in library developers, who are easy to forget as being mere humans, often working alone or with small teams, yet depended upon by many.

The hallmark example of malign protestware is node-ipc [7], which was modified in response to the Russia-Ukraine conflict to overwrite the files of Russian and Belarusian users of software which imported node-ipc with a single heart emoji [5]. While those of us who disagree with Russia in their invasion of Ukraine may not be able to immediately recognize the dangers posed by actions patterned after this kind of protestware, we should recognize that developer's intentions may not always be so noble as protesting social injustice. Indeed, a developer can inject malign protestware, or even actual malware, into a library they maintain for a wide swath of reasons; or, through standard phishing tactics, the credentials of a developer may be compromised, and any library they maintain may have malware injected into it.

4.3 Discussion

Generally, many software vulnerabilities can arise due to misplaced trust, either in general language features, or in third-party libraries.

What mitigations can be enacted to combat vulnerabilities in third-party, whether ReDoS or library poisoning? The most immediate mitigations for software development are to integrate code review of third-party sources into the development life cycle, as well as fixing your software requirements to not automatically use versions newer than those cleared via code review [20]. Unfortunately, language design tactics alone cannot address these vulnerabilities outside of removing the affected features.

What about mitigations on the language design front? We once again encounter the tension between convenience and security.

In the first case of ReDoS, giving the programmer the tool of regex leaves them vulnerable to unwittingly writing SL regexes, a concept relatively foreign to many programmers. Conceivably, warnings could be issued for detected SL regexes during run or compile time. However, static SL detection tools are heuristics which yield false positives and negatives [13], unsuitable for language-integrated warning generation.

Other potential mitigations could be to run regular expression queries on a separate thread so that the main process is not entirely stunted by an explosive regex search. However, this makes regex engine computation as a whole suffer, especially for simpler regexes. Limiting the length of the input, while a possible reasonably effective mitigation, is not generally desirable, as the size limit required may be too short to meet program requirements [12]. Further separating regular expressions into two separate engine (one that supports backtracking and one that avoids it) puts a significant mental burden on the programmer to analyze every regular expression they write for use of backtracking.

In the second case of protestware and related vulnerabilities, having a centralized library repository and library import syntax within a language is the ultimate convenience which can inevitably lead to introducing vulnerabilities. Although software inherits the security vulnerabilities of the third-party software it uses, it is becoming increasingly clear that the main issue in both of these cases lies with *trust*, trust in the language features given to the programmer. In our two cases, the issue is the trust placed in the capacities of regular expressions, and in the third-party library dependency chain of supply.

Library repository maintainers could maintain trust by engaging in code reviews of their own. However, this solution scales incredibly poorly when faced with the volume any remotely popular library receives code and code updates; the alternative of using a selective process of reviewing only those libraries that are the most used discourages the usage of nicher third-party software. It is unclear what can even be done on the language front to address what is primarily an interpersonal problem between programmers. Further research is certainly required, as suggested by [20], as to the best ways to implement automatic protestware detection.

This section showcases the necessity of programmer discipline and that not all security issues are rectifiable at the level of language design. While language designers can keep security in mind when designing a language and its features, by virtue of wanting to give programmers powerful and convenient tools, it may not be possibly, at least not without much difficulty and research, to prevent programmers from performing insecure actions.

5 Conclusion

I have surveyed language-specific features as well as language-agnostic features, and how the vulnerabilities incurred through their usage can be mitigated from the language design side. In some cases, as with `gets`, vulnerabilities are entirely fixable on the language designer's part, although in practice this may conflict with other language design goals, such as backwards compatibility. In yet other cases, other language design goals end up trumping security concerns, leaving vulnerable PLFs in languages. Last, in many cases, some vulnerabilities that initially arise due to a programming language feature seem to be irreparable on the language design front without entirely removing what makes those features useful, as seen with the existence of ReDoS attacks and the corresponding limited mitigations outside of programmer discipline and awareness.

I also introduced three language design maxims: Security-convenience alignment, the concept of making security and convenience align as much as possible in PLFs, Security consciousness, the concept of avoiding vulnerable PLFs either

in design or in language choice, and Programmer-first safety, the concept of not letting programmers commit disastrous mistakes without warning them, or forcing them to take a more intentional route.

