

A method to assess 'forgivable' vs 'unforgivable' vulnerabilities

Research from the NCSC designed to eradicate vulnerability classes and make the top-level mitigations easier to implement.

Executive Summary

All systems contain vulnerabilities. In fact, the number of Common Vulnerabilities and Exposures (CVEs) in commodity technology continues to rise. While there are a number of factors that are driving the increasing numbers, the NCSC expect this trend to continue unless interventions are made.

We know many vulnerabilities are complex and hard to avoid. But vulnerabilities that are trivial to find (and that occur time and time again) are ones the NCSC are aiming to drive down at scale. These 'unforgivable vulnerabilities', a phrase coined by [Steve Christie in his 2007 MITRE paper](#), *'are beacons of a systematic disregard for secure development practices. They simply should not appear in software that has been designed, developed, and tested with security in mind.'*

This paper extends the ideas in the MITRE paper and proposes a method to assess a vulnerability as 'forgivable' or 'unforgivable'. More importantly, this paper intends to generate discussion with vendors, and is a call on them to work to eradicate vulnerability classes and make the top-level mitigations discussed in this paper easier to implement.

The NCSC's analysis seeks to identify the **root cause** of vulnerabilities (opposed to the details provided in the individual vulnerability advisory), using the CWE Top 25 Most Dangerous Software Releases for 2023. Having identified 11 top-level mitigations required to manage these vulnerabilities, we assigned an 'ease of implementation' score to each top-level mitigation, based on:

- direct and indirect costs
- knowledge (that is, how widely known and understood is the mitigation)
- technical feasibility

Researchers can then assess an individual vulnerability, and use the ‘ease of implementation’ scores (which we classify as ‘easy’, ‘medium’ or ‘hard’ to implement) to assess how difficult it is to apply the mitigations. Vulnerabilities with ‘easy’ mitigations are declared ‘unforgivable’.

Most of the 13 ‘unforgivable vulnerabilities’ mentioned in the original MITRE 2007 paper still exist in one form or another. At the core of our research is the desire to eradicate vulnerability classes and make the top-level mitigations easier to implement. The NCSC believe this can be best done by making operating systems more secure, by improving development frameworks, and by encouraging developers and vendors to adopt secure programming concepts.

Note:

This guide is written for software development and security professionals, and assumes a familiarity with modern development techniques. The following abbreviations are used throughout.

CISA Cybersecurity and Infrastructure Security Agency (US government agency)

CVE Common Vulnerabilities and Exposures

CVSS Common Vulnerability Scoring System

CWE Common Weakness Enumeration

KEV Known Exploited Vulnerabilities (CISA KEV List/Catalog)

KLOC Thousand Lines of Code

NCSC National Cyber Security Centre


NVD National Vulnerabilities Database

OWASP Open Web Application Security Project

SQL Structured Query Language

Scope

The aim of this technical paper is to define a method that allows security professionals to determine if a given software vulnerability is ‘forgivable’ or ‘unforgivable’ as defined below.

 Show all

Forgivable vulnerabilities

Show

Unforgivable vulnerabilities

Show

Non-exploitable vulnerabilities

Hide

There is a third class of vulnerability which we have called **Non-exploitable**. We don’t discuss this class further in this paper, but arguably this is a vulnerability that is not a security risk. This could be because:

- there is no code path to be able to exploit it, or
- there are mitigations in place to prevent the vulnerability being exploited, or
- it is unlikely that chaining vulnerabilities will allow exploitation

Background

Analysis by Eloff and Bella (2018), Williams et al. (2018) and Zheng et al. (2019) suggests that no single control has had a significant effect on reducing the number of software bugs. The average number of bugs in software has remained constant over the last 20 years (McConnell, 2004; Coverity, 2014; Coverity, 2019). Table 1 provides an overview of the statistics. Coverity (2019)

identified the average defect density for the software industry to be 1 defect per thousand lines of code (KLOC). This highlights that whatever the software product and whatever the programming language used, software defects will always be present.

Table 1: Average number of software defects over time

Source	Defects per KLOC
Jones (1986)	0.5
Microsoft Apps production (McConnel, 2004)	0.5
Coverity (2014)	0.76
Coverity (2019)	1

Hatton et al. (2017) found that software source code in systems doubles on average every 3.5 years. This is largely due to:

- user demand for additional functionality and sophistication
- the increasing hardware processing power required to accommodate the increase in demands

If the source code base doubles in size, the number of defects will increase accordingly. Gaikovina Kula et al. (2010) argue that the increasing complexity of source code has been observed to correlate with increasing the time to remediate a software defect. Therefore, source code base size and the average number of defects per KLOC should be taken into consideration when assessing ease of implementation of mitigations.

