What is a Secure Programming Language?

Cristina Cifuentes
Oracle Labs, Australia
cristina.cifuentes@oracle.com

Gavin Bierman
Oracle Labs, UK
gavin.bierman@oracle.com

Abstract

Our most sensitive and important software systems are written in programming languages that are inherently insecure, making the security of the systems themselves extremely challenging. It is often said that these systems were written with the best tools available at the time, so over time with newer languages will come more security. But we contend that all of today's mainstream programming languages are insecure, including even the most recent ones that come with claims that they are designed to be "secure". Our real criticism is the lack of a common understanding of what "secure" might mean in the context of programming language design. We propose a simple data-driven definition for a secure programming language: that it provides first-class language support to address the causes for the most common, significant vulnerabilities found in real-world software. To discover what these vulnerabilities actually are, we have analysed the National Vulnerability Database and devised a novel categorisation of the software defects reported in the database. This leads us to propose three broad categories, which account for over 50% of all reported software vulnerabilities, that *as a minimum* any secure language should address. While most mainstream languages address at least one of these categories, interestingly, we find that none address all three.

Looking at today's real-world software systems, we observe a paradigm shift in design and implementation towards service-oriented architectures, such as microservices. Such systems consist of many fine-grained processes, typically implemented in multiple languages, that communicate over the network using simple web-based protocols, often relying on multiple software environments such as databases. In traditional software systems, these features are the most common locations for security vulnerabilities, and so are often kept internal to the system. In microservice systems, these features are no longer internal but external, and now represent the attack surface of the software system as a whole. The need for secure programming languages is probably greater now than it has ever been.

1   Analysing software vulnerabilities

The National Vulnerability Database (NVD) is the US government repository of standards-based vulnerability management data. It takes as its data feed the vulnerabilities given in the Common Vulnerabilities and Exposures (CVE) directory, and performs various analyses to determine impact metrics (CVSS), vulnerability types (CWE), and applicability statements (CPE), and other useful metadata.

For the past few decades, this data has been extensively analysed, aggregated and categorised, and gives probably the best insight into the most common security issues facing the software industry. In 1995, the NVD contained only 25 entries; in 2017, this had ballooned to more than 10,000 entries for that year alone. The software industry clearly faces a serious problem. Moreover, there is strong evidence that this is an *underestimation* of the actual problem. For example, many cloud software defects are not even assigned a CVE, because cloud software is typically architected to be continuously updated, making it difficult to be tracked by services such as the CVE list.

The main technical contributions of this paper are an analysis of the labelled data in the NVD for the five years from 2013 to 2017, where we have normalised and aggregated the data related to source code vulnerabilities into broad categories, and a more precise and meaningful definition of a *secure* language.

In our analysis of the NVD data, interestingly, we discovered that three of the top four most common vulnerabilities are actually issues that can be considered to be in the realm of programming language design. Moreover, when combined, these three categories of vulnerabilities represent 53% of all labelled exploited vulnerabilities listed in the NVD for that period. The complete analysis is presented in Appendix A. The three categories and the number of reported vulnerabilities they represent are as follows:

- 5,899 buffer errors
- 5,851 injection errors[1]
- 3,106 information leak errors.

It is a little depressing that two of these vulnerability categories are very old. The Morris worm [16] exploited a buffer error in the Unix finger server in 1988 – over 30 years ago. SQL injections [14] and XSS exploits [2] have been documented in the literature since 1998 and 2000, respectively.

A 2018 study [9] revealed that the average total cost of a data breach is US$3.86 million, with an average cost per record of US$157 due to hacker or criminal attacks. With all the additional costs of software defects, including reputational loss, productivity loss, among many others, there is an obvious incentive to try to address more than 50% of vulnerabilities by better programming language design.

This paper is organised as follows. In §2 we give the definition and an example of each of the three types of vulnerabilities discussed in this paper. In §3 we take a look at mainstream languages, and consider their support for the three categories of software vulnerabilities. We consider this problem abstractly, examining the trade-offs when using new abstractions provided by a programming language. In §4 we look at the common programming language abstractions to address buffer overflows. In §5 we look at the common programming language abstractions to address the most frequently occurring forms of injection error. Finally, in §6 we consider the issue of information leak errors, and point to a promising technique from the research community. We conclude in §7, and give details of our categorisation of the CWE enumerations used in the NVD, and analysis of recent five years of NVD data in Appendix A.

