Teaching Secure Software Development in Master's Studies: Course Design and Challenges

Petar Rajković[1,*,†] and Dragan Janković[1,†]

[1] University of Niš, Faculty of Electronic Engineering, Aleksandra Medvedeva 4, Niš, Serbia

Abstract

Secure software development has become a critical competence in modern engineering practice, driven by the rapid proliferation of cyber threats, increasingly complex software ecosystems, and the growing dependence of society on digital infrastructure. This paper analyzes the structure, objectives, and pedagogical approach of the course Design and Implementation of Secure Software, taught within the master's program at the University of Niš, Faculty of Electronic Engineering. Building on the official curriculum, this study provides a conceptual overview of course topics, a pedagogical rationale for their inclusion, recommendations for strengthening teaching methodologies, and alignment with international best practices and industry standards. By integrating theory, hands-on exercises, and applied cryptographic and secure coding techniques, the course effectively prepares students for real-world security engineering. Recommendations for further curriculum modernization—such as integrating DevSecOps, threat-modeling workshops, and automated code-analysis tools—are also provided.

Keywords

Secure software design, code vulnerabilities detection, code execution analysis

1. Introduction

Software systems are among the major building blocks of critical infrastructure, commercial services, and personal communication. As a consequence, secure software design, along with software security, becomes a separate discipline in computer science and a universal requirement in industry. Traditional software engineering education commonly emphasizes algorithms, data structures, and design patterns, while neglecting thinking about possible attacks and hidden vulnerabilities. Thus, pointing out secure coding principles and threat modeling became skills necessary to prevent breaches and exploitation of active live systems.

Software vulnerabilities are among the primary causes of modern security breaches. While traditional software engineering focuses on functionality and performance, today's digital ecosystem demands software that is secure by design. Attacks such as SQL injection, buffer overflows, credential stuffing, supply-chain compromise, and cryptographic implementation errors routinely exploit weaknesses introduced during development.

Master-level courses in secure software development serve to introduce new chapters by teaching students not only to write functioning software but also to write resilient, robust, and attack-aware software systems. The course Design and Implementation of Secure Software [1] is created with this purpose in mind: to provide students with an understanding of vulnerabilities, practical skills in secure implementation, and knowledge of the background theory that forms modern security mechanisms.

CEUR-WS.org/Vol-4333/paper1.pdf

NWISEd 2026: Workshop on Co-Creating New Ways of Information Systems Education, July 1--3, 2026, Swansea, United Kingdom

* Corresponding author.

† These authors contributed equally.

petar.rajkovic@elfak.ni.ac.rs (P. Rajković); dragan.jankovic@elfak.ni.ac.rs (D. Janković)

0000-0003-4998-2036 (P. Rajković); 0000-0003-1198-0174(D. Janković)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-19

2. Related Work

Research and pedagogical literature on secure software engineering consistently emphasize the need to integrate security principles early in the software development lifecycle. For us, the key works are McGraw's books [2][3]which introduced the concept of “software security” as a discipline distinct from traditional network or operating system security, arguing that secure design must be embedded in architectural design, requirements engineering, and coding methods and practices. This view has influenced numerous academic courses and industry frameworks that emphasize software assurance from the earliest stages of design.

Howard and Lipner’s Security Development Lifecycle (SDL) [4] represents another considerable foundation stone in the field, formalizing Microsoft’s internal security practices into a standardized process. Subsequently, many universities and organizations adopted SDL principles, including threat modeling, secure coding checklists, and mandatory code reviews, as necessary components of secure development curricula. These practices closely align with the objectives of contemporary master ’s-level courses, which emphasize an engineering-driven, process-oriented approach to application security.

Complementing process-oriented frameworks, the CERT Secure Coding Standards, developed by Carnegie Mellon University’s Software Engineering Institute [5] provide rigorous, language-specific rules designed to eliminate common coding defects. These standards have established themselves as one of the main topics of academic curricula focused on memory-unsafe languages like C and C++, where vulnerabilities such as buffer overflows, integer wrapping, and use-after-free cause significant risks. Many secure coding courses adopt these standards to provide concrete guidance and promote consistent coding hygiene.