Figure 1 illustrates the ‘System Programming Stack’, a generic representation of how a modern operating system is designed. The example vulnerabilities discussed in this paper affect different operating systems / platforms at different levels of this generic stack.

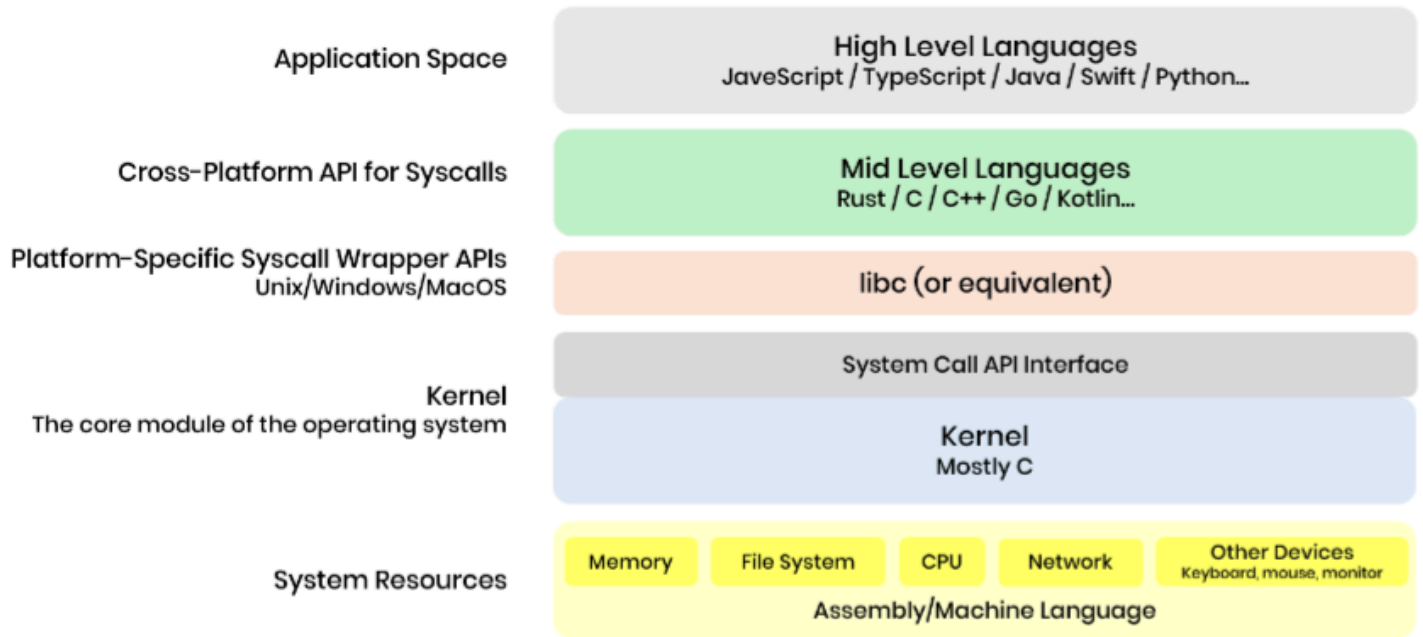


Figure 1: System Programming Stack (image: iglu.net)

Research methodology

The research methodology used is as follows:

1. Analyse the 'CWE Top 25 Most Dangerous Software Weaknesses 2023' (shown in table 2 below).
2. Identify the top-level mitigations required to manage the vulnerabilities.
3. Assign each mitigation with a score that ranks its 'ease of implementation' (that is, how easy it is to implement the mitigation, based on **cost**, **technical feasibility** and **knowledge**).
4. To assess an individual vulnerability, identify the top-level mitigations required to manage that vulnerability.
5. Use the total implementation scores to *quantify* how easily the mitigations could be applied.
6. Vulnerabilities with 'easy' mitigations are declared 'unforgivable'.

Note: The ease of implementation is predicated on the mitigation being implemented at the beginning of the product’s development and not retrofitted. The cost and technical feasibility scores will increase dramatically if the mitigation is retrofitted.

