New ISO/IEC Technical Report Describes Vulnerabilities in Programming Languages

James W. Moore, The MITRE Corporation
John Benito, Blue Pilot
Larry Wagoner, National Security Agency

Abstract. A recent joint technical report from two major international standards bodies, the International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), identifies classes of vulnerabilities in programming languages—those features of the languages that encourage or permit the writing of code that contains application vulnerabilities—and suggests ways to avoid or mitigate them. According to the report, programming language vulnerabilities should especially be avoided "in the development of systems where assured behavior is required for security, safety, mission critical and business critical software. [However], this guidance is applicable to the software developed, reviewed, or maintained for any application." This paper provides a brief summary of the ISO/IEC Technical Report.

Introduction

The necessity of mitigating vulnerabilities in software applications is well understood by organizations today. To identify them in existing applications, organizations can use vendor alerts along with public resources such as the Common Vulnerabilities and Exposures [1] and the Open Web Application Security Project's Top 10 Web Application Security Flaws [2] lists. Programmers can help to avoid including them in new applications or maintenance of existing applications by consulting the public Common Weakness Enumeration (CWE™) [3] and CWE/SANS Top 25 Most Dangerous Software Errors [4] lists. Other organizations (e.g., CERT [5] and MISRA [6]) have developed public or private style guides to assist programmers in avoiding application vulnerabilities.

An application vulnerability is a weakness in a software application that permits exploitation by unauthorized persons or contributes to safety hazards. The frequent patches provided by our software vendors have alerted most of us to the problem of vulnerabilities in software designs. Not as well known, however, is that the programming languages in which software applications are written, also have vulnerabilities of their own that can cause applications not to work as intended, behave in unpredictable ways, or lead to application vulnerabilities. Simply stated, deficiencies in the design of programming languages encourage programmers to code in a manner that creates application vulnerabilities. The consequences for organizations can be costly as well as dangerous.

To address this problem, ISO [7] and IEC [8] issued a Technical Report entitled ISO/IEC TR 24772:2010, Information technology—Programming languages—Guidance to avoiding vulnerabilities in programming languages through language selection and use [9], in September 2010 that lists 51 common types of vulnerabilities found in programming languages, along with suggestions for how to avoid them. The report also lists 20 application vulnerabilities that could be addressed by improved language library routines.

No one language contains all of the vulnerabilities described in the report, but most are very common. In addition, 17 of the vulnerabilities detailed in the report also appear on the 2010 CWE/SANS Top 25 Most Dangerous Software Errors list.

Reduce Risk by Mitigating Programming Language Vulnerabilities

By understanding the different ways in which their programming languages might be vulnerable, writers of language standards can eliminate or reduce those vulnerabilities in their languages and thereby make them more secure. In turn, application developers can know how secure a language is before choosing it. Developers will also be able to ensure that the potential for vulnerabilities in their applications are minimized in their software applications, and that they have chosen the most effective and comprehensive source code evaluation tools. Project managers can use the guide to make better-informed selections of programming languages and establish mitigations for the risks inherent in the chosen language.

This is of special importance to those who develop, maintain, and regulate:

- Safety-critical applications that might cause loss of life, human injury, or damage to the environment.
- Security-critical applications that must ensure properties of confidentiality, integrity, and availability.
- Mission-critical applications that must avoid loss or damage to property or finance.
- Business-critical applications where correct operation is essential to the successful operation of the business.
- Scientific, modeling, and simulation applications that require high confidence in the results of possibly complex, expensive, and extended calculation.

Reducing risk in all of these areas will, over time, yield organizations cost savings due to less work, and ultimately lead to more secure systems.

Types of Programming Language Vulnerabilities

When a programmer writes a software application, regardless of the programming language used—be it Ada, C, COBOL, Fortran, etc.—the code should execute in a manner that can be predicted by the developer. If it does not, and an attacker can then make use of the mistake to access a system or network, it is considered a vulnerability in the software code.

With programming languages, vulnerabilities arise in six main ways:

Incomplete or Evolving Programming Language Behavior

Programming language standards are continuously evolving with new releases and features, resulting in issues that might affect predictability. Such issues include the need for compatibility with previous releases, and the interaction of that language's features, separately and in any combination, under all foreseeable circumstances.

Choice of compiler can also have an effect. Compilers are used by programmers to transform their source code to a binary code (commonly called object code). However, unless the compiler comes from a trusted source and was developed according to agreed standards, it could inadvertently or maliciously insert bad code into the binary, resulting in a potential vulnerability. This is especially important to avoid because this type of vulnerability would then be inserted into every piece of software that the compiler processes.

Unspecified behavior must also be avoided. While most behavior is specified by programming languages, unspecified behaviors can result when a programming language construct is specified to have two or more possibilities of behavior. In such a case, different compilers may generate different behaviors from the same source code, resulting in a vulnerability. The problem is exacerbated if the compiler(s) are run on different computers; if the compilers use different software libraries; or if they run on different operating systems, different releases of an operating system, or different configurations of an operating system.

Another issue is implementation-defined behavior. Programming languages sometimes allow compilers to support a variety of behaviors for a single language feature, or combination of features, that may enable usage on a wider range of hardware or enable use of the language in a wider variety of circumstances. However, there is a requirement that each implementation document the behavior. Vulnerabilities can occur when the programmer does not take into account this documented behavior or ports code from one machine to another without considering changes in implementation-defined behavior.

Undefined behavior is also a threat. Programming languages sometimes specify that program behavior is undefined or simply leave some behavior undefined. Common examples include recovery from an error in the software, and use of the value of a variable that has not yet been assigned. In some cases, attackers can use expert knowledge to stimulate behavior that can lead to a vulnerability.

Human Cognitive Limitations

Programming languages are created with different purposes, some are for general use and others for specific tasks or needs, but all are created as tools to be used by software programmers to manipulate data and produce a desired result. This means the intended audiences for the languages are different. For instance, C was created for programmers implementing system software, while COBOL was created for programmers writing business applications.

Because everyone is different and each person has their own levels of understanding and areas of expertise, vulnerabilities can occur because of the abilities of the person writing the code as well as by those who maintain it. Programmers may choose syntaxes that make the most sense to them, even though the language provides another syntax that would accomplish the same task, or may have performed the function more efficiently. Also, as people, programmers have to deal with the stresses of their personal and professional lives, any of which may have an impact on the quality of code that person writes, which could in turn result in a potential vulnerability.

This can be addressed by standardizing and simplifying as much as possible, and by improving documentation and resources, including project coding standards and review processes, that directly deal with these issues.

Lack of Portability and Interoperability

In addition to potential issues resulting from how code is written and from variations in the compilers or configurations of the same compiler, other factors can result in potential vulnerabilities when the application is run, such as if the application is used with different software libraries, on different operating systems, or on different hardware.

Developers must be aware of these possibilities and plan for them, for instance, by using only semantics specifically defined by the language, and by using software libraries specifically created for the language whenever possible.

Inadequate Intrinsic Support in the Language

Although using specified software libraries for an application can reduce risk, sometimes no libraries are specified by the programming language or the libraries used are not validated to the same standard as the compiler and the applications being developed, are proprietary and inclined to change in later releases, or are discontinued and no longer supported by the vendor. Such instances can lead to potential vulnerabilities.

A programmer can reduce this risk by using stronger types or controls to perform certain operations, though this may reduce the performance and flexibility of the application. Therefore, the developer must strike a balance between the intrinsic support provided by the language to help avoid vulnerabilities and the ultimate utility of the application.

Language Features Prone to Erroneous Use

In some programming languages the syntactic constructs used by the language are simple and straightforward to use, while others in that same language are extremely complex. Vulnerabilities can result when language constructs are used improperly, when complex constructs are misused in acceptable but unintended ways, or when complex constructs that can be substituted for by a series of simpler constructs are used without an understanding of the full effects of the constructs.

Such vulnerabilities can be reduced by those creating the language by identifying such constructs, and providing standardized ways for dealing with them.

The common strand throughout all of the causes listed above is lack of knowledge. With perfect knowledge, the execution of code can be predicted, but this is seldom the case. Expert attackers can exploit superior knowledge to "trick" the code into executing function that the code's developer did not intend or foresee.

Example Vulnerabilities

An example of a vulnerability described in the Technical Report would be the following:

When subexpressions with side effects are used within an expression, the unspecified order of evaluation can result in a program producing different results on different platforms, or even at different times on the same platform. For example, consider

```
a = f(b) + g(b);
```

where f and g both modify b. If f(b) is evaluated first, then the b used as a parameter to g(b) may be a different value than if g(b) is performed first. Likewise, if g(b) is performed first, f(b) may be called with a different value of b.

Other examples of unspecified order, or even undefined behavior, can be manifested, such as

```
a = f(i) + i++;
```

or

```
a[i++] = b[i++];
```

Parentheses around expressions can assist in removing ambiguity about grouping, but the issues regarding side effects and order of evaluation are not changed by the presence of parentheses; consider

```
j = i++ * i++;
```

where even if parentheses are placed around the i++ subexpressions, undefined behavior still remains. (This example uses the syntax of C; the effects can be created in any language that allows functions with side effects in the places where the example shows the increment operations.)

In this case, the report suggests that programmers should decompose the expression into sequential statements so that the order of evaluation can be controlled.

The unpredictable nature of the calculation means that the program cannot be tested adequately to any degree of confidence. A knowledgeable attacker can take advantage of this characteristic to manipulate data values triggering execution that was not anticipated by the developer.

An example of an application vulnerability included in the report would be storing a password in vulnerable cleartext because the programming language did not provide a library function for encrypting the password. For this problem, the project should acquire a subroutine library that provides the functionality missing from the language library.

A Catalog of Language Vulnerability Types

Vulnerabilities included in the report were identified and selected using two different analyses. A bottom-up analysis surveyed application security vulnerabilities observed "in the wild" and identified language characteristics that can serve as root causes of the application vulnerabilities. A top-down analysis surveyed existing language style and usage guides for the production of safety-related software.

All language vulnerabilities in the ISO/IEC report are described in a language-independent manner allowing readers to quickly comprehend and utilize the information.

Programming Language Vulnerability Description

Each type of programming language vulnerability is described in a uniform format to permit easy reference. Information in the description includes:

- An arbitrary three-letter identifier that can be used to identify the vulnerability.
- A brief summary of the programming language vulnerability.
- Cross-references, such as CWE identifier.
- A description of the mechanism of failure, giving the link between the programming language vulnerability and resulting application vulnerabilities.
- A list summarizing the characteristics of languages for which this vulnerability is applicable.
- A brief description of how application developers can avoid the vulnerability or mitigate its negative effects.
- Comments regarding how the maintainers of the language's specification might make improvements.

Application Vulnerability Description

The report also lists a handful of application vulnerabilities that might be mitigated if better support were provided in programming language libraries. These are described similarly to the language vulnerabilities, except that the comments to language maintainers are omitted.

An Ongoing Process

The list of vulnerabilities detailed in the ISO/IEC report is not complete. With new vulnerabilities being discovered regularly, the process will always be ongoing. The report therefore only describes those programming language vulnerability types that were determined to have sufficient probability and significance to date.

In addition, the following five subject areas were not addressed in this initial release but will be addressed in future editions of the report:

- Object-oriented language features, though certain simple issues related to inheritance are discussed.
- Concurrency.
- Numerical analysis, though certain simple items regarding the use of floating point are discussed.
- Scripting languages.
- Inter-language operability.

The second edition of the Technical Report will also add annexes describing how the vulnerabilities appear in particular programming languages. Currently, annexes are planned for Ada, C, Python, Ruby and SPARK. Future editions will add more language-specific annexes as well as describing additional vulnerabilities.

The report is available for purchase from <http://www.iso.org> and <http://www.ansi.org>. Individual users can obtain the report for free at <http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html>.

About ISO/IEC Standards and Technical Reports

There are three major international standards associations that bring together national bodies from participating nations, as well other international governmental and nongovernmental organizations, to focus on the development of international standards for business, government, and society—the ISO, IEC, and International Telecommunication Union [10].

While the primary work of ISO and IEC is to prepare international standards, some subjects are not appropriate for standardization but are suitable for technical reports that provide guidance and information that have been formed via consensus.

The ISO/IEC report about programming language vulnerability types discussed in this article, Technical Report 24772:2010, was published by a subcommittee working group of the ISO/IEC Joint Technical Committee for the field of information technology that is responsible for, "programming languages, their environments, and system software interfaces."

REFERENCES

1. <http://www.cve.mitre.org/>
2. <http://www.owasp.org/index.php/OWASP_Top_Ten_Project>
3. <http://www.cwe.mitre.org/>
4. <http://cwe.mitre.org/top25/index.html>
5. <http://www.cert.org/>
6. <http://www.misra-c.com/>
7. <http://www.iso.org/>
8. <http://www.iec.ch/>
9. <http://www.iso.org/iso/catalogue_detail.htm?csnumber=41542>
10. <http://www.itu.int/>

ABOUT THE AUTHORS

James W. Moore is a 40-year veteran of software engineering in IBM and, now, the MITRE Corporation. He is a leader in software and systems engineering standardization for the IEEE, serving as its liaison to ISO/IEC JTC1/SC7 and as a member of the Executive Committee of the IEEE Software and Systems Engineering Standards Committee. He serves as a member of the IEEE Computer Society's Board of Governors. He was an Executive Editor of the Society's 2004 "Guide to the Software Engineering Body of Knowledge" and a member of the Editorial Board of the revision of the "Encyclopedia of Software Engineering." The IEEE Computer Society has recognized him as a Charter Member of their Golden Core, and the IEEE has named him a Fellow of the IEEE. His work on software engineering standards has been recognized by the International Committee on Information Technology Standards (INCITS) with their International Award, by the Computer Society with the Hans Karlsson Award, and by the IEEE with the Charles Proteus Steinmetz Award. His latest book on software engineering standards was published in 2006 by John Wiley & Son. He holds two US patents and, dating to times when software was not regarded as patentable, two "defensive publications". He graduated from the University of North Carolina with a B.S. in Mathematics, and Syracuse University with an M.S. in Systems and Information Science.

E-mail: James.W.Moore@ieee.org
Phone: 301-938-0260

John Benito is an independent consultant providing software development, project management, and software testing. He is the current Convener of ISO/IEC JTC 1/SC 22/WG14 the ISO group responsible for Standard C, the Convener of ISO/IEC JTC 1/SC 22 WG 23 (was OWG Vulnerabilities), the project editor for the Technical Report 24772, and a member of the INCITS PL22.11 (ANSI C) technical committee. He previously was a member of INCITS PL22.16 (ANSI C++) and the ISO Java Study group. He has been in software development, project management, and testing for over 35 years. Mr. Benito has been participating in International Standard development for the past 22 years, and is the recipient of the INCITS Exceptional International Leadership Award.

E-mail: benito@bluepilot.com
Phone: 831-427-0528

Dr. Larry Wagoner has served in a variety of technical and/or analytic organizations within the National Security Agency for over 25 years. Before coming to the Information Assurance Directorate, he worked primarily in the Signals Intelligence Directorate and the Research Directorate. He has a Ph.D. in computer science from the University of Maryland, Baltimore County.

E-mail: l.wagone@radium.ncsc.mil