In the general academic context, secure software engineering education has been influenced by courses such as MIT’s 6.858 [6] which integrates research papers and hands-on exploitation laboratories to teach system-level security concepts. Similarly, programs at ETH Zürich and EPFL prioritize formal techniques and conceptual checks, mirroring ongoing acidic approaches that advocate for mathematical correctness and verification in secure system design [7]. These examples show the range of pedagogical approaches, from systems-centric to mathematically formal models, employed internationally to prepare students for complex cybersecurity challenges.

Recent studies on cybersecurity education underline the challenge of linking theory and practice. Researchers contend that secure coding should be taught alongside practical case studies, vulnerability analysis exercises, and the practical application of cryptographic APIs [8]. The results support pedagogical strategies that integrate basic knowledge with applied tools and hands-on laboratories, ensuring that students understand both the importance of security and the practical methods for effective protection.

Taken together, these works do not represent isolated approaches but rather complementary dimensions of secure software engineering education. McGraw’s conceptualization of software security establishes the need for security to be integrated throughout the development lifecycle, while Howard and Lipner’s SDL operationalizes this idea into a structured, process-oriented methodology. The CERT Secure Coding Standards further concretize these principles at the implementation level by providing enforceable rules for eliminating common vulnerabilities. In parallel, academic courses such as MIT’s 6.858 and programs at ETH Zürich and EPFL demonstrate how these conceptual and procedural elements can be translated into effective teaching practices, ranging from system-level analysis to formal verification.

This layered perspective—spanning conceptual foundations, development processes, concrete coding practices, and pedagogical strategies—directly informs the design of our course. Specifically, our curriculum integrates: (1) security-by-design thinking inspired by McGraw, (2) lifecycle-oriented development practices aligned with SDL, (3) hands-on secure coding guided by CERT standards, and (4) a balanced teaching methodology combining theory, tools, and project-based learning as reflected in leading academic programs. This synthesis ensures that the course is not a collection of topics, but

a coherent framework that mirrors the multi-layered nature of real-world secure software development.

3. Objectives, Content, and Competencies

The official course description outlines primary learning objectives, including fostering in-depth knowledge of fundamental security concepts in software design, identifying common vulnerabilities, and applying secure coding principles and strategies to reduce them or keep them as controllable as possible. The course structure is envisioned to help students develop proficiency in secure coding standards, cryptographic algorithms, authentication procedures, secure input validation, and the secure management of memory, concurrency, and configuration. The course should influence students' approach to work by raising awareness of the responsible handling of sensitive data to ensure confidentiality and integrity among an enterprise's interconnected software systems.

Such a set of objectives could be defined at the master's level, since students are expected to have strong basic knowledge of programming, operating systems, and computer networks. Our students' backgrounds are bachelor 's-level in computer science from various technical universities and faculties (mostly from Serbia, since we have a minor number of international students), so we can place greater emphasis on applied techniques and practical software implementation approaches. In contrast to undergraduate security courses, this curriculum prioritizes programming practices when developing defense mechanisms and prepares students for professional roles such as secure software engineer, application security analyst, security architect, DevSecOps practitioner, application-layer code auditor, and penetration tester. The explicit inclusion of topics such as buffer overflows, pseudorandom generators, memory-management pitfalls, and deadlock prevention is technically sound and distinguishes the course from those focused more on abstract or theoretical security offerings.

The course is focused on applied knowledge and skills necessary to develop secure software in both execution and access. Besides, there are courses focused only on one of these two components. Initially, we focused only on secure coding techniques, since our students have a course in their bachelor's studies that is fully focused on cryptographic algorithms. During this time, we decided to move toward efficiently covering both groups of topics that can be assigned to software security. Focusing on secure coding techniques without considering cryptography makes one feel there is an incomplete story.

Upon completion, students will be able to identify common vulnerabilities, implement secure coding principles, design authentication workflows, appropriately use cryptographic components, defend against memory-safety attacks, validate input to prevent injection attacks, and address race conditions and other concurrency-related issues. The concrete actions our students will take are to optimize a safe environment for sensitive data and configurations. These competencies follow widely recognized industry frameworks, including the OWASP Top 10, the NIST Secure Software Development Framework (SSDF), Microsoft's Security Development Lifecycle (SDL), and ISO/IEC 27001 secure development guidelines.

To support the approach to realistic cases, we introduced well-known scenarios that resulted in serious problems and were followed by successful solutions, leading to practices now considered best practices (Table 1). The course begins with the foundations of secure coding, introducing core security principles, adversarial thinking, and an overview of use cases that provide background for designing software with explicit threat models. Initial modules explain fundamental concepts such as least privilege, defense in depth, and common attack patterns. This core knowledge extends to password and authentication security, where students analyze modern password storage mechanisms, including hashing, salting, and key derivation functions such as PBKDF2, bcrypt, and Argon2. Collectively, these topics establish the conceptual basis for subsequent modules on cryptographic and system security.

The curriculum then addresses randomness, memory safety, and low-level security concerns, which are important to understand the origins of vulnerabilities in practical systems. Students

examine the design and limitations of cryptographically secure pseudorandom generators, the architecture of secure random number generators such as Fortuna, Yarrow, and NIST SP 800 90 DRBGs, and the importance of entropy in secure protocols. Memory-focused modules explore buffer overflow exploitation techniques, including stack- and heap-based attacks and return-oriented programming, followed by secure memory management practices such as preventing use-after-free and double-free errors, zeroizing sensitive data, and employing safer abstractions in modern programming languages.

The final thematic units of the course address concurrency, data control, encryption schemes, and secure deployment. Students analyze deadlocks, race conditions, time-of-check to time-of-use (TOCTTOU) vulnerabilities, and secure file-handling strategies to understand the impact of resource conflicts on security in multi-threaded and distributed systems. Modules on input validation and data processing cover injection prevention, trust boundaries, secure parsing, and effective management of logs, temporary files, and secrets. The curriculum concludes with an examination of security procedures and protocols, including symmetric and asymmetric encryption, secure key exchange, HMACs, authentication frameworks such as Kerberos and OAuth2, and secure configuration and deployment practices. These concluding units underscore that software with solid cryptographic foundations may remain vulnerable if misconfigured, misused, or deployed insecurely, thereby integrating the full spectrum of secure design and development considerations.

Table 1
Comprehensive Comparison Table of Course Modules & Case Studies

Module	Topics Covered	Real-World Case Study (Year)	Case Study Summary & Relevance
Password Handling and Authentication Security	• Hashing and Salting • Common weaknesses: password reuse, weak policies, plaintext storage	LinkedIn Password Breach (2012) [9]	6.5 million passwords were leaked due to unsalted SHA-1 hashing. Attackers cracked most hashes quickly. Demonstrates why proper hashing, salting, and modern KDFs are essential.
Pseudorandom Number Generation	• PRNG vs. CSPRNG • Entropy sources • Common implementation flaws	Debian OpenSSL Vulnerability (2008) [10]	The removal of two lines of OpenSSL code drastically reduced entropy. All cryptographic keys generated for the past two years have become predictable. Highlights the importance of handling entropy correctly and conducting code review.
Buffer Overflow Vulnerabilities	• Stack & heap overflows • Return-oriented programming (ROP) • Mitigations: ASLR, DEP, stack canaries	Heartbleed (2014) [11]	A bounds-checking failure in OpenSSL allowed attackers to read server memory, exposing keys and sensitive data. Shows how small memory errors can compromise global infrastructure.
Memory Management & Secure Coding in Low-Level Languages	• Safe allocation/deallocation • Avoiding use-after-free & double-free • Secure memory wiping	Firefox Use-After-Free Vulnerabilities (2023-2025) [12]	Mozilla frequently patches UAF bugs, which can enable remote code execution. Demonstrates the complexity of manual memory management and the need for secure coding practices.

Concurrency, Deadlocks & Race Conditions	• Thread synchronization • File locking • TOCTTOU flaws	Chrome Sandbox Race Condition (2019) [13]	A race condition in Chrome's IPC system allowed privilege escalation. Shows that concurrency bugs are security vulnerabilities, not just correctness issues.
Input Validation & Injection Attacks	• Whitelisting vs. blacklisting • Command injection, SQL injection, XSS • Framework-level validation	Equifax Breach (2017) [14]	Caused by an Apache Struts OGNL injection vulnerability. Attackers accessed massive amounts of data. Reinforces the importance of proper input sanitization and patching.
Cryptography & Authentication Protocols	• Symmetric/asymmetric cryptography • HMAC, digital signatures • TLS basics • Kerberos, OAuth2	Microsoft Teams OAuth Vulnerability (2020) [15]	Misconfigured OAuth flow enabled account hijacking. Shows that crypto failures often occur in integrations, not algorithms.
Secure Configuration & Secrets Management	• Secure defaults • Avoiding hard-coded secrets • Environment-based configuration • Protecting config files	Uber AWS Secret Exposure (2016) [16]	AWS credentials were accidentally uploaded to GitHub, exposing 50+ million records. Emphasizes secure handling of secrets and configuration files.

4. Content Overview and Thematic Structure

The course consists of lectures, exercises, and active student participation, with assessments distributed among projects, presentations, and final or oral examinations. This structure addresses both the conceptual and practical aspects of secure software development.

Effective instruction in secure software development requires a combination of conceptual understanding, acceptance of proper analytical methodology, and practical experience. Teaching such a course should be based on appropriate pedagogical approaches for master 's-level studies. It should find a delicate balance between the theoretical, experiential, and practical dimensions of secure development. Even more important, use teaching and knowledge-delivery approaches that most effectively promote critical thinking, procedural knowledge, and the application of security principles throughout various systems and software architectures. At this level, it is not only important to bring precise technical knowledge, but also to provide a complete environment in which the designed software should work and be aware of all potential threats.

One of the approaches we used could be called Threat Modeling Workshops. Threat modeling workshops present a formalized approach for analyzing and understanding security risks as early as possible in the development lifecycle. These activities enable students to identify assets, attackers, trust boundaries, and weakest and attack points within a system. Students also construct data flow diagrams to visualize how data flows through applications and to pinpoint potential vulnerabilities.

Students are also introduced to established methodologies such as STRIDE and LINDDUN, which support systematic threat classification and address problems arising from both security and privacy attack points. Developing risk-reduction strategies before implementation provides direct experience with how design decisions affect software security outcomes. This approach helps students better

understand industry practices, in which threat modeling is a key element of modern security engineering.

The use of Automated Security Tools is another real-world element we brought to our course. Integrating automated security tools into coursework aligns with current software development practices, where tool-assisted analysis is important for identifying and preventing vulnerabilities. Students use static analysis tools, such as SAST and SonarQube, to detect insecure coding patterns and create fixes and acceptable workarounds. Additionally, hands-on experience with fuzz testing tools such as AFL and libFuzzer enables students to uncover memory corruption issues and unexpected program behaviors by randomized input generation.

Dynamic analysis tools such as OWASP ZAP familiarize students with active scanning techniques for identifying runtime vulnerabilities. Dependency analysis tools, including Dependabot and Trivy, highlight risks associated with third-party libraries and outdated packages. Including these tools in assignments and running them demonstrates how automation perfectly complements manual review and supports continuous security integration in modern DevSecOps pipelines.

Project-Based Learning is the key concept we promote in our course. Each student has a main project, which accounts for the most important part of their score, and participates with colleagues in case-study analysis. A capstone project enables students to integrate techniques and knowledge acquired throughout the course into a comprehensive practical exercise and case-study evaluation. Students are commonly required to develop a functional application and conduct a complete security assessment of their work. This process consists of identifying vulnerabilities using the methodologies and tools presented in the course, hardening the application according to best practices, and documenting all their design decisions and implemented security measures.

These projects improve students' technical knowledge and essential communication skills by requiring them to identify and communicate risks, justify design choices, and explain risk reduction tactics. Project-based learning reinforces the iterative nature of secure development and underscores the importance of integrating security considerations throughout the software lifecycle, from initial design to deployment.

5. Discussion and Comparison with International Practices

Leading institutions such as MIT [17], Carnegie Mellon [18], ETH Zürich [19], and KU Leuven [20] prioritize secure software development by offering detailed lectures and serious assignments on memory safety, coding standards, cryptography, system security, and thorough monitoring of code execution and exploitation. In its current form, our course follows a similar philosophy in core areas. Similarly, the current course composition includes concurrency, randomness, and secure configuration, ensuring a high technical level. We see the room for further improvement by adding emerging topics, including cloud-native security, container and Kubernetes isolation policies, Rust-based secure coding, and modern software supply chain protections such as SBOMs (Software Bill of Materials) and SLSA (Supply-chain Levels for Software Artifacts. These additions will position our course alongside top programs and make it more engaging for both students and software engineers looking to deepen their knowledge.

The comparative analysis can be interpreted through the same layered framework identified in the related work: conceptual foundations, development processes, implementation practices, and pedagogical approaches. The surveyed institutions emphasize different layers of this spectrum, highlighting the necessity of integrating them into a single course design.

Table 2.
Comparison of the major-related curricula to our course

Institution	Primary Focus Areas	Distinctive Strengths	Comparison to Our Course
Massachusetts Institute of Technology (MIT) *6.858 – Computer Systems Security*	• OS & systems security • Information flow & capabilities • Language-based security • Hardware, network, and web security	• Strong research orientation • Integrates cutting-edge papers • System-level exploitation & defense	Broader in system security. MIT emphasizes full-system threat modeling and advanced attacks; our course focuses more on secure coding, memory safety, crypto implementation, and vulnerability mitigation at the code level.
Carnegie Mellon University (CMU) – SEI *CERT Secure Coding in C and C++*	• Integer handling • Array & buffer misuse • Memory safety issues • Insecure I/O • CERT C/C++ secure coding standards	• Highly specialized in C/C++ • Direct application of CERT standards widely used in defense/industrial sectors	CMU's course is narrower and industry-focused. Our course is broader academically (covering passwords, PRNGs, and protocols) and not limited to C/C++. CMU provides deep specialization rather than systemic coverage.
ETH Zürich / EPFL *MSc in Cyber Security (full program)*	• Applied cryptography • Network & system security • Hardware security • Formal methods • Cross-institutional advanced modules	• Full-spectrum cybersecurity education • Heavy emphasis on mathematical rigor & formal verification • Collaboration between ETH & EPFL	ETH's program is much broader (an entire degree vs. a single course). Our course fits the "secure software implementation" niche that ETH covers across many specialized modules.
KU Leuven *Development of Secure Software (H04K5A)*	• Threats, risks, countermeasures • Web & low-level vulnerabilities • Authentication & access control	• Strong focus on architectural security and secure process integration	Difference: KU Leuven covers more architectural patterns, while our course provides greater depth on low-level memory issues, PRNGs, and cryptographic implementation details.

Besides, our course covers fundamental concepts such as buffer overflow analysis, cryptographic implementation, and secure programming methods; we see technology-specific topics as an additional improvement path. The course could be further enhanced by:

- Including DevSecOps and CI/CD pipeline security
- Covering modern languages such as Rust for memory safety
- Incorporating cloud native security concepts (container security, secrets management, Kubernetes policies)

The four courses we used as inspiration and comparison targets — MIT 6.858, CMU SEI Secure Coding, ETH/EPFL's Cyber Security MSc, and KU Leuven's Development of Secure Software — are a bit different, as they reflect different teaching philosophies.

MIT's 6.858 provides a research-oriented, systems security–driven perspective. It is focused on a deep understanding of threat models, exploitation techniques, and secure system architecture. Students base their work on the latest research papers and complete labs that involve exploiting and

defending real systems. This positions MIT's course toward students seeking a strong systems-theoretic and offensive/defensive mix at the architectural and OS level.

CMU SEI's Secure Coding in C and C++ stands at the opposite end: coding-focused, industry-driven expertise in memory-unsafe languages. Rather than broad coverage, it focuses on robust coding patterns grounded in the CERT Secure Coding Standards. Its hands-on exercises demonstrate how real-world vulnerabilities arise directly from the misuse of C/C++ constructs. Having this in mind, the course is particularly relevant to the embedded systems, defense, and industrial software sectors.

ETH Zürich and EPFL's joint MSc in Cyber Security offers a full-spectrum, two-year program. It covers applied cryptography, networks, systems, hardware, and formal methods. Unlike a single course, as in other examples, here a complete program is defined to help develop high-end security engineers capable of both research and advanced engineering, supported by cross-institutional faculty expertise.

KU Leuven's Development of Secure Software most closely aligns with a classical secure software engineering course model. It targets practical vulnerabilities, application security, authentication, and secure development practices. Supplementary courses extend coverage to architectural patterns and lifecycle security. In contrast to MIT or ETH's research-intensive approach, Leuven offers a balanced, structured, practice-oriented education grounded in the SDLC.

Our master's course is closest to KU Leuven's, as both focus on vulnerabilities, mitigation, and lifecycle integration. However, our course places greater emphasis on memory management, PRNG security, secure and cryptographic implementations, and correctness of low-level coding. At other institutions, these topics are addressed differently or spread across different modules.

The current evaluation system uses final exams (either written or oral), participation, regular exercises, and, most importantly, project and seminar work. However, these methods primarily assess conceptual comprehension rather than practical skills. Adding lab-based exams, secure code audits, CTF-style challenges, peer reviews, and written analyses of real incidents would better evaluate students' ability to detect, exploit, and remediate vulnerabilities in real-world scenarios. These additions would ensure students retain fundamental knowledge and apply secure development practices in realistic contexts.

Teaching secure development at the master's level is a very challenging academic endeavor. The most important points that need to be revised each year are:

1. Rapid evolution of threats – New vulnerabilities emerge constantly
2. Understanding complex modern systems – Cloud, distributed systems, microservices, and other layered systems bring different risk models.
3. Combining theory and practice – Students must understand cryptography mathematically, yet apply APIs safely.
4. Balancing breadth and depth – A limited course duration cannot cover all topics comprehensively.

Teaching secure software development at the master's level presents a range of pedagogical challenges, largely because the discipline evolves more rapidly than traditional software engineering. Unlike areas where core principles remain the same for decades, security demands that educators constantly refresh materials, adapt assessments, and prepare students for threats that may not have existed just months earlier. Because of all these factors, we must maintain constant innovation in course design, agility, and a curriculum that balances purely academic topics with real-world applicability.

Maintaining our course to match the expected level requires a flexible, regularly updated, and industry-aligned set of topics with a strong practical component. We believe that the most effective secure software development courses combine theory with experiential learning, incorporate well-known and standardized case studies, and focus on developing adaptable, lifelong learners.

6. Conclusion

Secure software development should be considered as one of the most important disciplines within modern computer engineering. In this paper, we present our vision for a course that should help future software engineers shift their mindset from a purely implementation-focused approach to designing systems that not only work correctly but also address various threats. The presented course offers a comprehensive introduction to core security principles, vulnerability analysis, cryptographic mechanisms, memory safety, concurrency challenges, input validation, and secure configuration practices.

When designing our course, we consulted many successful approaches and chose the four most influential. By combining their elements with our master's curriculum in software engineering, we aimed to design a course that would provide students with the knowledge needed to design and build resilient software systems. This paper proposes several enhancements, including adopting DevSecOps practices, integrating automated security tools, and expanding project-based learning, to better align the curriculum with current industry requirements.

The course design is grounded in a synthesis of established research perspectives, combining conceptual, process-oriented, and implementation-level principles into a coherent educational framework. Overall, the course constitutes a robust and relevant component of master 's-level studies in computer science and serves as an effective platform for preparing professionals capable of designing and implementing a more secure digital ecosystem.

Acknowledgements

This work has been supported by the Ministry of Science, Technological Development and Innovation of the Republic of Serbia [Grant Number: 451-03-137/2025-03/ 200102]

Declaration on Generative AI

During the preparation of this work, the authors used Grammarly to check grammar and spelling.

References

[1] Secure software design, course description, Master studies accreditation 2019, https://www.elfak.ni.ac.rs/downloads/akreditacija-2019/mas/rii-eng/secure-software-design.pdf

[2] Software security engineering: A guide for project managers, Addison-Wesley, Upper Saddle River, N.J, 2008.

[3] J. Viega, G. McGraw, Building Secure Software: How to Avoid Security Problems the Right Way, Portable Documents, Pearson Education, Limited, 2021.

[4] S. Lipner, M. Howard, The Security Development Lifecycle, Microsoft Press, 2006.

[5] R. Seacord, CERT C Secure Coding Standard, Pearson Education, Limited, 2021.

[6] 6.858: Computer Systems Security, https://css.csail.mit.edu/6.858/2019/general.html, 2019

[7] A. Perrig, P. Szalachowski, R. M. Reischuk, L. Chuat, SCION: A Secure Internet Architecture, Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-67080-5.

[8] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, C. Stransky, Comparing the Usability of Cryptographic APIs, in: 2017 IEEE Symposium on Security and Privacy (SP), IEEE, 2017. doi:10.1109/sp.2017.52.

[9] LinkedIn Corporation. LinkedIn Password Breach Incident Report (2012). 2012. Available at: https://arxiv.org/pdf/1703.06586 [arxiv.org]

[10] Debian Project Security Team. Debian OpenSSL Predictable Random Number Vulnerability Advisory (DSA 1571 1). 2008. Available at: https://www.debian.org/security/2008/dsa-1571

[11] OpenSSL Software Foundation. OpenSSL Heartbleed Vulnerability Report (CVE 2014 0160). 2014. Available at: https://heartbleed.com/
[12] Mozilla Security Response Team. Firefox Use After Free Vulnerability Series Reports (2023–2025). Available at: https://www.mozilla.org/en-US/security/advisories/
[13] Chromium Security Team. Chrome Sandbox Race Condition Advisory (2019). Available at: https://chromereleases.googleblog.com/
[14] Equifax Inc. Equifax Cybersecurity Incident Report (2017). Available at: https://www.equifaxsecurity2017.com/
[15] Microsoft Security Response Center (MSRC). Microsoft Teams OAuth Token Security Advisory (2020). Available at: https://msrc.microsoft.com/
[16] Uber Technologies, Inc. Uber AWS Credential Exposure Incident Report (2016). Available at: https://www.ftc.gov/system/files/documents/cases/1523054ubertechnologiescmpt.pdf
[17] MIT OpenCourseWare. (2014). Computer Systems Security (6.858). Massachusetts Institute of Technology. Retrieved from [csdiy.wiki]
[18] Software Engineering Institute. (2023). CERT Secure Coding in C and C++ Professional Certificate. Carnegie Mellon University. Retrieved from [reg131.imp...risoft.com]
[19] EPFL & ETH Zürich. (2026). Cyber Security Study Plan – ETH Zürich Courses. Retrieved from [ethz.ch]
[20] KU Leuven. (2026). Development of Secure Software (H04K5A). Retrieved from [eng.kuleuven.be]