The source data for this paper comes from both the Common Weakness Enumeration (CWE) Top 25 Most Dangerous Software Weaknesses of 2023 ([MITRE, 2023](#)), shown in table 2 below. This list was calculated by MITRE analysing public vulnerability data in the U.S. National Vulnerability Database (NVD) for their root causes via CWE mappings. Note that:

- the list is based on 43,996 CVE records for vulnerabilities in 2021 and 2022
- the mapping data was pulled from the NVD on March 27, 2023
- the top-level mitigations are drawn from the detailed Common Weakness Enumeration, and summarised for the purposes of this research paper
- further analysis can be undertaken against the ‘2023 CWE Top 10 KEV Weaknesses’ (which consists of CVE records that appear in the CISA Known Exploited Vulnerabilities [KEV] Catalog, CISA 2023)

Table 2: CWE Top 25 Most Dangerous Software Weaknesses 2023

Rank	ID	Name	CVEs in KEV	Top-Level Mitigation
1	CWE-787	Out-of-bounds Write	70	<ul style="list-style-type: none"> • Language Selection • Libraries or Frameworks • Input Validation
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	4	<ul style="list-style-type: none"> • Libraries or Frameworks
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	6	<ul style="list-style-type: none"> • Libraries or Frameworks • Output Encoding • Input Validation
4	CWE-416	Use After Free	44	<ul style="list-style-type: none"> • Language Selection
5	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	23	<ul style="list-style-type: none"> • Sandbox or Jail • Reduce the Attack Surface • Libraries or Frameworks • Output Encoding • Input Validation • Enforcement by Conversion • Compilation or Build Hardening

				<ul style="list-style-type: none"> Secure Architecture and Design
6	CWE-20	Improper Input Validation	35	<ul style="list-style-type: none"> Reduce the Attack Surface Libraries or Frameworks Input Validation
7	CWE-125	Out-of-bounds Read	2	<ul style="list-style-type: none"> Input Validation Language Selection
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	16	<ul style="list-style-type: none"> Input Validation Libraries or Frameworks Sandbox or Jail
9	CWE-352	Cross-Site Request Forgery (CSRF)	0	<ul style="list-style-type: none"> Libraries or Frameworks
10	CWE-434	Unrestricted Upload of File with Dangerous Type	5	<ul style="list-style-type: none"> Input Validation Enforcement by Conversion
11	CWE-862	Missing Authorization	0	<ul style="list-style-type: none"> Reduce the Attack Surface Libraries or Frameworks

12	CWE-476	NULL Pointer Dereference	0	<ul style="list-style-type: none"> Input Validation
13	CWE-287	Improper Authentication	10	<ul style="list-style-type: none"> Libraries or Frameworks
14	CWE-190	Integer Overflow or Wraparound	4	<ul style="list-style-type: none"> Language Selection Libraries or Frameworks Input Validation
15	CWE-502	Deserialization of Untrusted Data	14	<ul style="list-style-type: none"> Input Validation
16	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	4	<ul style="list-style-type: none"> Libraries or Frameworks Input Validation
17	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	7	<ul style="list-style-type: none"> Libraries or Frameworks Input Validation
18	CWE-798	Use of Hard-coded Credentials	2	<ul style="list-style-type: none"> Secure Architecture and Design
19	CWE-918	Server-Side Request Forgery (SSRF)	16	<ul style="list-style-type: none"> Language Selection Input Validation

20	CWE-306	Missing Authentication for Critical Function	8	<ul style="list-style-type: none"> Secure Architecture and Design Libraries or Frameworks
21	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	8	<ul style="list-style-type: none"> Language Selection
22	CWE-269	Improper Privilege Management	5	<ul style="list-style-type: none"> Separation of Privilege
23	CWE-94	Improper Control of Generation of Code ('Code Injection')	6	<ul style="list-style-type: none"> Secure Architecture and Design Input Validation
24	CWE-863	Incorrect Authorization	0	<ul style="list-style-type: none"> Secure Architecture and Design Libraries or Frameworks
25	CWE-276	Incorrect Default Permissions	0	<ul style="list-style-type: none"> Separation of Privilege

Assessing 'ease of implementation'

To assess the ‘ease of implementation’ of the top-level mitigations, several methods were considered:

1. **Cost-effectiveness analysis:** a form of economic analysis that compares the relative costs and outcomes (effects) of different courses of action. While typically used in the healthcare industry, it is increasingly used in other domains (ICEAA, 2024).
2. **Cost-benefit analysis:** the systematic and analytical process of comparing benefits and costs in evaluating the desirability of a project or programme. It attempts to answer such questions as whether a proposed project is worthwhile, the optimal scale of a proposed project and the relevant constraints (Mishan and Quah, 2020).
3. **Ease/impact matrix:** the effort-to-impact matrix is a simple but valuable tool that can help instructional coaches, teacher teams and administrators to prioritise their efforts and make strategic choices about which steps to take and when. It allows the user to map out potential strategies and identify how much effort each will take versus how much impact it is likely to make (Helmke, 2022).

Based on the understanding of the different assessment methods, a cost-effectiveness analysis approach was chosen with the following factors taken into consideration:

1. **Cost:** including direct costs (such as licensing a library) and indirect costs (such as additional time needed to code/recode).
2. **Knowledge:** how widely-known and understood is the mitigation.
3. **Technical Feasibility:** a feasibility analysis evaluates the mitigation’s potential for success, and how easy the mitigation is to achieve. This includes evaluating the advantages and disadvantages of the mitigation, and any technical requirements or prerequisites for the mitigation (such as hardware support).

For each top-level mitigation, the **cost**, **knowledge** and **technical feasibility** is assigned a score out of 3, based on the NCSC’s analysis of the literature review of academic papers and industry white papers:

- 1 indicates ‘easy/inexpensive’
- 3 indicates ‘hard/expensive’
- 2 indicates the intermediate

When the scores are added together, they provide an overall ‘Implication Score’ for each mitigation, which can be assigned a level of difficulty.

Table 3: Implementation Score

Implementation Score	Difficulty
3-4	Easy
5-6	Medium
7-9	Hard

For example, for the top-level mitigation **Input Validation**:

- cost = **1**
- knowledge = **1**
- technical feasibility = **1**

This gives a total of **3**, which means the difficulty for this mitigation can be declared **Easy**.

The following table lists the implementation score for each top-level mitigation, and the corresponding difficulty level.

Table 4: Implementation Score for top-level mitigations

Top-level mitigation	Implementation Score	Difficulty
Input Validation	3	Easy
Output Encoding	3	Easy
Reduce the Attack Surface	5	Medium
Enforcement by Conversion	5	Medium
Sandbox or Jail	5	Medium
Secure Programming	6	Medium
Compilation or Build Hardening	6	Medium
Separation of Privilege	6	Medium
Libraries or Frameworks	6	Medium
Secure Architecture and Design	7	Hard
Language Selection	8	Hard

Analysis of top-level mitigations

This section explains how we arrived at the Implementation Score for each of the top-level mitigations. They are presented in order of difficulty, starting with the easiest mitigations first.

Input Validation

The OWASP Cheat Sheet (OWASP, 2021) provides the following examples for input validation strategies:

- data type validators available natively in web application frameworks (such as [Django Validators](#) and [Apache Commons Validators](#))
- validation against [JSON Schema](#) and [XML Schema \(XSD\)](#) for input in these formats
- type conversion (e.g. `Integer.parseInt()` in Java, `int()` in Python) with strict exception handling

Input Validation is a subset of the Libraries or Frameworks mitigation. However, its prevalence in the mitigation list means it's worth listing separately.

	Score	Evidence
Cost	1	Low/no direct and indirect costs
Knowledge	1	Widely understood and widely available
Feasibility	1	Many implementations for most/all languages
Total	3 (Easy)	

Output Encoding

Encoding and escaping are defensive techniques meant to stop injection attacks. Encoding (commonly called 'output encoding') involves translating special characters into some different but equivalent form that is no longer dangerous in the target interpreter, for example translating the `<` character into the `<` string when writing to an HTML page.

Escaping involves adding a special character before the character/string to avoid it being misinterpreted, for example, adding a `\` character before a `"` (double quote) character so that it is interpreted as text and not as closing a string.

Output Encoding is a subset of the Libraries or Frameworks mitigation. However, its prevalence in the mitigation list means it's worth listing separately.

	Score	Evidence
Cost	1	Low/no direct and indirect costs
Knowledge	1	Widely understood and widely available
Feasibility	1	Many implementations for most/all languages
Total	3 (Easy)	

Reduce the Attack Surface

Originating from the security sector, an ‘attack surface’ measure typically reflects the number of input points and output points that can be exploited by a potential attacker. A larger attack surface provides more places to attack, and more opportunities for developers to introduce weaknesses. In some cases, this measure may reflect other aspects of quality besides security. For example, a product with many inputs and outputs may require a large number of tests in order to improve code coverage. This can also include the use of functions, libraries, and frameworks.

The attack surface of an application is the union of code, interfaces, services, protocols, and practices available to all users, with a strong focus on what is accessible to unauthenticated users (Microsoft, 2019). Therefore, developers should look to reduce the exposure of the application at the earliest stages of development, specifically at the design stage. Further considerations should be given at the coding stage and finally at runtime.

	Score	Evidence
Cost	2	Removing code/functionality will take time and thus cost
Knowledge	1	Concept is widely known and understood
Feasibility	2	Technically feasible but challenges in justifying removal of functionality/features
Total	5 (Medium)	

Enforcement by Conversion

This mitigation involves converting the input into a different, well-controlled representation. For example, in PHP, a common mechanism for avoiding SQL injection is to apply `intval()` to all numeric inputs, which guarantees that the generated value is a number.

	Score	Evidence
Cost	1	Usually available as part of the language
Knowledge	3	Not widely used or understood
Feasibility	1	Does not rely upon any prerequisite
Total	5 (Medium)	

Sandbox or Jail

Run the code in a ‘jail’ or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict which files can be accessed in a particular directory, or which commands can be executed by the software.

OS-level examples include the Unix chroot jail, AppArmor, and SELinux. In general, managed code may provide some protection. For example, `java.io.FilePermission`

in the Java SecurityManager allows the software to specify restrictions on file operations.

This may not be a feasible solution, and it only limits the impact to the operating system; the rest of the application may still be subject to compromise.

	Score	Evidence
Cost	1	Usually part of the language/OS
Knowledge	2	Concept is known and partially understood
Feasibility	2	Technically feasible but with limitations
Total	5 (Medium)	

Secure Programming

This is a large subject and typically is down to the developers. Examples of secure programming include:

- when freeing pointers, be sure to set them to NULL once they are freed
- if all pointers that could have been modified are sanity-checked previous to use, nearly all NULL pointer dereferences can be prevented
- ensure that all protocols are strictly defined such that all out-of-bounds behaviour can be identified simply, and require strict conformance to the protocol
- when deserializing data, populate a new object rather than just deserializing
- explicitly define a final object() to prevent deserialization
- use library calls rather than external processes to recreate the desired functionality

	Score	Evidence
Cost	2	Requires skilled developers
Knowledge	2	Concept is known and partially understood
Feasibility	2	Technically feasible with few technical prerequisites
Total	6 (Medium)	

Compilation or Build Hardening

There are 4 areas to be examined when hardening the toolchain: configuration, preprocessor, compiler, and linker. Developers and their development environments are part of the software supply chain, so if their accounts get compromised, attackers get control over parts of this chain. Nowadays, many developers are working on these environments from their homes and can end up as entry points for malicious code or let attackers steal credentials to production services. Therefore, while ‘hardening’ used to mean securing a developer’s local computer, it now also means bolstering the security of the tools they need to do their work. These include source code management (SCM) tools, binary artifacts, and build/CI/CD pipelines.

Build hardening includes using automatic buffer overflow detection mechanisms. These are offered by certain compilers or compiler extensions, e.g. Microsoft Visual Studio /GS flag, Fedora/Red Hat FORTIFY_SOURCE GCC flag, StackGuard, and ProPolice, which provide various mechanisms including canary-based detection and range/index checking. D3-SFCV (Stack Frame Canary Validation) from D3FEND (MITRE, 2023) discusses canary-based detection in detail. Other examples include the Control-flow Enforcement Technology (CET) Shadow Stack computer processor feature. It provides capabilities to defend against return-oriented programming (ROP) based attacks.

	Score	Evidence
Cost	1	Usually available as part of the tool chain
Knowledge	2	Not widely understood
Feasibility	3	Impact on performance and usually specific to compiler or architecture
Total	6 (Medium)	

Separation of Privilege

Separation of privilege, also called privilege separation, refers to both the 'segmentation of user privileges across various, separate users and accounts' (Microsoft, 2021), and the compartmentalisation of privileges across various application or system sub-components, tasks, and processes.

A different, more generic description is that multiple conditions need to be met in order to gain access to a given process or object. A control could be a permission, for example.

	Score	Evidence
Cost	2	Will take time and thus cost
Knowledge	2	Concept is widely known and understood
Feasibility	2	Technically feasible (see XP to Vista implementation) but requires prerequisites
Total	6 (Medium)	

Libraries or Frameworks

The difference between a library and a framework can be summarised as follows:

- for a library, your application code calls the library code
- for a framework, your application code is called by the framework

The general advice is to use the latest version of a vetted library or framework to fulfil the functionality that is required. This is largely to avoid duplication, development time and in some cases to avoid creating your own implementation of a complex function such as crypt. The main security assumption is that the library or framework would not allow a weakness to occur, or provides constructs that make weakness easier to avoid.

As with the choice of programming language, the choice of library or framework also has several considerations, including:

- size and complexity – ideally you would want to use as lightweight library / framework as possible (to avoid bloatware etc).
- performance requirements
- use automated bundle tracking, e.g. to manage the latest version or to check for large updates
- impact to web accessibility
- backward compatibility
- licensing
- documentation
- correctly implemented
- updates
- security
- support / community / vendor support – some libraries are backed by vendors or communities who invest time and money into keeping libraries up to date and secure

However, there are significant implications of migrating between frameworks. It is typically non-trivial, time-consuming, can often be fraught with complex technical challenges and can result in a lot of code to be re-written (LinkedIn, 2023; Opsview, 2023).

	Score	Evidence
Cost	2	Licensing costs
Knowledge	2	Current level of knowledge as highlighted by JetBrains (2022)
Feasibility	2	Performance and security impact
Total	6 (Medium)	

Secure Architecture and Design

Ensuring the product is securely architected and designed should be done at the beginning of the product development. Redesign of the product's security architecture and/or design, or to retrospectively add core security mitigations, is expensive and difficult.

Examples include:

- divide the product into anonymous, normal, privileged, and administrative areas
- refactor the product so that you do not have to dynamically generate code

	Score	Evidence
Cost	2	A critical, core process of software development
Knowledge	3	Not widely used or understood
Feasibility	2	Technically feasible with few technical prerequisites
Total	7 (Hard)	

Language Selection

The choice of programming language will have a profound impact on the security of the application. While modern languages, such as Rust and Swift, are both type-safe and memory-safe, languages such as C, C++, and assembly do not offer full type and memory safety for developers.

In theory an engineer has a choice of programming languages to develop in. However, there are a number of constraints:

1. The ecosystem, what support is there, will it be around for a long time?
2. What is the environment / platform for the project (web, mobile, embedded device, OS application, etc.)?
3. What are the specific requirements for libraries, features, and tools for the programming language?
4. What is the performance consideration and is the languages suitable to accommodate this performance?
5. Is the developer able to code in this language?

At the beginning of a new project, developers should choose a type-safe and memory-safe programming language. The State of Developer Ecosystem 2022 survey (JetBrains, 2022) of 29,269 developers from around the world highlighted that one out of every two developers is planning to adopt a new language. The top choices for next languages are Go, Rust, Kotlin, TypeScript, and Python. This highlights that the current level of knowledge is assessed at level 2.

However, there are a vast majority of even modern applications and operating systems written in C and C++. As an example, Chrome announced that they will soon support third-party Rust libraries (Google Security Blog, 2023). But Chrome has millions of lines of C++ and it will take a long time to migrate. This would mean the technical feasibility of retrospectively applying this mitigation is assessed at level 3.

	Score	Evidence
Cost	3	Listed constraints would increase costs
Knowledge	2	Current level of knowledge as highlighted by JetBrains, 2022
Feasibility	3	Could be limited by requirements or time taken if retrospectively applying this mitigation (Google Security Blog, 2023)
Total	8 (Hard)	

Worked example: applying methodology to a recent vulnerability

Below is an example of how the analysis can be used to determine if a vulnerability is ‘unforgivable’.

CVE-202X-XXXXX Vendor Application Unauthenticated SQL Injection Vulnerability

Vendor Advisory: <http://www.example.com/security>

CVSS v3.x Base: 9.8 Critical

Analysis: <https://www.ncsc.gov.uk/sql-deep-dive-and-indicators-of-compromise/>

Analysis of the vulnerability by examining changes to the code in the updated version identified three issues:

Analysis	Relevant Mitigation	Ease of implementation
Building the SQL query by concatenating arguments passed from user input has been replaced with a safer SQL library.	Libraries or Frameworks	Medium
<code>GetQueryVarsFromUser()</code> function removed.	Reduce the Attack Surface	Medium
Variables are set to NULL before they are used.	Input Validation	Easy
<p>While the exploitable vulnerability was difficult to find, one of the root causes was deemed easy to implement, and two were rated medium.</p> <p>This vulnerability should not have existed and is unforgivable.</p>		

Conclusions

At the core of this paper is the need to eradicate vulnerability classes and make the top-level mitigations easier to implement. This can be focussed on the following three areas:

- 1 **Operating Systems**
 - Banning/remove unsafe functions. For example, there was a 41% of vulnerability reduction with the move from XP to Vista from banning strcpy and associated functions (Howard, 2007).
 - Handle security and reliability in common frameworks, APIs, and libraries. Only exposing an interface that makes writing code with common classes of security

vulnerabilities impossible.

- Make high impact mitigations much easier to implement via APIs (sandboxing, privilege separation), or enforce the use of these.

2 **Development Environments**

The development environments, and specifically the Integrated Development Environment (IDE), need to make the following easier for developers:

- programming language documentation
- understanding (and fixing) compiler warnings
- using secure programming languages in default new projects
- migrating code from existing projects to secure programming languages
- finding and integrating trusted third party frameworks and libraries
- using built-in tools to highlight vulnerabilities in source code during development
- eradicating vulnerability classes at the beginning of development

3 **Developers (vendors)**

Developers and vendors need to adopt secure programming concepts, and enforce their use to make it harder for the mistakes to be made at source. Vulnerabilities must be caught early in the development process. This includes the understanding of the following topics:

- methodology and process to catch vulnerabilities during development
- basics of C programming and emphasising programming languages
- memory layout/architecture and emphasising/adopting the use of technologies such as Rust and CHERI

Further research

The method of assessing forgivable and unforgiveable vulnerabilities discussed in this paper could be matured to take account of other factors, including:

- the role of the product and the severity of the vulnerability (for example, if the product is an internet-facing service and the vulnerability can be exploited pre-authentication)
- how to determine the vulnerability management maturity of the vendor/developer (for example, is the vendor monitoring vulnerabilities throughout the expected lifecycle of the product)

- is being unaware of a vulnerability (that is later exploited in the wild) ‘unforgivable’, and should other factors also be taken into consideration (including know bug/vulnerability classes and their typical mitigations)

References

Apple (2022) *Blog - Towards the next generation of XNU memory safety: kalloc_type - Apple Security Research*. Available at: [Blog - Towards the next generation of XNU memory safety: kalloc_type - Apple Security Research](#) (Accessed: 20 November 2023).

Baires Dev (2022) *How to Choose the Right Programming Language for a New Project*, BairesDev Available at: [How to Choose the Right Programming Language for a New Project | BairesDev](#) (Accessed: 7 March 2024).

Christey, S. (2007) ‘*Unforgivable Vulnerabilities*’

CISA (2023a) *2022 Top Routinely Exploited Vulnerabilities | CISA*. Available at: [2022 Top Routinely Exploited Vulnerabilities | CISA](#) (Accessed: 28 September 2023).

CISA (2023b) *Known Exploited Vulnerabilities Catalog | CISA*. Available at: [Known Exploited Vulnerabilities Catalog | CISA](#) (Accessed: 23 January 2024).

Coverity (2014) *2014 Coverity Scan Report*. Available at: <https://news.blackduck.com/2015-07-29-Coverity-Scan-Open-Source-Report-Shows-Commercial-Code-Is-More-Compliant-to-Security-Standards-than-Open-Source-Code> (Accessed: 17 June 2019).

Coverity (2019) *Coverity*. Available at: [Coverity Scan - Static Analysis](#) (Accessed: 17 June 2019).

Eloff, J. and Bella, M.B. (2018) ‘Software Failures: An Overview’, in *Software Failure Investigation*. Cham: Springer International Publishing, pp. 7–24. Available at: 10.1007/978-3-319-61334-5_2 (Accessed: 13 June 2019).

Fulton, K.R., Chan, A., Votipka, D., Hicks, M. and Mazurek, M.L. (2021) '*Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study*', , pp. 597–616. Available at: [Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study | USENIX](#) (Accessed: 28 September 2023).

Gaikovina Kula, R., Fushida, K., Kawaguchi, S. and Iida, H. (2010) 'Analysis of Bug Fixing Processes Using Program Slicing Metrics', in Ali Babar, M., Vierimaa, M. and Oivo, M. (eds.) *Product-Focused Software Process Improvement*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 32–46. Available at: 10.1007/978-3-642-13792-1_5 (Accessed: 10 July 2019).

Google Security Blog (2023) Supporting the Use of Rust in the Chromium Project. *Google Online Security Blog*. Available at: [Supporting the Use of Rust in the Chromium Project](#) (Accessed: 28 September 2023).

Hanley, Z. (2023) *MOVEit Transfer CVE-2023-34362 Deep Dive and Indicators of Compromise*, [Autonomous Pentesting Platform](#) Available at: [MOVEit Transfer CVE-2023-34362 Deep Dive and Indicators of Compromise](#) (Accessed: 24 January 2024).

Helmke, S (2022) *Where do you start when everything feels urgent? Use an effort-to-impact matrix*. Available at: [WHERE DO YOU START WHEN EVERYTHING FEELS URGENT? - ProQuest](#) (Accessed: 22 March 2024).

Howard, M (2007) *Howard.pdf*. Available at: <https://www.acsac.org/2007/workshop/Howard.pdf> (Accessed: 24 January 2024).

ICEAA (2024) *International Cost Estimating and Analysis Association*. Available at: [International Cost Estimating and Analysis Association](#) (Accessed: 22 March 2024).

Jaiswal, H (2023) *CVE-2023-36934 Analysis: MOVEit Transfer SQL Injection*, [ProjectDiscovery - Vulnerability management / Blog](#) Available at: [CVE-2023-36934 Analysis: MOVEit Transfer SQL Injection – ProjectDiscovery Blog](#) (Accessed: 28 September 2023).

JetBrains (2022) *The State of Developer Ecosystem in 2022 Infographic*, *JetBrains: Developer Tools for Professionals and Teams* Available at: [The State of Developer](#)

[Ecosystem in 2022 Infographic](#) (Accessed: 28 September 2023).

Kohlhase, M. (2023) *Rust OS comparison*. Available at: [GitHub – flosse/rust-os-comparison: A comparison of operating systems written in Rust](#) (Accessed: 20 November 2023).

Lam, J., Fang, E., Almansoori, M., Chatterjee, R. and Soosai Raj, A.G. (2022) 'Identifying Gaps in the Secure Programming Knowledge and Skills of Students', *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education – Volume 1*. New York, NY, USA: Association for Computing Machinery. SIGCSE 2022, Vol.1, pp. 703–709. Available at: [10.1145/3478431.3499391](#) (Accessed: 28 September 2023).

LEXFO (2023) *Lexfo's security blog – XORTigate: Pre-authentication Remote Code Execution on Fortigate VPN (CVE-2023-27997)*. Available at: [Lexfo's security blog – XORTigate: Pre-authentication Remote Code Execution on Fortigate VPN \(CVE-2023-27997\)](#) (Accessed: 28 September 2023).

LinkedIn (2023) *How can you switch web development frameworks?*. Available at: [How can you switch web development frameworks?](#) (Accessed: 28 September 2023).

McConnell, S. (2004) *Code complete*. 2nd ed. Redmond, Wash: Microsoft Press.

Microsoft (2019) *Security Tips: Minimizing the Code You Expose to Untrusted Users*. Available at: [Security Tips: Minimizing the Code You Expose to Untrusted Users](#) (Accessed: 22 March 2024).

Microsoft (2021) *Security: Separation of Privilege*, [Microsoft Community Hub](#) Available at: <https://techcommunity.microsoft.com/t5/azure-sql-blog/security-separation-of-privilege/ba-p/2393637> (Accessed: 28 September 2023).

Mishan, E.J. and Quah, E. (2020) *Cost-Benefit Analysis*. 6th edn. Sixth edition. | Milton Park, Abingdon, Oxon; New York: Routledge, 2020.: Routledge. Available at: [10.4324/9781351029780](#) (Accessed: 22 March 2024).

MITRE (2023) *CWE – 2023 CWE Top 25 Most Dangerous Software Weaknesses*. Available at: [CWE – 2023 CWE Top 25 Most Dangerous Software Weaknesses](#) (Accessed: 23 January 2024).

MITRE (2023) *Stack Frame Canary Validation – Technique D3-SFCV* / MITRE D3FEND™. Available at: [Stack Frame Canary Validation – Technique D3-SFCV](#) (Accessed: 29 February 2024).

Morrison, P.J., Pandita, R., Xiao, X., Chillarege, R. and Williams, L. (2018) ‘Are vulnerabilities discovered and resolved like other defects?’, *Empirical Software Engineering*, 23(3), pp. 1383–1421. Available at: 10.1007/s10664-017-9541-1 (Accessed: 7 December 2018).

Opsview (2023) *Migrating between JavaScript frameworks* / Opsview. Available at: [Migrating between JavaScript frameworks | Opsview](#) (Accessed: 28 September 2023).

Outsystems (n.d.) *How to Reduce Attack Surface: Best Practices and Key Steps*. Available at: <https://www.outsystems.com/blog/posts/attack-surface/> (Accessed: 7 March 2024).

OWASP (2021) *Input Validation – OWASP Cheat Sheet Series*. Available at: [Input Validation – OWASP Cheat Sheet Series](#) (Accessed: 28 September 2023).

Simplilearn (2012) *Feasibility Study and Its Importance in Project Management*, [Simplilearn | Online Courses – Bootcamp & Certification Platform](#) Available at: [How to Conduct a Feasibility Study: Key Steps & Examples](#) (Accessed: 7 March 2024).

ZDI (last) (2021) *Zero Day Initiative – CVE-2021-26084: Details on the Recently Exploited Atlassian Confluence OGNL Injection Bug*, *Zero Day Initiative* Available at: [Zero Day Initiative – CVE-2021-26084: Details on the Recently Exploited Atlassian Confluence OGNL Injection Bug](#) (Accessed: 23 January 2024).

Zheng, W., Feng, C., Yu, T., Yang, X. and Wu, X. (2019) ‘Towards understanding bugs in an open source cloud management stack: An empirical study of OpenStack software bugs’, *Journal of Systems and Software*, 151, pp. 210–223. Available at: 10.1016/j.jss.2019.02.025 (Accessed: 13 June 2019).

PUBLISHED

28 January 2025

WRITTEN FOR