A JSFuck program for `document.cookie`

Generated by using the program at [19]. This is a valid JavaScript program that, when executed, returns the value stored in the user's page's `document.cookie`.

```
[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[
])[+[]]][([][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[
]]+(!![]+[])[+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[][(![]+[])[+[]
]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]
+[+[]]]+([][[]]+[])[+!+[]]+(![]+[])[!+[]+!+[]+!+[]]+(!![]+[]
)[+[]]+(!![]+[])[+!+[]]+([][[]]+[])[+[]]+([][(![]+[])[+[]]+(
![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]]+[])[!+[]+
!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[][(![]+[])[+[]]+(![]+[])[!+
[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+[]]]+(!![]
+[])[+!+[]]]((!![]+[])[+!+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![
]+[])[+[]]+([][[]]+[])[+[]]+(!![]+[])[+!+[]]+([][[]]+[])[+!+
[]]+(+[![]]+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!
+[]]+(!![]+[])[+[]]])[+!+[]+[+!+[]]]+(!![]+[])[!+[]+!+[]+!+[
]]+(+(!+[]+!+[]+!+[]+[+!+[]]))[(!![]+[])[+[]]+(!![]+[][(![]+
[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]]
)[+!+[]+[+[]]]+([]+[])[([][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]
+(![]+[])[+!+[]]+(!![]+[])[+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[
][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[]
)[+[]]])[+!+[]+[+[]]]+([][[]]+[])[+!+[]]+(![]+[])[!+[]+!+[]+
!+[]]+(!![]+[])[+[]]+(!![]+[])[+!+[]]+([][[]]+[])[+[]]+([][(
![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+
[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[][(![]+[])[+[
]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[
]+[+[]]]+(!![]+[])[+!+[]]][([][[]]+[])[+!+[]]+(![]+[])[+!+[]
]+((+[])[([][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[
]]+(!![]+[])[+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[][(![]+[])[+[]
]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]
+[+[]]]+([][[]]+[])[+!+[]]+(![]+[])[!+[]+!+[]+!+[]]+(!![]+[]
)[+[]]+(!![]+[])[+!+[]]+([][[]]+[])[+[]]+([][(![]+[])[+[]]+(
![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]]+[])[!+[]+
!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[][(![]+[])[+[]]+(![]+[])[!+
[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+[]]]+(!![]
+[])[+!+[]]]+[])[+!+[]+[+!+[]]]+(!![]+[])[!+[]+!+[]+!+[]]]](
!+[]+!+[]+!+[]+[!+[]+!+[]])+(![]+[])[+!+[]]+(![]+[])[!+[]+!+
[]])()(([][[]]+[])[!+[]+!+[]]+(!![]+[][(![]+[])[+[]]+(![]+[]
)[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+[]]]+(
[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[
])[+[]]]+[])[!+[]+!+[]+!+[]]+([][[]]+[])[+[]]+((+[])[([][(![
]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]
]]+[])[!+[]+!+[]+!+[]]+(!![]+[][(![]+[])[+[]]+(![]+[])[!+[]+
!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+[]]]+([][[]]+
[])[+!+[]]+(![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])
[+!+[]]+([][[]]+[])[+[]]+([][(![]+[])[+[]]+(![]+[])[!+[]+!+[
]]+(![]+[])[+!+[]]+(!![]+[])[+[]]]+[])[!+[]+!+[]+!+[]]+(!![]
+[])[+[]]+(!![]+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[]
)[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+[]]]+(!![]+[])[+!+[]]]+[])
[+!+[]+[+!+[]]]+(!![]+[])[!+[]+!+[]+!+[]]+([][[]]+[])[+!+[]]
+(!![]+[])[+[]]+(+(+!+[]+[+!+[]]+(!![]+[])[!+[]+!+[]+!+[]]+[
!+[]+!+[]]+[+[]])+[])[+!+[]]+([][(![]+[])[+[]]+(![]+[])[!+[]
+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]]+[])[!+[]+!+[]+!+[]]+(
!![]+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!
![]+[])[+[]]])[+!+[]+[+[]]]+(!![]+[][(![]+[])[+[]]+(![]+[])[
!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+[]]]+(+(
!+[]+!+[]+[+[]]))[(!![]+[])[+[]]+(!![]+[][(![]+[])[+[]]+(![]
+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+[]]
]+([]+[])[([][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+
[]]+(!![]+[])[+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[][(![]+[])[+[
]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[
]+[+[]]]+([][[]]+[])[+!+[]]+(![]+[])[!+[]+!+[]+!+[]]+(!![]+[
])[+[]]+(!![]+[])[+!+[]]+([][[]]+[])[+[]]+([][(![]+[])[+[]]+
(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]]+[])[!+[]
+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[][(![]+[])[+[]]+(![]+[])[!
+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+[]]]+(!![
]+[])[+!+[]]][([][[]]+[])[+!+[]]+(![]+[])[+!+[]]+((+[])[([][
(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[
+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[][(![]+[])[+[]]+(![]+[])[!+
[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+[]]]+([][[
]]+[])[+!+[]]+(![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+
[])[+!+[]]+([][[]]+[])[+[]]+([][(![]+[])[+[]]+(![]+[])[!+[]+
!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]]+[])[!+[]+!+[]+!+[]]+(!
![]+[])[+[]]+(!![]+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]
+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+[]]]+(!![]+[])[+!+[]]]+
[])[+!+[]+[+!+[]]]+(!![]+[])[!+[]+!+[]+!+[]]]](!+[]+!+[]+[+!
+[]])+([![]]+[][[]])[+!+[]+[+[]]]+(!![]+[])[!+[]+!+[]+!+[]])
```