---

[1] Injection errors include Cross-Site Scripting (XSS), SQL injection, Code injection, and OS command injection.

2 The Three Categories of Vulnerabilities

In this paper we focus on three vulnerabilities that are widely exploited year over year, and yet, those vulnerabilities could be prevented through programming language design. Throughout this section we make use of NIST's Computer Security Resource Center definitions [4] and CWE examples from Mitre [5].

2.1 Buffer Errors

A buffer overflow attack is a method of overloading a predefined amount of memory storage in a buffer, which can potentially overwrite and corrupt memory beyond the buffer's boundaries. Buffer overflow errors can be stack-based or heap-based, depending on the location of the memory that the buffer uses.

An example of a stack-based buffer overflow, CWE-121, follows. In this C code, the function `host_lookup` allocates a 64-byte buffer, `hostname`, to store a hostname. However, there is no guarantee in the code that the hostname will not be larger than 64 bytes, and hence, this is a vulnerability. An attacker can attack this vulnerability by supplying an address that resolves to a large hostname, overwriting sensitive data or even relinquishing control to the attacker:

```
void host_lookup ( char *user_supplied_addr ){
struct hostent *hp;
in_addr_t *addr;
char hostname [64];
in_addr_t inet_addr ( const char *cp );

/*routine that ensures user_supplied_addr is in the right format
 for conversion */

validate_addr_form ( user_supplied_addr );
addr = inet_addr ( user_supplied_addr );
hp = gethostbyaddr ( addr , sizeof ( struct in_addr), AF_INET );
strcpy ( hostname , hp->h_name );
}
```

A canonical example of a heap-based buffer overflow, CWE-122, follows. In this C code, a heap-based buffer `buf` is allocated of a given size. When a copy of the string `argv[1]` into that buffer is made, there is no check on the size of the buffer, leading to an overflow for strings larger than the allocated size. This is a vulnerability because the user/attacker has control over the string `argv[1]`.

```
#define BUFSIZE 256
int main ( int argc , char **argv ) {
char *buf;
buf = ( char *) malloc ( sizeof ( char )* BUFSIZE );
strcpy (buf , argv [1]);
}
```

2.2 Injection Errors

Injection errors include several types of vulnerabilities, the most common ones being: cross-site scripting (XSS), SQL injection, code injection, and OS command injection. We provide examples for the first two types of vulnerabilities.

Cross-site scripting is a vulnerability that allows attackers to inject malicious code into an otherwise benign website. These scripts acquire the permissions of scripts generated by the target website and can therefore potentially compromise the confidentiality and integrity of data transfers between the website and client. Websites are vulnerable if they display user-supplied data from requests or forms without sanitising the data so that it is not executable. There are three main kinds of XSS, all part of CWE-79: reflected XSS (or non-persistent); stored XSS (persistent); and DOM-based XSS.

An example of a reflected XSS follows. The JSP code fragment reads an employee ID, `eid`, from an HTTP request and displays it to the user without sanitising the employee ID. This vulnerability can be exploited by an attacker by including meta-characters or source code in the input, then that code will be executed by the web browser as it displays the HTTP response.

```
<% String eid = request.getParameter("eid"); %>
...
Employee ID: <%= eid %>
```

An example of a stored XSS follows. The JSP code fragment queries a database for a particular employee ID and prints the corresponding employee's name. This is a vulnerability if the value of the name originates from user-supplied data and has not been validated. As such, without proper input validation on all data stored in the database, this vulnerability can be exploited by an attacker by executing malicious commands on the user's web browser.

```
<%Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select * from emp where id="+eid);
if (rs != null) {
 rs.next();
 String name = rs.getString("name");
}%>

Employee Name: <%= name %>
```

An SQL injection vulnerability is one that allows attackers to execute arbitrary SQL code on a database back-end. A canonical example of an SQL injection vulnerability, CWE-89, follows. In this $C^\sharp$ code fragment, an SQL query is dynamically constructed that searches for items matching a currently authenticated user name. Because the query concatenates a user-defined string, `ItemName.Text`, the query will behave correctly only if the item name does not contain a single-quote character. This vulnerability is exploited by an attacker with the user name wiley by entering an item name such as `name' OR 'a'='a`, leading to the SQL query `SELECT * FROM items WHERE owner = 'wiley' AND itemname = 'name' OR 'a'='a';`, which effectively becomes `SELECT * FROM items;` as the `WHERE` clause evaluates to true. Even worse, if the SQL API supports multiple statements, then a malicious user would enter an item name such as `' OR 1=1; DROP TABLE users; SELECT * FROM secretTable WHERE 't' = 't` that would delete the table of users and extract all the details from a secret table.

```
...
string userName = ctx.getAuthenticatedUserName();
string query = "SELECT * FROM items WHERE owner = '" + userName +
 "' AND itemname = '" + ItemName.Text + "'";
sda = new SqlDataAdapter(query, conn);
DataTable dt = new DataTable();
sda.Fill(dt);
...
```

2.3 Information Leak Errors

Information leakage is the intentional or unintentional release of information to an untrusted environment. There are many forms of information leakage in today's software systems. Information can be leaked through log files, caching, environment variables, test code, shell error messages, servlet runtime error messages, Java runtime error messages, and more. The most common type of information leak is through log files (whether debug logs, server logs, or other).

An example of an information leak through log files, CWE-532, follows. In this Java code fragment, the `this` object contains the location of the user. When the application encounters an exception, it will write the user object to the log, including the location information.

```
locationClient = new LocationClient(this, this, this);
locationClient.connect();
currentUser.setLocation(locationClient.getLastLocation());
...

catch (Exception e) {
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setMessage("Sorry,␣this␣app␣has␣experienced␣an␣error.");
AlertDialog alert = builder.create();
alert.show();
Log.e("ExampleActivity", "Caught␣exception:␣" + e + "␣While␣on␣User:"
 + User.toString());
}
```

3 Mainstream languages and vulnerabilities

For the purposes of this paper, we restrict our attention only to what we term mainstream languages. We use this term a little loosely, and we are certainly not arguing the relative merits of these languages over each other, or any other language for that matter. To be concrete, we used the data from the respected TIOBE Index [17], though interestingly, other similar indexes yield very similar, if not identical, results. According to the TIOBE index of January 2019, the current top 10 mainstream programming languages are: Java, C, Python, C++, Visual Basic .NET, JavaScript, C$^\sharp$, PHP, SQL and Objective-C. However, for this paper, we take the cumulative data for the past 10 years, and therefore we consider a language to be mainstream if it has been in the top 10 for every year of the past 10 years. Accordingly, we take Java, C, C++, Python, C$^\sharp$, PHP, JavaScript and Ruby as the mainstream programming languages to be considered in this paper.

So, we have our mainstream languages, and from our analysis of the NVD we have our three security vulnerability categories. Our thesis is that any secure programming language worthy of its name should be one that has first-class support for all three categories. Put diagrammatically, a secure language is one that lives in the intersection of the following Venn diagram:
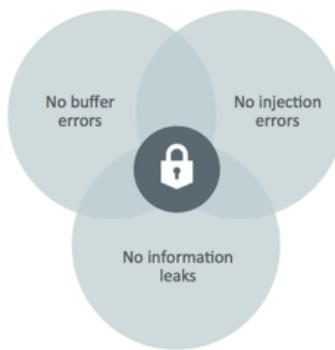
Unfortunately, given this definition, it is the case that *none of our mainstream languages can be considered secure.*

It is worth taking a step back, and considering the causes of vulnerabilities in software. Developers do not write incorrect code because they want to. Vulnerable code is written inadvertently because our mainstream programming languages do not provide the right abstractions to support developers in writing vulnerable-prone code. Buffer errors are introduced because of manual management (allocation, reallocation, and deallocation) of pointers. Injection errors are introduced because of the representation of code as strings, use of manual string concatenation, and sanitisation of strings that reach a sensitive location (e.g., an SQL execute statement). And information leak errors are introduced because of manual tracking of sensitive data, trying to ensure it does not leak to less sensitive objects.

Abstractions in programming languages introduce different levels of cognitive load. The easier it is for an abstraction to be understood, the more accepted that abstraction becomes. For example, managed memory is an abstraction that frees the developer from having to manually keep track of memory (both allocation and deallocation). As such, this abstraction is widely used in a variety of mainstream and non-mainstream languages. At the same time, performance of the developed code is also relevant to developers. If an abstraction introduces a high performance overhead, it makes it hard for that abstraction to be used in practice in some domains. For example, managed memory is not often used in embedded systems or systems programming due to the difficulty of predicting its performance overhead.

The space of these two criteria that needs to be considered when developing safe abstractions can be visualised as follows:
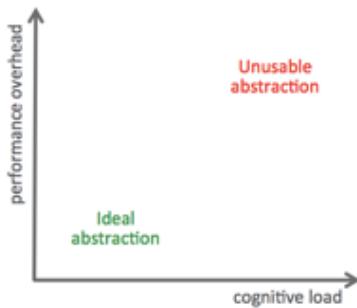
The ideal abstraction is clearly one that has low cognitive load and low performance overhead. By contrast, abstractions that provide a high cognitive load and high performance overhead will never be used in a mainstream programming language. There are abstractions in between that provide a low cognitive load yet a high performance overhead. Those abstractions may be usable in certain domains, as are abstractions that may have a high cognitive load and low performance overhead.

For the issues addressed in this paper, we are interested in abstractions that address the vulnerabilities from our analysis of the NVD. We refer to these abstractions as *safe abstractions*. Managed memory is thus an example of a safe abstraction with respect to buffer errors. In §§4–6 we provide examples of other safe abstractions.

4 Language support addressing buffer overflows

Providing language support that addresses buffer overflows is almost as old as programming language design itself. The traditional technique is to use managed memory, discussed in §4.1. However, this technique has not had great acceptance in the systems community, where concerns about performance overheads, as per the discussion in the previous section, have dominated. In §4.2 we mention an interesting new development addressing buffer overflows using an ownership-inspired type system.

4.1 Managed memory

In 1958, the LISP language introduced the concept of managed memory by adding garbage collection in the runtime of the language. Garbage collection provides a solution that frees up developers from the cognitive load of having to allocate, reallocate and deallocate memory objects correctly. Developers do not need to specify any allocation and deallocation instructions because it is done under the hood by the runtime of the language.

Managed memory took some time to become widely used due to its performance overhead. With the advent of faster computers, it is now widely used by object-oriented languages such as Smalltalk, Java, C$^\sharp$, JavaScript and Go; functional languages such as ML, Haskell and APL; and dynamic languages such as Ruby, Perl and PHP. Note that many of today's mainstream languages use managed memory, however, in some domains, such as embedded systems and operating systems, the performance overhead of a managed memory runtime is still not feasible.

LISP's managed memory abstraction was introduced to free up developers from the cognitive load of allocating and deallocating memory objects manually. At the time, exploitation of buffer errors was not even known, c.f., the Morris worm didn't appear until 1988. As such, this abstraction was not introduced for security, however, it is a great example of a safe abstraction that can be used to provide security against buffer errors and other types of memory-related errors.

4.2 Ownership and borrowing

Rust is a systems programming language introduced in 2009 that runs fast, prevents memory corruption, and is designed to guarantee memory and thread safety, though has not been formally verified yet. Not only does it prevent buffer errors, it prevents various other types of memory corruptions, such as null pointers and use after free. This feature is provided by the introduction of ownership and borrowing into the type system of the language:

- Ownership is a programming pattern employed in C++ and other languages whereby a resource can have only one owner. Ownership with "resource acquisition is initialisation" (RAII) ensures that whenever an object goes out of scope, its destructor is called and its owned resource is freed. Ownership of a resource is transferred (i.e., moved) through assignments or passing arguments by value. When a resource is moved, the previous owner can no longer access it, therefore preventing dangling pointers.

─ Borrowing is an abstraction that allows a reference to a resource to be made available in a secure way – either through a shared borrow (`&T`), where the shared reference cannot be mutated, or a mutable borrow (`&mut T`), where the shared reference cannot be aliased – but not both at the same time. Borrowing allows for data to be used elsewhere in the program without giving up ownership. It prevents use after free and data races.

Ownership and borrowing provide abstractions suitable for memory safety, and prevent buffer errors from happening in the code. Anecdotal evidence seems to suggest that the learning curve to become fluent in the use of these abstractions can be long, pointing to a high cognitive load. Nevertheless, given its low performance overhead, this safe abstraction with respect to memory-related errors is an exciting development in the systems programming domain.

5 Language support addressing injection errors

No mainstream language offers a general safe abstraction for injection errors. .NET offers support for language-level query abstractions, including queries over a relational database. This removes the possibility of SQL injection errors, which we consider in more detail in §5.1. Although Perl is not strictly a mainstream language by our criteria, in §5.2 we consider its taint tracking supported, that was later extended in Ruby (which is a mainstream language by our criteria). Taint tracking could form the basis of a more general abstraction to address injection errors.

5.1 LINQ to SQL

Language INtegrated Query (LINQ) is a framework introduced in 2007 by Microsoft for .NET 3.5 that adds language-level query facilities to the .NET languages. It extends the languages by adding first-class query expressions (along with some supporting extensions to the type system and other language extensions) that can be used to extract and process data from any source that supports a predefined set of methods, known as the standard query operator API. A data type that supports this API is known as a LINQ provider. By default, all arrays and enumerable classes are LINQ providers. This allows developers to write high-level declarative queries over their in-memory collections.

Moreover, .NET also provides other implementations of LINQ providers, in particular, LINQ-to-SQL. This allows queries to be written in a .NET language and be translated into a semantically equivalent query that executes on an SQL engine. In addition, it comes with an object-relational mapping framework that automatically generates strongly typed .NET class declarations that correspond to tables in the database. LINQ-to-SQL also solves the SQL injection problem because the programmer no longer constructs SQL code using strings and string concatenation (the source of SQL injection errors) but uses the language-level queries, which themselves pass all data to the database using injection-safe SQL parameters.

LINQ-to-SQL allows for a large percentage of SQL queries to be written in a simpler language, but not all SQL queries can be represented. A number of more advanced SQL queries, for example, that involve selecting into temporary tables and then querying those tables, predicated updates and bulk inserts, and triggers, are not supported. Vendor-specific extensions to SQL are also not supported. These features have to be accessed through stored procedures that have to be defined on the database side.

5.2 Taint tracking

Perl is a rapid prototyping language introduced in 1987. In 1989, Perl 3 introduced the concept of taint mode to track external input values (which are considered tainted by default), and to perform runtime taint checks to prevent direct or indirect use of a tainted value in any command that invokes a sub-shell, or any command that modifies files/directories/processes, except for arguments to `print` and `syswrite`, symbolic methods and symbolic subreferences, or hash keys. Default tainted values include all command-line arguments, environment variables, locale information, results of some system calls (`readdir()`, `readlink()`), etc. Importantly, taint mode is supported in the runtime of the language. Perl 5 supports taint mode, but Perl 6 does not. Note that because there is no tracking of taint to `print` statements, cross-site scripting attacks on web applications are still possible.

Ruby is a dynamic programming language developed in 1993 with a focus on simplicity and productivity. It supports multiple programming paradigms, including functional, object-oriented, imperative, and reflective. Ruby extends Perl's taint mode to provide more flexibility. There are four safe levels available, of which the first two are as per Perl, as follows:

0. No safety.
1. Disallows use of tainted data by potentially dangerous operations. This level is the default on Unix systems when running Ruby scripts as `setuid`.
2. Prohibits loading of program files from globally writable locations.
3. Considers all newly created objects to be tainted.

In Ruby, every object has a `Trusted` flag. There are methods to make an object tainted, check whether the object is tainted, or untaint the object (only for levels 0–2). At runtime, Ruby tracks direct data flows through levels 1–3; it does not track indirect/implicit data flows.

The taint tracking abstraction provides a way to prevent some types of injection errors with low cognitive load on developers. Apart from Perl and Ruby, PHP [18] and version 1.1. of JavaScript implemented taint tracking in the runtime. Trade-offs in performance overhead need to be made to determine how much data can be tracked, what target locations should be tracked, and whether direct and indirect uses can be tracked. Livshits [10] suggests several reasons why taint tracking is still an unsolved problem including the following: predictable performance needs; whether control flow tracking is needed; value tracking of data that may be safe though deemed to be potentially tainted at runtime; determining declassifiers; specification inference at runtime; and configurable runtime support for taint. Proposals for reducing performance overhead include combining the approach with static analysis. Further, for some applications, values need to be tracked not only for strings, but also for primitive values, collections, or other objects, resulting in performance overheads.

6 Language support addressing information leak errors

As far as we are aware, there is no language-level feature in any mainstream language to address vulnerabilities resulting from information leak errors. This is clearly an area requiring further research. We briefly review one interesting approach from the research community, although other approaches such as tracking sensitive data [6], data shadowing of sensitive data [8], and a DSL for data-centric applications [13] are being pursued.

6.1 Faceted values

Jeeves is an experimental academic language for automatically enforcing information flow policies, first released in 2014 [21]. It is implemented as an embedded DSL in Python. Jeeves makes use of the faceted values abstraction, which is a data type used for sensitive values

that stores within it the secret `s` (high-confidentiality) and non-secret `ns` (low-confidentiality) values, guarded by a policy `p`, e.g., `<s | ns> (p)`. A developer specifies policies outside the code, making the code policy-agnostic, and the language runtime enforces the policy by guaranteeing that a secret value may flow to a viewer only if the policies allow the viewer to view secret data.

Many applications today make use of a database. To make the language practical, faceted values need to be introduced into the database when dealing with database-backed applications. A faceted record is one that guards a secret and non-secret pair of values. Jacqueline, a web framework developed to support faceted values in databases, automatically reads and writes metadata in the database to manage relevant faceted records. The developer can use standard SQL databases through the Jacqueline object relational mapping.

The faceted values abstraction provides a way to prevent information leaks with low cognitive load on developers, at the expense of performance overhead. This work is yet to determine the lower bound on performance overhead, in order to provide direct and indirect tracking of the data flows for leaks of sensitive data purposes.

We note that faceted values could also be used to prevent injection errors, as shown by the DroidFace implementation for an intermediate Java-like language [15]. Conversely, taint tracking can also be used to track sensitive user data leaving an application through a network, file system, or similar [6].

7   Concluding Remarks and Further Work

The mainstream languages over the past 10 years are Java, C, C++, Python, C$^\sharp$, PHP, JavaScript and Ruby. In this paper we have considered these languages with respect to their support for features that directly address the sorts of vulnerabilities in real-world software systems.

To identify these vulnerabilities, we have analysed the labelled data in the NVD for the five-year period of 2013–2017. Employing a novel categorisation, we have identified that three of the top four categories of vulnerabilities for that period actually represent issues that can be considered to be in the realm of programming language design; namely, prevention of buffer errors, injection errors, and information leak errors.

We observe that none of today's mainstream languages provide safe abstractions that address all three of these prominent types of exploited vulnerabilities. By our criteria, we claim that none of our mainstream languages can be considered secure.

Buffer errors are addressed by all mainstream languages, apart from C, C++, and PHP, through use of the managed memory. We note that taint tracking is in place in only the Ruby and PHP (through an extension) mainstream languages to avoid injection errors, and that some classes of injection errors have been addressed by LINQ for the .NET languages. Taint tracking solutions are not yet mature enough to capture all aspects of what needs to be tracked with a good performance trade-off. As far as we are aware, no mainstream language supports a safe abstraction to deal with information leak errors.

We note that a couple of upcoming languages, Rust and Pony [3], have designed their languages to not only avoid buffer errors, but also concurrency (data race) errors. They use different safe abstractions for data race prevention, and neither provides abstractions to prevent injection or information leak errors.

Many languages today provide a foreign function interface (FFI) to "escape" into another language that normally provides other properties, such as performance or low-level system access. Such an FFI escape hatch breaks any concept of a safe abstraction in the language because the compiler and/or runtime does not track information across the language boundaries.

Ideally, in order to help developers in writing vulnerability-prone code, we need to provide safe abstractions that complement existing mainstream languages, because it is impractical to migrate the millions of lines of code already written and relied upon by millions of developers. Indeed, much of the existing software is developed in languages that are distinctly *in*secure! Two interesting and somewhat orthogonal approaches to this problem are being explored in the research community:

1. One approach is to explore multilingual compilers and runtimes that have been architected to simultaneously support multiple languages, with differing security abstractions (e.g. [20]). More foundational work in this area is also encouraging [12].
2. Another approach is to maintain a monolingual runtime but employ possibly many stages of secure compilation; that is, compilation that preserves security properties via translations that are fully abstract (e.g. [1, 7]).

Further work also involves continued analysis of the vulnerability data. While we have paid attention to the largest categories, and hope that they will be addressed, the remaining categories will become increasingly more relevant. Put another way, these other vulnerability categories should be drivers for programming language design of the future.

Finally, we conclude by drawing attention to the significant change in the design of real-world systems from large, standalone software systems to software-as-a-service. This reflects both the change in underlying infrastructure from in-house systems to rented cloud platforms, and also the requirements of systems to seamlessly scale up/down. Many existing applications, as well as new applications, are being re-architected using quite radically different design patterns, such as microservices (functions-as-a-service). While it should be expected that familiar vulnerabilities will be reported from these applications, it is entirely reasonable to expect that either new vulnerabilities will emerge, or existing ones will become much more common. There is some work on security aspects of microservice-style programming – for example, Whip [19] proposes a formal contracts system for microservices code that prevents software errors at the edge – but it is clear that further research is needed, for example, how to prevent second-order SQL injections in a microservice.

References

1. Martín Abadi. Protection in Programming-Language Translations. In *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*, pages 19–34, 1999. `doi:10.1007/3-540-48749-2_2`.
2. CERT. Malicious HTML Tags, 2000.
3. Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny Capabilities for Safe, Fast Actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2015, pages 1–12, New York, NY, USA, 2015. ACM. `doi:10.1145/2824815.2824816`.
4. NIST Computer Security Resource Center – Glossary, Last accessed April 8, 2019. URL: `https://csrc.nist.gov/Glossary`.
5. Common Weakness Enumeration – CWE list version 3.2, Last accessed April 8, 2019. URL: `https://cwe.mitre.org/data/index.html`.
6. William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 393–407, Berkeley, CA, USA, 2010. USENIX Association. URL: `http://dl.acm.org/citation.cfm?id=1924943.1924971`.
7. Cédric Fournet, Nikhil Swamy, Juan Chen, Pierre-Évariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to JavaScript. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 371–384, 2013. `doi:10.1145/2429069.2429114`.

8   Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These Aren't the Droids You'Re Looking for: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 639–652, New York, NY, USA, 2011. ACM. `doi: 10.1145/2046707.2046780`.

9   Ponemon Institute. 2018 Cost of Data Breach Study: Global Overview, July 2018. URL: `https://databreachcalculator.mybluemix.net/assets/2018_Global_Cost_of_a_Data_Breach_Report.pdf`.

10   Benjamin Livshits. Dynamic Taint Tracking in Managed Runtimes. Technical Report MSR-TR-2012-114, Microsoft Research, November 2012.

11   National Vulnerability Database – NVD CWE Slice, Last accessed February 7, 2019. URL: `https://nvd.nist.gov/vuln/categories`.

12   Daniel Patterson and Amal Ahmed. Linking Types for Multi-Language Software: Have Your Cake and Eat It Too. *CoRR*, abs/1711.04559, 2017. `arXiv:1711.04559`.

13   Nadia Polikarpova, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. Enforcing Information Flow Policies with Type-Targeted Program Synthesis, 2018. `arXiv:1607.03445v2`.

14   rain.forest.puppy. NT web technology vulnerabilities. *Phrack Magazine*, 8(54), 1998. URL: `http://www.phrack.org/archives/issues/54/8.txt`.

15   Daniel Schoepe, Musard Balliu, Frank Piessens, and Andrei Sabelfeld. Let's Face It: Faceted Values for Taint Tracking. In *ESORICS (1)*, volume 9878 of *Lecture Notes in Computer Science*, pages 561–580. Springer, 2016.

16   E. H. Spafford. Crisis and Aftermath. *Commun. ACM*, 32(6):678–687, June 1989. `doi: 10.1145/63526.63527`.

17   TIOBE index, February 2019. URL: `http://www.tiobe.com/tiobe-index`.

18   Wietse Venema. Taint Support for PHP, Last modified: 2017/09/22. URL: `https://wiki.php.net/rfc/taint`.

19   Lucas Waye, Stephen Chong, and Christos Dimoulas. Whip: higher-order contracts for modern services. *PACMPL*, 1(ICFP):36:1–36:28, 2017. `doi:10.1145/3110280`.

20   Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 662–676, New York, NY, USA, 2017. ACM. `doi:10.1145/3062341.3062381`.

21   Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, Dynamic Information Flow for Database-backed Applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 631–647, New York, NY, USA, 2016. ACM. `doi: 10.1145/2908080.2908098`.

22   Yves Younan. 25 Years of Vulnerabilites: 1988–2012. Research report, Sourcefire, 2013.

A   Appendix – NVD data analysis

CWE Classification

The Common Weakness Enumeration (CWE) [11] specification provides a classification of software security vulnerabilities as they are found in source code, design, or system architecture. An individual CWE represents a single vulnerability type. CWEs are organised in a hierarchical structure – CWEs at the highest levels of the structure provide an overview of a vulnerability category (e.g., "data handling"), and the leaves of the tree represent actual vulnerability types (e.g., "out-of-bounds read", "SQL injection", "cross-site scripting"). The National Vulnerability Database makes use of 34 CWEs for its classification of the various reported CVEs. We focus on the ones related to source code errors.

The first 25 years of vulnerability data, from 1988 to 2012, were collated and analysed by Sourcefire [22]. From September 2007, the NVD changed its methodology and adopted 19 CWE categories to classify vulnerabilities, but chose not to reclassify the earlier ones. Sourcefire normalised the data by manually mapping from the old categories to the 19 new ones. NVD also makes use of two CWEs that do not provide information for classification purposes and are not counted towards the 19. They are "Insufficient Information" and "Other". The 19 CWE categories are:

1. Buffer Errors
2. SQL Injection
3. Information Leak
4. Configuration
5. Credentials
6. Link Following
7. Race Conditions
8. Cross-Site Scripting
9. Input Validation
10. Resource Management
11. Numeric Errors
12. Crypto
13. OS Command Injection
14. Access Control
15. Code Injection
16. Path Traversal
17. Authentication
18. CSRF
19. Format String

Figure 1 shows the subtree of the CWE tree hierarchy that represents vulnerabilities in the source code. As can be seen, there are four main CWE categories: (1) indicator of poor code quality, (2) data handling, (3) security features, and (4) time and state.

Data for 2013–2017

We took the data from the NVD for the years 2013 to 2017 and recategorised the 32 CWEs that have been used since July 2016 into the 19 CWE NVD categories of 2007 because the 12 new CWEs are all inner nodes of the CWE tree hierarchy. As such, they represent a subcategory of vulnerabilities rather than a vulnerability type *per se*. There are 16 leaves that are part of the 2007 vulnerability types, with two others ("input validation", and "path traversal") that are inner nodes of the subtree, and one other ("configuration") that is not part of the source code CWE subtree hierarchy.

The summary data from NVD for the years 2013–2017, categorised into 19 categories after removing the entries "Not Enough Information" and "Other", is as follows:

```
1- CWE-398: Indicator of poor code quality
 CWE-399: Resource management errors
2- CWE-19: Data handling
 CWE-20: Input validation
 CWE-119: Buffer errors
 CWE-134: Format string vulnerability
 CWE-74: Injection
 CWE-77: Command injection
 CWE-78: OS command injection
 CWE-89: SQL injection
 CWE-79: XSS
 CWE-94: Code injection
 CWE-21: Path equivalence
 CWE-22: Path traversal
 CWE-59: Link following
 CWE-189: Numeric errors
 CWE-129: Improper validation of array index
 CWE-190: Integer overflow or wraparound
 CWE-682: Incorrect calculation
 CWE-190: Integer overflow or wraparound
 CWE-191: Integer underflow
 CWE-369: Divide by zero
 CWE-199: Information management errors
 CWE-200: Information Leak
3- CWE-254: Security features
 CWE-287: Authentication issues
 CWE-345: Insufficient verification of data authenticity
 CWE-352: CSRF
 CWE-255: Credentials management
 CWE-264: Permissions, privileges and access control
 CWE-284: Improper access control
 CWE-287: Authentication issues
 CWE-310: Cryptographic issues
4- CWE-361: Time and state
 CWE-362: Race conditions
```

Figure 1 Subtree of the CWE tree hierarchy that focuses on source code.

Vulnerability Type	Number of Unique Exploits
Buffer errors	5899
Permissions, privileges and access control	3951
Cross-site scripting	3914
Information leak	3106
Input validation	2618
Cryptographic issues	1873
Resource management	1422
SQL injection	1299
Cross-site request forgery	1062
Path traversal	690
Authentication	485
Code injection	450
Numeric errors	420
Credentials management	370
Race conditions	256
OS command injection	188
Link following	112
Configuration	38
Format string	29

We aggregated the data from all forms of injection attacks (namely, "cross-site scripting", "SQL injection", "code injection", and "OS command injection") into a single category, and then considered the resulting top four vulnerability categories.

Vulnerability Category	Count
Buffer errors	5899
Injections	5851
Permissions, privileges and access control	3951
Information leak	3106

It is important to note that these top four categories represent **64%** of all labelled data for the five years.

The categories "buffer errors", "injections", and "information leak" represent categories of vulnerabilities that could be prevented through first-class language support. However, the category "Permissions, privileges and access control" represents vulnerabilities in the implementation of a security solution (whether in the implementation of the language solution or through own implementation of a solution in the source code). It remains future work to establish to what degree language-level support could alleviate vulnerabilities in this category. Accordingly, we have focused our attention in this paper on the other three categories, which still represent **53%** of all labelled data in the NVD for the recent five years of 2013–2017.