Acknowledgments

To Luke, for helping me find the word "idiomatic".

References

[1] 2022. About npm. https://www.npmjs.com/about

[2] 2022. About RubyGems.org. https://rubygems.org/pages/about

[3] 2022. Clang: a C language family frontend for LLVM. https://gcc.gnu.org/

[4] 2022. Computer Programmers: Occupational Outlook Handbook. https://www.bls.gov/ooh/computer-and-information-technology/computer-programmers.htm

[5] 2022. CVE-2022-23812. Available from MITRE, CVE-ID CVE-2022-23812.. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-23812

[6] 2022. GCC, the GNU Compiler Collection. https://gcc.gnu.org/

[7] 2022. node-ipc. https://github.com/RIAEvangelist/node-ipc

[8] 2022. pip 22.3.1. https://pypi.org/project/pip/

[9] 2022. Software Developers, Quality Assurance Analysts, and Testers: Occupational Outlook Handbook. https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm

[10] Fabrice Bellard, grischka, and Thomas Preud'homme. 2002. Tiny C Compiler - Summary. http://savannah.nongnu.org/projects/tinycc

[11] Erik Corry, Christian Plesner Hansen, and Lasse Reichstein Holst Nielsen. 2009. Irregexp, Google Chrome's New Regexp Implementation. https://blog.chromium.org/2009/02/irregexp-google-chromes-new-regexp.html

[12] James C Davis, Christy A Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. In *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 246–256.

[13] James C Davis, Francisco Servant, and Dongyoon Lee. 2021. Using selective memoization to defeat regular expression denial of service (ReDoS). In *2021 IEEE symposium on security and privacy (SP)*. IEEE, 1–17.

[14] Jos de Jong. 2022. math.js: An extensive math library for JavaScript and Node.js. https://mathjs.org/

[15] Mark. Dowd. 2007. *The art of software security assessment : identifying and preventing software vulnerabilities* (1st edition ed.). Addison-Wesley, Indianapolis, Ind.

[16] Raphael A. Finkel. 1996. *Advanced programming language design*. Addison-Wesley Reading.

[17] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. 2017. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)* 50, 4 (2017), 1–36.

[18] ISO IEC. 2010. WG14 N1539 Committee Draft ISO/IEC 9899: 201x. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf

[19] Martin Kleppe. 2022. JSFuck. http://www.jsfuck.com/

[20] Raula Gaikovina Kula and Christoph Treude. 2022. In war and peace: the impact of world politics on software ecosystems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1600–1604.

[21] David Larochelle and David Evans. 2001. Statically detecting likely buffer overflow vulnerabilities. In *10th USENIX Security Symposium (USENIX Security 01)*.

[22] Kyung-Suk Lhee and Steve J Chapin. 2003. Buffer overflow and format string overflow vulnerabilities. *Software: practice and experience* 33, 5 (2003), 423–460.

[23] Bingchang Liu, Liang Shi, Zhuhua Cai, and Min Li. 2012. Software vulnerability discovery techniques: A survey. In *2012 fourth international conference on multimedia information networking and security*. IEEE, 152–156.

[24] Conor O'Brien. 2022. Flowo: A simple, extensible testing language. https://github.com/limitlessSocks/flowo

[25] Yixin Qiu, Anandasivam Gopal, and Il-Horn Hann. 2017. Logic pluralism in mobile platform ecosystems: A study of indie app developers on the iOS app store. *Information Systems Research* 28, 2 (2017), 225–249.

[26] Gerardo Richarte et al. 2002. Four different tricks to bypass stackshield and stackguard protection. *World Wide Web* 1, 7 (2002).

[27] Guido van Rossum. 2022. CPython. https://github.com/python/cpython.

[28] Adar Weidman. 2022. Regular expression Denial of Service - ReDoS. https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS