applied sciences

MDPI

Article

Enhancing Security in Industrial Application Development: Case Study on Self-Generating Artificial Intelligence Tools

Tomás de J. Mateo Sanguino

Escuela Técnica Superior de Ingeniería, Universidad de Huelva, Av. de las Artes, s/n, 21007 Huelva, Spain; tomas.mateo@diesia.uhu.es; Tel.: +34-959-217665

Abstract: The emergence of security vulnerabilities and risks in software development assisted by self-generated tools, particularly with regard to the generation of code that lacks due consideration of security measures, could have significant consequences for industry and its organizations. This manuscript aims to demonstrate how such self-generative vulnerabilities manifest in software programming, through a case study. To this end, this work undertakes a methodology that illustrates a practical example of vulnerability existing in the code generated using an AI model such as ChatGPT, showcasing the creation of a web application database, SQL queries, and PHP server-side. At the same time, the experimentation details a step-by-step SQL injection attack process, highlighting the hacker's actions to exploit the vulnerability in the website's database structure, through iterative testing and executing SQL commands to gain access to sensitive data. Recommendations on effective prevention strategies include training programs, error analysis, responsible attitude, integration of tools and audits in software development, and collaboration with third parties. As a result, this manuscript discusses compliance with regulatory frameworks such as GDPR and HIPAA, along with the adoption of standards such as ISO/IEC 27002 or ISA/IEC 62443, for industrial applications. Such measures lead to the conclusion that incorporating secure coding standards and guideline—from organizations such as OWASP and CERT training programs—further strengthens defenses against vulnerabilities introduced by AI-generated code and novice programming errors, ultimately improving overall security and regulatory compliance.

Keywords: self-generative vulnerability; AI-based code generation; security risk; ChatGPT; software development; industrial application

Citation: Mateo Sanguino, T.d.J. Enhancing Security in Industrial Application Development: Case Study on Self-Generating Artificial Intelligence Tools. *Appl. Sci.* **2024**, *14*, 3780. https://doi.org/10.3390/app14093780

Academic Editor: Paolino Di Felice

Received: 11 April 2024
Revised: 24 April 2024
Accepted: 26 April 2024
Published: 28 April 2024

1. Introduction

In today's industrial landscape, programmers play a pivotal role in the development and optimization of industrial applications. As an example, the demand for software developers among automobile manufacturers has grown 200% over just four years, surpassing the average demand for other job roles in the sector [1]. Moreover, 91% of manufacturing companies have increased their digital transformation investments—such as the Internet of Things and data analytics—to enhance efficiency and productivity, necessitating programming skills [2]. Surveys conducted by the Federation of German Industries (BDI) underscore the scarcity of information and communication technology (ICT) skills, including programming, which stands as a significant barrier to the adoption of digital technologies in the industry [3]. Consequently, programmers are indispensable in automating processes, integrating systems, analyzing vast datasets, and ensuring the smooth operation of industrial machinery, ultimately driving innovation and competitiveness in the industrial sector [4].

Nevertheless, novice programmers tend to exhibit an average rate of 16 errors per 100 lines of code, a statistic influenced by factors such as code complexity and the programmer's skills [5]. In certain instances, particularly with more experienced programmers, this error rate can be halved [6]. Despite the decrease in error frequency with increasing experience, errors can still have dramatic consequences for organizations (e.g., financial transactions, medical software, aircraft systems, or the energy grid, among others). These examples, regardless of the industry, can sometimes have irreversible effects for the organizations and stakeholders they serve.

Artificial General Intelligence (AGI) models can generalize, learn, and understand like humans. This makes current generative artificial intelligence (AI) systems “competent AI”, meaning that they are better at a limited set of tasks than 50% of people [7]. In this context, the advent of online self-generative tools—like ChatGPT—have facilitated the widespread adoption of AI by programmers, who increase its usage by creating code within their organizations. This democratization has brought a new potential security gap for the industry that still goes unnoticed, which aligns with the concerns expressed by other researchers [8].

In this context, new roles emerge such as the “prompt engineer”, a person that is involved in understanding the capabilities and limitations of the AI system, as well as knowing how to effectively formulate questions to obtain useful and relevant responses. This may involve selecting keywords, tone of the question, grammatical structure, and other language aspects that influence the response generated by the AI. Nonetheless, becoming a prompt engineer also entails certain risks. These risks include the misuse of AI, which can lead to application risks and mistrust in its technology, which could negatively affect its adoption and use [9].

For programming tasks, a large language model (LLM) can be used to self-generate source code, based on descriptions or instructions in natural language. LLMs are AI systems that undergo training on extensive textual datasets, to comprehend language patterns and structures [10]. Theoretically, self-generative tools can help programmers write code faster, more efficiently, and with fewer bugs, thus increasing their productivity. However, this is not always necessarily true.

Consequently, LLMs introduce their own challenges, encompassing concerns related to efficiency, security, or adherence to programming best practices. LLMs do not inherently generate secure code, unless explicitly denoted (i.e., they are designed mainly to fulfill their functionality). This happens because most self-generative tools, trained to date, do not consider possible vulnerabilities. Accordingly, LLMs default to adopting code that is prevalent on the Internet, which may lack considerations for security [11]. Therefore, the use of LLMs in code generation must require careful oversight and review by human developers, a condition that is not consistently met [12]. Recently, the generation of disentangled representations in neural networks has emerged as a promising strategy to improve the comprehensibility and robustness of AI models [13]. This approach could address some of the limitations identified in AI-based code generation tools, by enhancing the quality and coherence of the generated code.

According to this, Table 1 shows a comparison of the main characteristics of the most representative online self-generative tools. In general, their ability to perform programming tasks includes basic code, functions, classes, data structures, interfaces, libraries, and applications for a wide range of programming languages (e.g., Python, JavaScript, Java, C++, C#, or Swift). Additionally, they can also solve incomplete or incorrect code, generate new functions for an existing program, and identify and repair errors in the code. In addition, they can also perform more specific tasks such as generating code for algorithms and data structures (e.g., trees, lists, stacks, queues, and arrays), generating code for graphical user interfaces (e.g., windows, buttons, and menus), as well as generating code for relational and non-relational databases.

Table 1. Main features of the current online self-generative tools.

Provider	Tool	LLM	Data Size	Availability	Year
OpenAI (San Francisco, CA, USA)	ChatGPT-3 (https://openai.com/chatgpt)	Transformer	100 terabytes	Public	2020
Tabnine (Tel Aviv, Israel)	Tabnine (https://www.tabnine.com/)	CodeDaVinci	25 petabytes	Public	2022
GitHub (San Francisco, CA, USA)	Copilot (https://github.com/features/copilot)	Transformer	100 petabytes	Public	2022
Microsoft and NVIDIA (Santa Clara, CA, USA)	Megatron (https://github.com/NVIDIA/Megatron-LM)	Turing NLG	1.75 exabytes	Limited	2022
Microsoft AI (Redmond, WA, USA)	DeepCoder (https://snyk.io/platform/deepcode-ai/)	Transformer-Decoder	1.2 exabytes	Public	2022
Codesmith Software (Los Angeles, CA, USA)	CodeSmith (https://www.codesmith.io/)	LaMDA	1.2 exabytes	Public	2023
Meta (Menlo Park, CA, USA)	Code Llama (https://huggingface.co/codellama)	Llama 2	1.5 exabytes	Public	2023
Google AI (Mountain View, CA, USA)	Bard/Gemini (https://gemini.google.com/app)	LaMDA	1.56 exabytes	Public	2023
GitHub and Google AI	Codex (https://openai.com/blog/openai-codex)	Transformer	1.6 exabytes	Public	2023
OpenAI	ChatGPT-4	Transformer	45 terabytes	Public	2023
OpenAI	ChatGPT-4.5	Transformer	75 terabytes	Beta	2023
DeepMind (London, UK)	AlphaCode (https://alphacode.deepmind.com/)	Transformer	100 exabytes	Beta	2023
Microsoft AI	Ada	Transformer	1.2 exabytes	Beta	2024

For a deeper understanding, a comprehensive review of resources relevant to LLMs can be found in the literature, with specific emphasis on ChatGPT [14]. The content covers a wide range of topics, including milestone documents, frameworks for LLM training and implementation, and tutorials for using these models effectively. This compilation serves as a valuable resource for people interested in exploring and learning more about the details covering frameworks for LLM training and implementation, tutorials for practical implementation, and a complete overview of the current landscape through leaderboards and visualization.

Research Hypothesis and Objectives

This manuscript focuses on the study of the prevalence and effects of vulnerabilities with self-generative tools in software development, which are particularly relevant for industrial applications where reliability and safety are crucial. Industrial environments often rely on software to monitor complex processes, machinery, and systems [15]. Software failures can have serious consequences—including workplace accidents, production disruptions, and financial losses—highlighting the importance of software security.

LLMs represent a paradigm shift in code generation, offering efficient and automated code creation capabilities. The selection of LLMs as the focal point of this study is justified, due to its increasing adoption in various fields, including industry. For this reason, the approach followed in this work involves the review of a case study related to an LLM. By examining a vulnerability associated with a self-generated tool, this work aims to provide valuable information for engineers tasked with implementing control and monitoring systems in industrial environments, thereby contributing to greater reliability and security in industrial applications. In this regard, the present manuscript proposes the term "self-generative vulnerability", to characterize scenarios where AI-based tools can facilitate the automatic creation of security weaknesses, as evidenced in the methodology and experimentation carried out. Specifically, the implications included in this study cover a security vulnerability related to SQL injection attacks.

While the notion of AI-generated code having security issues may not be groundbreaking in itself, the novelty of this manuscript lies in its practical demonstration of how such vulnerabilities manifest, particularly through the provided case study involving the MySQL and PHP code generated by ChatGPT. The research goes beyond simply identifying security issues and provides detailed information on how these issues can arise in real-world scenarios. It provides specific recommendations to address these concerns, including training programs, error analysis, responsible practices, tool integration, software development audits, regulation and legislation, and collaboration with third parties. Ultimately, the manuscript's value lies in its pragmatic approach and proposed solutions for mitigating security risks in AI-enabled development.

The hypothesis posits that early implementation of secure development practices can reduce the risk of exploitation, thereby improving the resilience of software systems against self-generated vulnerabilities. For this reason, the study aims to recommend the adoption of a comprehensive security framework in the context of AI-generated code development, including regulatory compliance, adoption of industry standards, continuous monitoring, and robust security testing.

To this end, this manuscript is structured as follows: Section 2 analyzes the underlying causes behind the use of new ICT in industrial applications. Section 3 presents the materials and methods used in an example of a self-generated vulnerability. Section 4 shows an example of an SQL injection attack. Section 5 analyzes traditional security methodologies versus AI-based code generation approaches and discusses security measures in industrial environments. Finally, this manuscript presents its conclusions and future work prospects.

2. Related Work

Experience shows that the rapid development of industrial applications introduces new, previously unidentified vulnerabilities, threats, and risks. As self-generated tools become an integral part of today's development, it is imperative to consider these tools separately to understand their potential vulnerabilities [16]. Analyzing security risks in AI-generated programming is crucial to ensure the integrity, confidentiality, and availability of software systems.

A lack of focus on security in education programs—especially in industrial engineering grades—is pointed out as a major contribution to security vulnerabilities, highlighting the need for adequate security training [17]. The learning difficulties faced by novice programmers, particularly those engaging in procedural programming for the first time, were addressed in a further study [18]. The study identified key problem areas, including poor planning and problem-solving abilities, a lack of programming language knowledge, limited understanding of the application domain, and difficulties in conceptualizing program execution. The solution involves the development of instructional strategies to help novices overcome these identified difficulties. Other study placed emphasis on obtaining information about students' deficiencies and misconceptions, by analyzing programming errors [19]. The findings revealed that a significant portion of errors is attributed to carelessness. Beyond that, students faced challenges primarily in strategic knowledge, specifically in problem-solving abilities.

The importance of errors in industrial applications as causes of security vulnerabilities is analyzed, finding a gap between developers' conceptual understanding of security, as well as their attitudes and practices related to personal responsibility for software security [20]. Later studies pointed to possible approaches to address this gap, such as the need for developers to ask the right questions and make strategic decisions when programming secure code [21]. Another work also highlighted the importance of including aspects related to the software, its social ecosystem, and related resources and tools [22]. In this sense, the approach proposed to use explainable AI in classifying suitable crops for precision agriculture offers an innovative perspective. This could improve the understanding and transparency of generated code, thereby contributing to the security and quality of industrial software [23].

The challenge of developing secure industrial applications was also investigated in experienced programmers, who make security-related errors despite their basic security knowledge. The results found patterns, indicating that simple mistakes are less usual, while vulnerabilities stemming from a misunderstanding of security concepts are more prevalent. Another study also addressed the gap between mid-level programmers making more sophisticated errors [24]. The study, focused on a semantic approach, revealed that those errors involved complex program constructs such as nested loops, arrays, recursion, and functions. In line with previous works, both studies suggest areas for improvement in security education, secure-programming APIs, documentation, and vulnerability-finding tools [25].

There is certainly a need for interactive tools to help engineers create more secure software, which is a common practice nowadays. For example, DeepState is presented as a tool that integrates vulnerability detection into the development cycle [26], while VulnEx has been proposed as a tool to audit applications, to identify and assess exposure to vulnerabilities due to the reuse of open-source software [27]. In this sense, LLMs have the potential to automate code generation, but their use must be combined with scalable defect elimination methods, to ensure the production of high-quality, secure code [28]. AI code generators are particularly vulnerable to data poisoning attacks, which can inject malicious samples into the training data and compromise the security of the generated code [29]. To address these challenges, AI techniques are increasingly being used for security risk assessment, with a focus on identifying and estimating cyber risks [30].

Additionally, collaborative efforts among engineers and cybersecurity experts are also considered essential to establish robust security protocols and to stay ahead for a secure industrial application environment [31]. This collaboration can be facilitated through security code reviews, which help engineers to ensure the security of their applications. The effectiveness of collaboration has been demonstrated in the context of a secure software engineering course with students from different backgrounds, which could also be extended to AI-generated programming [32].

3. Materials and Methods

This section aims to present a practical example of understanding vulnerabilities in code generated by an AI model (i.e., ChatGPT 3.5). The methodology covers the creation of a database, the definition of a table, the insertion of sample user data, and the secure storage of passwords. To do this, the resources involved a database for a web application, MySQL queries for database operations, and PHP for server-side logic.

Examples of industrial applications utilizing the components include inventory management systems, production tracking systems, and quality management systems, where web-based applications, SQL databases, and PHP queries play a significant role. For example, inventory management systems efficiently track stock levels and manage suppliers, leveraging SQL for database operations and PHP for server-side logic. Similarly, production tracking systems monitor schedules and manage inventory levels, using SQL for data extraction and PHP for dynamic functionalities. Quality management systems ensure product compliance, using SQL for data analysis and PHP for robust server-side logic.

In the context of self-configurable systems, these resources assume particular significance. That is, they not only facilitate data management and user interaction, but also have the capability to dynamically adapt to the evolving needs of industrial operations, thanks to the use of agents [33]. For instance, the web server can automatically adjust resource allocation to handle sudden traffic spikes, while the PHP code can adapt to modify existing behavior in response to changing system requirements. Similarly, the SQL database can be reconfigured to optimize performance or ensure data integrity in highly dynamic production environments.

3.1. Materials

The working model proposed is depicted in Figure 1, for which a detailed description of the typical components can be found in [34]. The process is as follows: upon entering a web address, the user's browser sends a request to the server, specifying the page name. The web server retrieves the corresponding PHP file and directs the PHP interpreter to process it. The PHP interpreter executes the PHP code, including MySQL database calls. After processing, the PHP interpreter sends the results back to the web server, which then delivers the HTML response to the user's browser. This process can be repeated by an attacker, to exploit a vulnerability in the target system.

To replicate the experiment, the installation of a MySQL database from the repository https://dev.mysql.com/downloads/mysql/ (accessed on 11 April 2024) is required, with the code statements being compatible with versions 5.0.13 onwards. A web server (e.g., Apache or Nginx) is also required to host the PHP files. Moreover, PHP version 7.x or higher can be utilized in conjunction with the MySQLi extension for interacting with MySQL databases. The MySQLi extension—short for MySQL Improved—is a PHP extension that provides an enhanced interface and security features. The MySQLi extension is typically included as part of the PHP installation package from the official website (https://www.php.net/downloads (accessed on 11 April 2024)) and can be enabled in the PHP configuration file (i.e., php.ini).

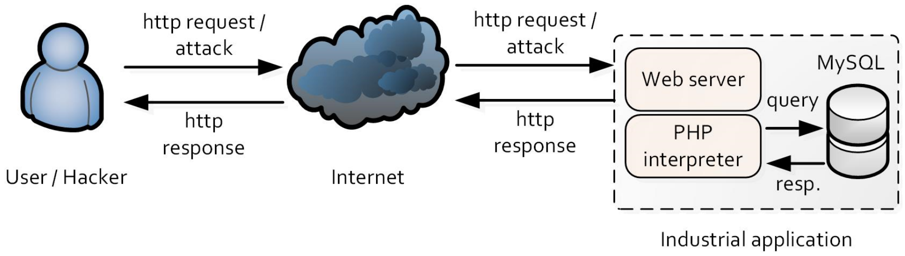
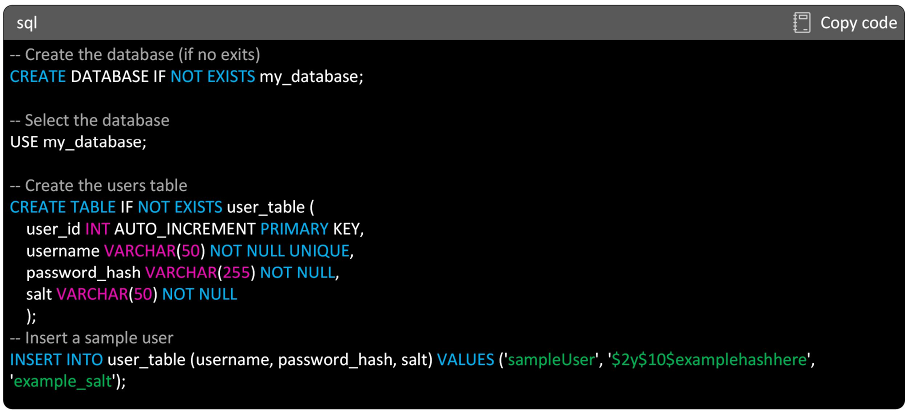


Figure 1. Components of the working model for an industrial application.

3.2. Methodology for Database Establishment

Figure 2 includes a representative example of SQL code used to create the database and the table, as well as to simulate the insertion of user data. The SQL script was generated by ChatGPT based on the following query:

- *"Generate a script to set up a database named 'my_database' and a table named 'user_table' in MySQL. The user table should have fields for user ID, unique username, password hash, and salt. Also, include a sample insertion to add a user named 'sampleUser' with an encrypted password and salt"*.



```sql
-- Create the database (if no exits)
CREATE DATABASE IF NOT EXISTS my_database;

-- Select the database
USE my_database;

-- Create the users table
CREATE TABLE IF NOT EXISTS user_table (
   user_id INT AUTO_INCREMENT PRIMARY KEY,
   username VARCHAR(50) NOT NULL UNIQUE,
   password_hash VARCHAR(255) NOT NULL,
   salt VARCHAR(50) NOT NULL
   );
-- Insert a sample user
INSERT INTO user_table (username, password_hash, salt) VALUES ('sampleUser', '$2y$10$examplehashhere',
'example_salt');
```

Figure 2. SQL code for creating a database, provided by ChatGPT.

Initially, the code checks if a database named "my_database" exists, creating it if not. Then, the code selects this database for subsequent operations. A table named "users" is created if it does not already exist, defining its structure to store user information, including a unique identifier (user_id), non-null and unique username, password hash, and salt value. A sample user is inserted into the "users" table with simulated values. A comment notes that the password has been hashed and salted in PHP, highlighting the importance of securely storing passwords. Additionally, the code provides commented PHP lines exemplifying how to verify a password. This code snippet demonstrates database setup, table creation, and password handling practices in a secure manner. Figure 2 highlights the syntax for better understanding, where a blue color is used to easily recognize the SQL keywords (e.g., create, if not exists, or primary key). Additionally, comments in both SQL and PHP, denoted by -- in SQL and // in PHP, are displayed in a gray color, to clearly distinguish them from the executable code. Moreover, specific values, including database names (my_database), table names (users), column names (user_id, username, etc.) and data types (VARCHAR, INT, etc.) are presented in black or purple colors, for clarity and differentiation within the code structure.

The code provided by ChatGPT in Figure 2 lacks certain security measures related to database management and password handling. It inserts a user with a plaintext password directly into the database, failing to employ proper encryption practices. While the code includes a comment on the hypothetical hashing and salting of the password, this is not reflected in the actual insertion of data. Furthermore, the absence of specific password complexity constraints, such as minimum length or the inclusion of special characters, is a notable security gap. The code also neglects the use of prepared statements, a recommended practice for preventing SQL injection attacks. Additionally, the handling of uniqueness constraints for the username is not explicitly addressed in the code. In a production environment, these security considerations are crucial for safeguarding data integrity and password security.

The issue lies in the fact that LLMs do not inherently generate secure code unless explicitly specified. This stems from the training data, predominantly consisting of programs created prior to considering potential security vulnerabilities such as SQL injections. The models tend to replicate the patterns present in the training data and, since secure coding practices were not comprehensively integrated at the time of training, LLMs may inadvertently generate code that lacks robust security measures. Consequently, the responsibility falls on developers to consciously incorporate secure coding practices and address potential vulnerabilities in the code generated by LLMs, to ensure a heightened level of security.

To obtain secure code that establishes a MySQL database, the user should have specified the following sentence at the prompt:

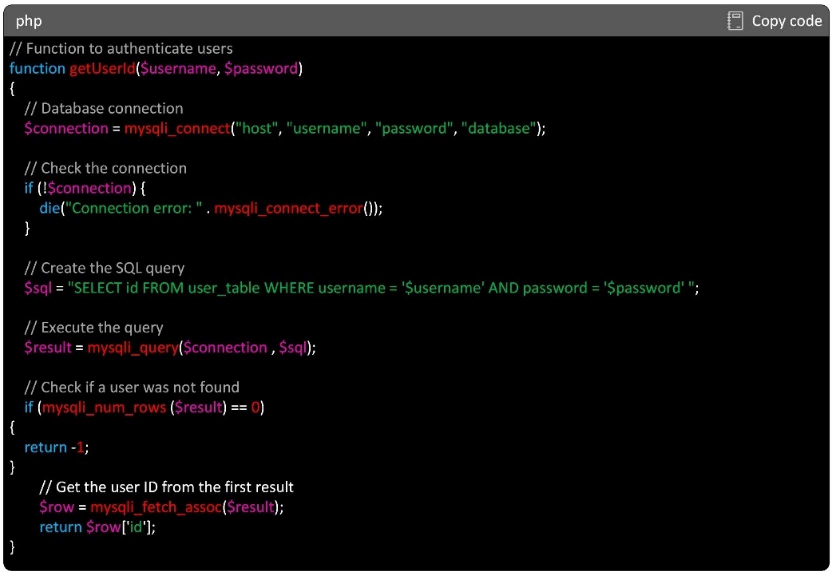
- *"Generate a secure script to set up a MySQL database named 'my_database' and create a table named 'user_table'. The user table should include fields for user ID, unique username, password hash, and salt. Additionally, ensure that the script utilizes secure password storage practices, such as hashing with a salt, and prevent SQL injection attacks by using prepared statements. Finally, include a sample insertion to add a user named 'sampleUser' with an encrypted password and a randomly generated salt"*.

The key difference between the two prompts lies in the focus on security. While the secure prompt emphasizes terms related to secure cryptography, prevention of SQL injection attacks, and safe password storage, the insecure prompt omits these critical considerations, focusing solely on the basic creation of a database and a user table. As a result, the new code will use secure hash functions to store passwords in the database, ensuring the protection of sensitive user credentials. Additionally, it will generate a random salt for each user, stored alongside the hashed password, enhancing the security measures against potential decryption attempts. Moreover, the code will use prepared statements to prevent SQL injection attacks, thus fortifying the database against malicious exploits. Furthermore, it will incorporate password complexity constraints—including minimum length and special character requirements—to bolster the resilience against brute-force attacks. Lastly, the code will explicitly manage the uniqueness of usernames, reducing the risk of unauthorized access and ensuring data integrity within the system.

3.3. Methodology for User Authentication

Figure 3 shows a representative example of a PHP code that defines a function named "getUserId", designed for user authentication in the SQL database. The PHP code was generated by ChatGPT based on the following prompt:

- *"Generate a PHP function to authenticate users using login credentials. The function should connect to a MySQL database and verify if the username and password match entries in the 'user_table' table. If successful, the function should return the corresponding user ID. Otherwise, return −1"*.



```php
// Function to authenticate users
function getUserId($username, $password)
{
  // Database connection
  $connection = mysqli_connect("host", "username", "password", "database");

  // Check the connection
  if (!$connection) {
    die("Connection error: " . mysqli_connect_error());
  }

  // Create the SQL query
  $sql = "SELECT id FROM user_table WHERE username = '$username' AND password = '$password' ";

  // Execute the query
  $result = mysqli_query($connection , $sql);

  // Check if a user was not found
  if (mysqli_num_rows ($result) == 0)
{
  return -1;
}
    // Get the user ID from the first result
    $row = mysqli_fetch_assoc($result);
    return $row['id'];
}
```

Figure 3. PHP code for authenticating users in a database, provided by ChatGPT.

The process begins with establishing a connection to the database using "mysqli_connect", where the placeholders 'host', 'username', 'password', and 'database' should be replaced with the actual configuration values. The connection success is verified and, if any errors occur, the program terminates with an error message. Subsequently, an SQL query is crafted to search for a user in the 'user_table', whose provided username and password match the parameters passed to the function. The query is executed using "mysqli_query". The function then checks if any user was found and, if not, it returns −1. If a user is found, it retrieves the user ID from the first result and returns it. In essence, the "getUserId" function serves to authenticate a user in the database, providing the user ID upon successful authentication, or −1 if no matching user is found.

The vulnerability in the PHP code provided by ChatGPT lies in its susceptibility to SQL injection attacks. The code constructs SQL queries by directly concatenating user-provided input, specifically the $username and $password variables, into the SQL statement, without proper sanitization or parameterization. This opens the door to malicious exploitation, as an attacker could manipulate input to inject arbitrary SQL code, potentially compromising the security and integrity of the database.

According to this, an attacker will attempt to manipulate the password field during the login process, by inputting the following syntax in the space to write the password during the login process of the web page (Figure 4).

```sql
DELETE FROM user_table WHERE ' OR 1 = 1 --
```

Figure 4. SQL code for manipulating the password field during the login process.

Then, the attacker will manipulate the SQL query by concatenating the username and password provided by the user directly into the query string, without proper sanitization. Here, the ' OR 1 = 1 syntax is added to the password field. This part is designed to make the condition always true and the -- symbol is used to comment out the rest of the query, disabling any additional conditions that may follow. The resulting query might look something like that shown in Figure 5.

```sql
SELECT id FROM user_table WHERE username = 'username' AND password = '' OR 1 = 1 --'
```

Figure 5. SQL code for selecting all records from the user table of the database.

Since 1 = 1 is always true, the AND password = '' condition becomes irrelevant. The query effectively reduces to selecting all records from the user_table and the -- symbol is used to comment out the rest of the query, to avoid syntax errors. This type of attack could lead to the deletion of records from the user_table, if the query were a DELETE type. Therefore, the attacker introduces the syntax "DELETE FROM user_table WHERE ' OR 1 = 1 --" in the space where the password must be written during the login process of the web page.

To obtain secure code that establishes a MySQL database, the user should have specified the following sentence at the prompt:

- *"Generate a secure PHP function to authenticate users using login credentials. The function should connect to a MySQL database and verify if the username and password match entries in the 'user_table' table. Ensure that the function is protected against SQL injection attacks by using prepared statements or parameterized queries. If the authentication is successful, the function should return the corresponding user ID. Otherwise, return −1"*.

The main difference between the two prompts lies in the security approach. While the insecure prompt does not specify the need for protection against SQL injection attacks and does not mention the use of prepared statements or parameterized queries, the secure prompt explicitly emphasizes the importance of these measures to mitigate such risks. As a result, in the modified version, direct concatenation of the $username and $password values into the SQL query will be replaced with placeholders (?). Subsequently, prepared statements will be employed to bind the $username and $password values to these placeholders prior to query execution. This ensures that user input values will be treated as data and not as part of the SQL query structure, thereby mitigating the risk of SQL injection vulnerability. It is a classic example of SQL injection, emphasizing the importance of using prepared or parameterized queries to mitigate these vulnerabilities. To enhance security, it is recommended to use prepared statements or parameterized queries, which effectively mitigate the risk of SQL injection, by separating user input from the SQL code and ensuring proper escaping of values.

4. Experimentation

This section describes a case study involving an SQL injection, detailing the steps taken by a hacker to exploit weaknesses in the construction and execution of SQL queries on a website, with the goal of removing user records from the organization database.

In industrial applications like inventory management, production tracking, and quality management systems, SQL injection attacks pose serious risks. For instance, in a web-based inventory management system, vulnerabilities in SQL queries could be exploited by injecting malicious code into input fields. This could allow attackers to manipulate queries, potentially deleting user records from the database. By bypassing authentication, attackers could gain unauthorized access to sensitive data, leading to disruptions in production processes, a loss of critical information, or compromised product quality, if not promptly addressed. Similarly, in self-configurable systems, SQL injection attacks represent a significant threat, where the dynamic adaptability introduced by agents creates new vulnerabilities. Automatic adjustments in the web server, PHP code, and database could be exploited by attackers to launch targeted attacks, thus interrupting the normal functioning of operations.

SQL Injection Attack

The self-generating vulnerability covered in this study consists of an SQL injection attack. This involves exploiting a security gap in the way SQL queries are constructed and handled on a website, specifically during the login process. The hacker will aim to delete the organization's user database and needs to ascertain the table name targeted for deletion. To achieve this, the attacker will seek vulnerabilities and, in this instance, exploitation occurs through the URL of the website.

Assuming the URL is http://www.domain.com/page.php?id=1&id2=1&id3=1 and the database produces an error when a single quote (′) is appended to id3, the manipulated URL becomes http://www.domain.com/page.php?id=1&id2=1&id3=1′. This process admits other variants than a single quote and involves iterative testing. After encountering an SQL 1064 error resembling "You have an error in your SQL syntax; check the manual that corresponds to your SQL server version for the right syntax", the hacker can infer a potential SQL injection vulnerability. The error message is essentially revealing to the attacker that the input provided interfered with the SQL query's structure, hinting at a potential breach for unauthorized access or manipulation of the database.

To be successful, the hacker must determine the number of columns used by the web page in the next step. To this end, the hacker engages in testing the column count, using the "order by" clause. For instance, the hacker manipulates the URL as http://www.domain.com/page.php?id=1&id2=1&id3=1′ order by 3 --+, where there is no error encountered. The segment ′ order by 3 --+ is an attempt to manipulate the SQL query executed by the backend script page.php, where the ′ character marks the end of a parameter value and by 3 seeks to sort the query results by the third column. The -- symbol initiates an SQL comment, ensuring that anything following it is disregarded by the database engine. The + symbol at the end is an effort to evade potential input filters that could be in place.

Subsequent attempts, such as http://www. domain.com/page.php?id=1&id2=1&id3=1′ order by 10 --+, will result in an error, specifically "The query did not execute: Unknown column '10' in 'order clause'". This iterative process allows the hacker to understand the structure of the table and to determine the exact number of columns used by the web page, enabling the attacker to proceed with more targeted and effective SQL injection attempts.

Later, the "union select" command will essentially grant the hacker the ability to execute statements directly in the database, with the results reflected on the webpage. Following the example, the hacker will manipulate the URL to http://www.domain.com/page.php?id=1&id2=1&id3=1′ union select 1,2,3,4,5,6,7,8,9,10. What this accomplishes is the presentation of column numbers to the hacker, indicating which columns are vulnerable for SQL injection. This step is crucial and requires attention to discern the changes. In some cases, alterations are quite noticeable and abrupt, while they are more subtle in others. As a representative example, imagine that the vulnerability is located in column 5. The webpage will reveal this number somewhere, providing the hacker with the information needed for subsequent injection exploits.

The hacker then needs to input "database()" in column 5, resulting in the URL appearing as http://www. domain.com/page.php?id=1&id2=1&id3=1′ union select 1,2,3,4,database(),6,7,8,9,10. This action leverages the SQL injection vulnerability previously identified in column 5. By inserting "database()" at this point, the hacker retrieves information about the current database being used by the webpage. Assuming the database in use is Oracle instead of MySQL, this detail could be determined by placing "@@datadir" where the hacker previously used "database()". This technique allows the hacker to gather critical details about the database structure, paving the way for a more sophisticated and targeted exploitation. The hacker, having successfully obtained user information, is then positioned to execute a specific SQL injection. Attackers have several ways to send HTTP requests in web applications, to carry out SQL injection attacks. In this case study, username and password information can be sent manually, using tools such as Postman or cURL, or by automating scripts in languages such as Visual Basic .NET. This would take on the form described in Figure 6.

```
Username: Anonymous
Password: DELETE FROM user_table WHERE ' OR 1 = 1
```

Figure 6. Visual Basic .NET code for deleting records from the user table of the database.

This malicious SQL injection attempts to delete records from the user table, where the condition ′ OR 1 = 1 is always true, effectively targeting all rows for deletion. This example underscores the severity of SQL injection vulnerabilities, emphasizing the importance of implementing robust security measures to safeguard against unauthorized access and the manipulation of sensitive data.

Figure 7 summarizes the SQL injection attack. The key actors in this scenario include the hacker, responsible for carrying out the attack; the website at the server, which serves as the destination; and the database, where sensitive information is stored. The attack phases include reconnaissance, during which the attacker gathers information about the website and database; crafting the malicious request, designed to exploit a vulnerability in the database; sending the malicious request to the website; exploiting the vulnerability within the database; and, finally, extracting the sensitive data from the website.

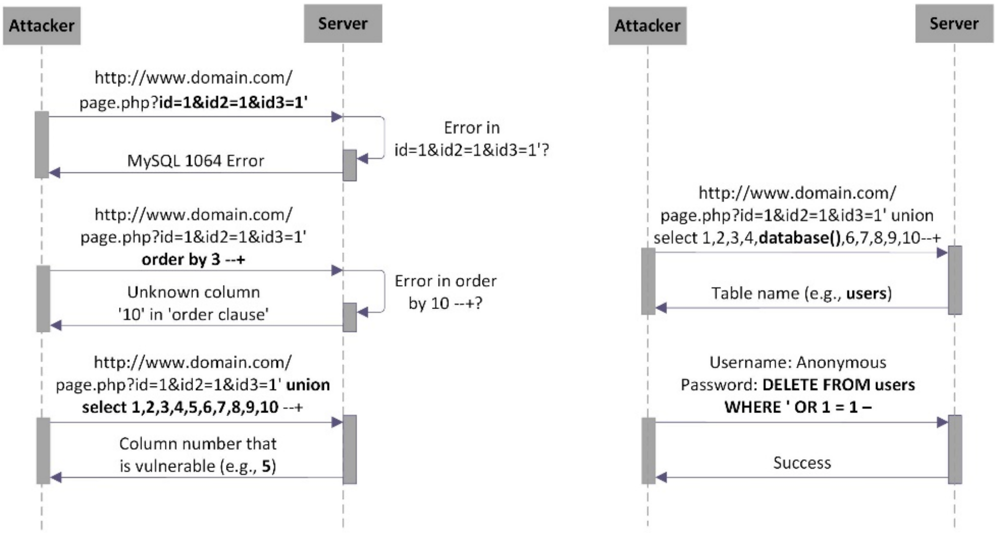


Figure 7. Overview of the SQL injection attack.

5. Discussion

This section discusses the limitations of traditional security practices in software development and introduces an AI-driven methodology for enhanced vulnerability detection. It compares the effectiveness of ChatGPT with other LLMs in identifying risks, emphasizing the importance of thorough evaluation in real-world applications. The section also covers regulations and legislation pertaining to industrial applications, emphasizing compliance with standards such as GDPR, HIPAA, NIST Cybersecurity Framework, and ISO/IEC 27001 [35], to ensure data security and mitigate legal and regulatory risks. It also highlights adherence to industry-specific standards like ISA/IEC 62443 [36] and provides guidelines for trustworthy AI development in industrial settings. Additionally, it addresses security enhancement measures in industrial applications, including continuous security monitoring, testing, and code review to identify and mitigate vulnerabilities. It emphasizes the importance of security awareness training for industrial developers.

5.1. Analysis of Traditional Security Methodologies Versus AI-Based Approaches

Traditional tools and methodologies for addressing security in software development often focus on static testing and static code analysis. These practices involve manual code review by developers, or the use of automated tools that scan code for known vulnerabilities. However, recent research found that 32% of web applications contained high-severity security vulnerabilities, even after static code analysis [37]. These data reflect the persistence of vulnerabilities in web applications, despite standard security practices. In contrast, the methodology based on AI code generation introduces an innovative and automated approach to vulnerability detection. Rather than relying solely on static testing, this methodology uses AI models to generate code, which can assist in identifying and mitigating vulnerabilities more efficiently and accurately [38]. This is because AI models can analyze large amounts of code and learn security patterns, to avoid the introduction of common vulnerabilities. This enables them to scale seamlessly to handle increasingly complex software systems.

A comparative analysis of the approach carried out with ChatGPT, with respect to other LLMs, can be seen in Table 2, where several indicators of performance can be found. The different LLMs exhibit varying degrees of proficiency in identifying risks within the generated code. In this sense, there are instances where LLMs detect fewer risks compared to ChatGPT, while they identify more in other scenarios. This difference highlights the risk in code generation tasks, indicating that the quality and security can significantly depend on the model utilized and the specific programming task. Moreover, the analysis indicates that ChatGPT and Copilot demonstrate robust scalability, as evidenced by their consistent performance in generating code and detecting risks, contributing to the overall security in different contexts. In contrast, Gemini exhibits limitations in certain scenarios, including the inability to generate code and lower efficacy in risk detection in specific tasks. Such findings underscore the importance of the thorough evaluation and comparison of codes in real-world applications, to ensure robust and secure software development practices.

For instance, numerous vulnerabilities have been identified within the script provided by ChatGPT in Figure 2. Pertaining to low-severity issues, shortcomings in error handling and input validation were observed, potentially leading to unexpected system behavior and susceptibility to SQL injection attacks. Moderately concerning are deficiencies in password management practices (e.g., the lack of a clear password policy and the usage of simplistic password examples), which could compromise the integrity of user credentials. However, of heightened concern are the absence of authentication mechanisms for database access, the potential use of insecure password hashing algorithms, and the exposure of sensitive user information through auto-incremented user IDs.

The revised code made with ChatGPT addresses various security concerns compared to the original version, but still requires improvements for optimal system security. Password storage is strengthened with hashing and salt, yet susceptibility to brute force attacks remains, if weak hashing algorithms or predictable passwords are used. Although the hashed password length is increased, adequacy depends on the hashing algorithm and security practices. Similarly, the salt field's length may need adjustment. Also, explicit error handling is absent, necessitating robust management for system resilience. Furthermore, the absence of a password policy and explicit input validation exposes vulnerabilities to potential attacks. Despite this, a 45% improvement in code security was obtained after a first review of the SQL script.

Similarly, the insecure version of the user authentication code generated with ChatGPT in Figure 3 also reveals differences, when compared with a secure version. While both versions share similar code structures, the secure rendition demonstrates a reduced risk profile, featuring one less low-risk issue compared to the insecure version (see Table 2).

Table 2. Performance of various LLMs, in terms of secure code generation.

Code	Description	Code Size	Risk Detected (by LLM Itself)	Risk Detected (by ChatGPT)
Insecure SQL	Database creation with ChatGPT	13 lines, 45 words	Total 10 (low: 3, medium: 4, high: 3	-
Secure SQL	Database creation with ChatGPT	17 lines, 56 words	Total 6 (low: 3, medium: 1, high: 2)	-
Insecure SQL	Database creation with Gemini	9 lines, 22 words	Total 3 (low: 0, medium: 2, high: 1)	Total 8 (low: 6, medium: 2, high: 0)
Secure SQL	Database creation with Gemini	17 lines, 39 words	Total 3 (low: 0, medium: 1, high: 2)	Total 5 (low: 1, medium: 3, high: 1)
Insecure SQL	Database creation with Copilot	9 lines, 38 words	Total 5 (low: 0, medium: 3, high: 2)	Total 5 (low: 2, medium: 2, high: 1)
Secure SQL	Database creation with Copilot	13 lines, 45 words	Total 5 (low: 1, medium: 3, high: 1)	Total 3 (low: 1, medium: 3, high: 1)
Insecure PHP	User Authentication with ChatGPT	35 lines, 115 words	Total 3 (low: 1, medium: 1, high: 1)	-
Secure PHP	User Authentication with ChatGPT	35 lines, 111 words	Total 2 (low: 0, medium: 1, high: 1)	-
Insecure PHP	User Authentication with Gemini	Unable to create	N/A	N/A

Secure PHP	User Authentication with Gemini	Unable to create	N/A	N/A
Insecure PHP	User Authentication with Copilot	45 lines, 129 words	Total 5 (low: 0, medium: 2, high: 3)	Total 2 (low: 0, medium: 1, high: 1)
Secure PHP	User Authentication with Copilot	31 lines, 106 words	Total 7 (low: 1, medium: 5, high: 1)	Total 2 (low: 1, medium: 1, high: 0)

5.2. Regulation and Legislation on Industrial Applications

Ensuring compliance with relevant industry regulations and standards, such as General Data Protection Regulation (GDPR) or the Health Insurance Portability and Accountability Act (HIPAA), can help protect sensitive data and mitigate the legal and regulatory risks associated with security breaches [39]. Additionally, compliance frameworks like NIST Cybersecurity Framework or ISO/IEC 27001 establish the requirements for implementing effective security controls and risk management practices related to SQL injection vulnerabilities. To fulfill it, ISO/IEC 27002 [40] offers a comprehensive framework for information security management systems, providing guidance on establishing, implementing, maintaining, and continually improving security controls and procedures within an organization. ISO/IEC 27002 emphasizes the necessity of robust security policies, organizational structures, and human resource management practices to effectively mitigate risks. For instance, ISO/IEC 27002 recommends implementing security policies addressing secure coding practices, including guidelines for parameterized queries and input validation to prevent SQL injection exploits. The organizational security measures outlined in ISO/IEC 27002 encompass role-based access controls, regular security training for developers and administrators, and incident response protocols to promptly address SQL injection breaches [41].

Additionally, compliance with industry-specific standards such as ISA/IEC 62443 for industrial control systems (ICSs) cybersecurity is crucial for ensuring the security of critical infrastructure and industrial operations against cyber threats. Adherence to this standard ensures that industrial control systems are designed, implemented, operated, and maintained securely. Certification bodies like the International Society of Automation (ISA) or the International Electrotechnical Commission (IEC) provide certification programs to validate compliance with ISA/IEC 62443, offering assurance to stakeholders regarding the security of industrial systems. Complementarily, RFC 2196 offers guidance on site security policies and procedures, focusing on establishing clear security guidelines and practices to protect against various security threats, including SQL injection attacks. RFC 2196 underscores proactive measures such as pre-employment background checks, ongoing security awareness training, and stringent access controls to mitigate the risks associated with SQL injection vulnerabilities [42].

Moreover, the use of AI-supported tools in the industry necessitates adherence to standards and best practices for ensuring the security and reliability of AI systems. Organizations like the Institute of Electrical and Electronics Engineers (IEEE) and the National Institute of Standards and Technology (NIST) provide guidelines and frameworks for trustworthy AI development, encompassing aspects such as transparency, accountability, fairness, and robustness [43]. Adhering to these guidelines helps to mitigate the risks associated with AI-generated code and ensures the integrity of industrial processes and systems.

Furthermore, collaboration with industry consortia and research organizations can facilitate the development and adoption of industry-specific security standards and best practices for AI-supported systems. Initiatives such as the Industrial Internet Consortium (IIC) or the Manufacturing Enterprise Solutions Association (MESA) promote collaboration among industry stakeholders to address cybersecurity challenges in industrial environments and foster innovation in AI applications. This is also the case for e-mark certification by TÜV SÜD, for the development of AI and machine learning in autonomous vehicles [44]. By actively participating in such initiatives, organizations can stay abreast of emerging threats and technologies, enhancing the security posture of AI-enabled industrial systems.

5.3. Measures for Enhancing Security in Industrial Applications

Implementing continuous security monitoring solutions designed to detect and respond to security threats in real-time for industrial applications can also help in mitigating the risks. Security information and event management (SIEM) systems, intrusion detection systems (IDSs), and endpoint detection and response (EDR) solutions are examples of technologies that allow for the detection of suspicious or malicious activity in the database in real-time [45]. Moreover, in industrial settings, destructive measures could involve actions taken on a non-production environment or a replicated database instance, without impacting the live production system. In this scenario, destructive measures could include deliberately exploiting SQL injection vulnerabilities to test the system's resilience, simulate potential attacks, or identify weaknesses in the security defenses, without risking harm to the live environment. The utilization of security tools such as OpenVAS, Nessus, Wireshark, Nmap, and SQLmap is essential for this purpose [46]. These tools aid in detecting potential vulnerabilities in network infrastructure, including misconfigurations exacerbating SQL injection risks. Specifically, SQLmap automates the detection and exploitation of SQL injection flaws within web applications, enabling security teams to assess defenses effectively and prioritize remediation efforts [47].

Regarding addressing the challenges posed by novice programming errors and vulnerabilities introduced by AI-generated code in industrial applications, it is essential to incorporate additional measures tailored to specific requirements, based on standards and industry best practices. Implementing secure coding standards and guidelines, such as those outlined by organizations like Open Web Application Security Project (OWASP) or Computer Emergency Response Team (CERT), can help mitigate common programming errors and vulnerabilities [46]. These practices include input validation, output encoding, proper error handling, and secure authentication mechanisms, which are particularly crucial in industrial settings, where data integrity and system reliability are paramount.

Establishing robust code review processes—where experienced developers review code written by novice programmers or generated by AI tools—can help identify and address potential security vulnerabilities early in the development lifecycle. Code reviews should focus not only on functionality, but also on security considerations specific to industrial applications, such as ensuring compliance with industry-specific regulations and standards.

Incorporating security testing into the software development lifecycle of industrial systems, including techniques such as static code analysis, dynamic application security testing (DAST), and penetration testing, can help identify and remediate vulnerabilities in both manually written and AI-generated code. Automated tools like Burp Suite, SonarQube, and Veracode can assist in these efforts, providing specialized features for assessing the security of industrial software systems [48].

Furthermore, providing ongoing security awareness and training programs for developers in industrial settings is essential. This includes education on secure coding practices focused on industrial applications (e.g., Cisco Certified Network Professional), common vulnerabilities specific to industrial environments, and threat modeling techniques adapted to the unique challenges faced by industrial systems. By enhancing the security knowledge and skills of developers, organizations can improve the overall security posture of their industrial systems and reduce the likelihood of introducing vulnerabilities through coding errors or AI-generated code [49].

6. Conclusions

Novice developers typically make 16 errors per 100 lines of code, while experienced ones reduce this rate by half, yet errors still present significant risks across industries. Large Language Models (LLMs) offer faster coding, but they do not always produce bug-free code, posing risks of application failure and mistrust if misused. LLMs, like ChatGPT, are popular among engineers, but concerns persist regarding security gaps. Consequently, new vulnerabilities arise as ICT evolves, necessitating scrutiny of self-generated tools. Moreover, novice programmers struggle with procedural programming, highlighting the need for strategies to address deficiencies. Therefore, analyzing security in AI-generated programming and enhancing security training in ICT education programs are crucial. Collaborative efforts between engineers and cybersecurity experts, facilitated by tools like DeepState and VulnEx, are essential for establishing robust security protocols and ensuring software security for industrial applications, through security code reviews.

Accordingly, the present research focuses on the vulnerabilities of LLMs with the aim of contributing to the understanding of the implications, through a practical case, and proposing effective strategies to enhance secure software development practices. The methodology carried out in this work describes the creation of a database, definition of tables, insertion of user data, and storage of passwords, typically used in industrial applications like inventory management systems, production tracking systems, and quality management systems, where web-based applications, databases, SQL queries, and PHP play a key role. However, the provided code is shown to lack strong security measures (e.g., password encryption and use of prepared statements to prevent SQL injection attacks). Furthermore, it is shown that PHP code for user authentication is vulnerable to SQL injection, due to concatenation of user input into SQL queries without proper sanitization. Thus, the importance of using prepared statements or parameterized queries to mitigate SQL injection vulnerabilities is emphasized.

The experimentation carried out showed an SQL injection attack as an example, which exploits vulnerabilities in the construction of SQL queries during login processes on a website. Through iterative URL manipulation tests, the attacker identifies potential vulnerabilities. By determining the number of columns and executing commands, the hacker gains access to the database information. The attacker then executes a malicious SQL injection to delete user records, which highlights the critical need for strong security measures to prevent unauthorized access and manipulation of data. This could lead to disruptions in production processes, loss of critical information, or compromised product quality.

The analysis on AI-based code generation models revealed significant variations between them in detecting security risks. A comparison between codes showed a trend towards reduced risk in secure versions, indicating the potential of LLMs to improve code security. However, it is essential to keep in mind that fewer problems detected using an LLM does not necessarily equate to better code quality; it could reflect a lower ability to identify problems. In this sense, ChatGPT generally detected more risks in the code compared to Gemini and Copilot, demonstrating better performance. These findings highlight the necessity for the comprehensive evaluation and comparison of AI-generated code in practical applications, to guarantee robust and secure software development practices.

Regarding recommendations on security, ensuring compliance with industry regulations such as GDPR and HIPAA helps protect sensitive data and mitigate legal risks. Frameworks such as NIST and ISO/IEC 27001 establish security controls to address SQL injection vulnerabilities. To this end, ISO/IEC 27002 emphasizes strong security policies and training to prevent breaches. In addition, compliance with industry-specific standards such as ISA/IEC 62443 for industrial control systems’ (ICSs) cybersecurity is crucial for ensuring the security of critical infrastructure and industrial operations. Additionally, other approaches such as continuous security monitoring with SIEM, IDS, and EDR systems help detect and respond to threats in real time. In addition, destructive testing in non-production environments can assess resiliency, without putting active systems at risk.

In this sense, tools like OpenVAS, Nessus, and SQLmap help in detecting and testing for vulnerabilities. Additionally, implementing secure coding standards, robust code review processes, and security testing in the development lifecycle are also noted as approaches to address novice programming errors and vulnerabilities in AI-generated code. Automated tools like Burp Suite and SonarQube assist in security testing efforts. In this regard, ongoing security awareness and training programs educate developers on secure coding practices to improve organizational security.

Regarding the limitations of this work, while the provided example illustrates potential vulnerabilities introduced by generative AI tools, a more thorough examination is needed, ideally with multiple scenarios beyond the current single example. Incorporating analyses that compare code generated by engineers and AI tools of varying expertise levels could provide statistical evidence. Furthermore, practical examples of mitigation strategies, such as modifying prompts to generate attack-resistant code, will be covered in future studies. Future work could focus on investigating additional approaches to address vulnerabilities in self-generated code, such as the development of static analysis tools specifically designed to detect and mitigate these types of security risks. Moreover, further exploration could be carried out into the ethical and social implications of the increasing reliance on AI in software development, as well as the legal and regulatory implications related to liability for self-generated code vulnerabilities.

Funding: This research received no external funding.

Data Availability Statement: The original contributions presented in the study are included in the article, further inquiries can be directed to the corresponding author.

Acknowledgments: I would like to express my very great appreciation to A. Fernández García for his valuable and constructive work that helped to improve this research.

Conflicts of Interest: The author declares no conflicts of interest.

References

1. BSA Foundation. Every Sector Is a Software Sector: Manufacturing. *Tech. Rep.* **2018**, 1–16. Available online: https://software.org/wp-content/uploads/Every_Sector_Software_Manufacturing.pdf (accessed on 11 April 2024).
2. Fictiv and Dimensional Research. 2021 State of Manufacturing Report. *eBook* **2021**. Available online: https://www.fictiv.com/ebooks/2021-state-of-manufacturing (accessed on 11 April 2024).
3. Federation of German Industries. *BDI Manifesto for Growth and Employment—Germany 2020*; BDI publication No. 412E; Bundesverband der Deutschen Industrie e.V: Cologne, Germany, 2020. ISSN 0407-8977.
4. Lemeš, S. The Role of Software Engineering in Industry 4.0. In Proceedings of the Conference on Basic technologies and models for implementation of Industry 4.0, Sarajevo, Bosnia and Herzegovina, 2023; Volume 209, pp. 57–88.
5. Wheeler, D.A.; Jones, D.A. An Empirical Study of Programming Errors. *IEEE Trans. Softw. Eng.* **1995**, *21*, 1269–1286.
6. Zimmermann, T.; Herzig, M.; Zeller, A. A Study of Programming Errors in Professional Software. *Empir. Softw. Eng.* **2020**, *25*, 49–88.
7. Morris, M.R.; Sohl-dickstein, J.; Fiedel, N.; Warkentin, T.; Dafoe, A.; Faust, A.; Farabet, C.; Legg, S. Levels of AGI: Operationalizing Progress on the Path to AGI. 2023. Available online: https://arxiv.org/pdf/2311.02462.pdf (accessed on 11 April 2024).
8. Himmelreich, J. Against "Democratizing AI". *AI Soc.* **2023**, *38*, 1333–1346. https://doi.org/10.1007/s00146-021-01357-z.
9. Spasić, A.J.; Janković, D.S. Using ChatGPT Standard Prompt Engineering Techniques in Lesson Preparation: Role, Instructions and Seed-Word Prompts. In Proceedings of the 2023 58th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST), Nis, Serbia, 29 June–1 July 2023; pp. 47–50.
10. Ressler, M. Automated inauthenticity. *AI Soc.* 2023, *epub ahead of print*. https://doi.org/10.1007/s00146-023-01795-x.
11. Nasr, M.; Carlini, N.; Hayase, J.; Jagielski, M.; Cooper, A.F.; Ippolito, D.; Choquette-Choo, C.A.; Wallace, E.; Tramèr, F.; Lee, K. Scalable Extraction of Training Data from (Production) Language Models. *arXiv* **2023**, arXiv:2311.17035v1.
12. Murray-Rust, D.; Lupetti, M.L.; Nicenboim, I.; van der Hoog, W. Grasping AI: Experiential exercises for designers. *AI Soc.* **2023**, *epub ahead of print*. https://doi.org/10.1007/s00146-023-01794-y.
13. Ke, Q.; Jing, X.; Woźniak, M.; Xu, S.; Liang, Y.; Zheng, J. APGVAE: Adaptive disentangled representation learning with the graph-based structure information. *Inf. Sci.* **2024**, *657*, 119903. https://doi.org/10.1016/j.ins.2023.119903.
14. Hannibal046. Awesome-LLM. 2023. Available online: https://github.com/Hannibal046/Awesome-LLM (accessed on 11 April 2024).
15. Zheng, J.; Woźniak, M. Design of Quick Search Method for Key Feature Images in Mobile Networks. *Mob. Networks Appl.* **2022**, *27*, 2524–2533. https://doi.org/10.1007/s11036-022-02077-4.

16. Lozhnikov, P.S.; Zhumazhanova, S.S. Potential Information Security Risks in The Implementation of AI—Based Systems. In Proceedings of the 2022 Dynamics of Systems, Mechanisms and Machines (Dynamics), Omsk, Russia, 7–12 November 2022; pp. 1–4.
17. Jøsang, A.; Ødegaard, M.; Oftedal, E. Cybersecurity Through Secure Software Development. In Proceedings of the 9th IFIP WG 11.8 World Conference, WISE 9, Hamburg, Germany, 26–28 May 2015.
18. Pillay, N.; Jugoo, V.R. An Analysis of the Errors Made by Novice Programmers in a First Course in Procedural Programming in Java. *Comput. Sci.* **2006**, 1–8.
19. Albrecht, E.; Grabowski, J. Sometimes It's Just Sloppiness—Studying Students' Programming Errors and Misconceptions. In Proceedings of the 51st ACM Technical Symposium on Computer Science Education, Portland, OR, USA, 11–14 March 2020.
20. Xie, J.; Lipford, H.R.; Chu, B. Why do programmers make security errors? In Proceedings of the 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Pittsburgh, PA, USA, 18–22 September 2011; pp. 161–164.
21. Smith, J.; Johnson, B.; Murphy-Hill, E.; Chu, B.; Lipford, H.R. How Developers Diagnose Potential Security Vulnerabilities with a Static Analysis Tool. *IEEE Trans. Softw. Eng.* **2018**, *45*, 877–897. https://doi.org/10.1109/tse.2018.2810116.
22. Smith, J.; Johnson, B.; Murphy-Hill, E.; Chu, B.; Lipford, H.R. Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, Bergamo, Italy, 30 August–4 September 2015.
23. Srinivasu, P.N.; Ijaz, M.F.; Woźniak, M. XAI-driven model for crop recommender system for use in precision agriculture. *Comput. Intell.* **2024**, *40*, 1. https://doi.org/10.1111/coin.12629.
24. Kopec, D.; Yarmish, G.; Cheung, P. A description and study of intermediate student programmer errors. *ACM SIGCSE Bull.* **2007**, *39*, 146–156. https://doi.org/10.1145/1272848.1272899.
25. Votipka, D.; Fulton, K.R.; Parker, J.; Hou, M.; Mazurek, M.L.; Hicks, M.W. Understanding security mistakes developers make: Qualitative analysis from Build It, Break It, Fix It. In Proceedings of the 29th USENIX Security Symposium 2020, in Boston, MA, USA, 12–14 August 2020.
26. Goodman, P.; Grieco, G.; Groce, A. Tutorial: DeepState: Bringing Vulnerability Detection Tools into the Development Cycle. In Proceedings of the 2018 IEEE Cybersecurity Development (SecDev), Cambridge, MA, USA, 30 September–2 October 2018; pp. 130–131.
27. Dennig, F.L.; Cakmak, E.; Plate, H.; Keim, D.A. VulnEx: Exploring Open-Source Software Vulnerabilities in Large Development Organizations to Understand Risk Exposure. In Proceedings of the 2021 IEEE Symposium on Visualization for Cyber Security (VizSec), New Orleans, LA, USA, 27 October 2021; pp. 79–83.
28. Bharadwaj, R.; Parker, I. Double-edged sword of LLMs: Mitigating security risks of AI-generated code. In *Disruptive Technologies in Information Sciences VII*; SPIE: Bellingham, WA, USA, 2023; Volume 12542.
29. Cotroneo, D.; Improta, C.; Liguori, P.; Natella, R. Vulnerabilities in AI Code Generators: Exploring Targeted Data Poisoning Attacks. *ArXiv* **2023**, arXiv:2308.04451.
30. Erdogan, G.; Garcia-Ceja, E.; Hugo, Å.; Nguyen, P.H.; Sen, S. A Systematic Mapping Study on Approaches for AI-Supported Security Risk Assessment. In Proceedings of the 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC), Madrid, Spain, 12–16 July 2021; pp. 755–760.
31. Assal, H.; Wilson, J.; Chiasson, S.; Biddle, R. Collaborative Security Code-Review Towards Aiding Developers Ensure Software-Security. 2015. Available online: https://chorus.scs.carleton.ca/wp-content/papercite-data/pdf/assal2015codereview-soups.pdf (accessed on 11 April 2024).
32. Ludi, S. Work in Progress: Effectiveness of Collaboration within a Secure Software Engineering Course for SE and Computing Students. Proceedings. Frontiers in Education. In Proceedings of the 36th Annual Conference, San Diego, CA, USA, 27–31 October 2006; pp. 15–16.
33. Rehman, H.U.; Chaplin, J.C.; Zarzycki, L.; Ratchev, S. A Framework for Self-configuration in Manufacturing Production Systems. *IFIP Adv. Inf. Commun. Technol.* **2021**, *626*, 71–79.
34. Davis, M.E.; Phillips, J.A. *Learning PHP & MySQL*, 2nd ed.; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2007.
35. *ISO/IEC 27001:2022*; Information security, cybersecurity and privacy protection. Technical Report. International Organization for Standardization: Geneva, Switzerland, 2022.
36. *ANSI/ISA-62443-4-2-2018*; Security for industrial automation and control systems, Part 4-2: Technical security require-ments for IACS components, 2nd Printing. Technical Report. International Society of Automation: Research Triangle Park, NC, USA, 2018.
37. OWASP. Los Diez Riesgos Más Críticos en Aplicaciones Web. *Tech. Rep.* **2017**. Available online: https://wiki.owasp.org/images/5/5e/OWASP-Top-10-2017-es.pdf (accessed on 11 April 2024).
38. Louati, A.; Gasiba, T. Source Code Vulnerability Detection Using Deep Learning Algorithms for Industrial Applications. *Commun. Comput. Inf. Sci.* **2023**, *1768*, 161–178.
39. Blue, J.; Furey, E. A Novel Approach for Protecting Legacy Authentication Databases in Consideration of GDPR. In Proceedings of the 2018 International Symposium on Networks, Computers and Communications (ISNCC), Rome, Italy, 19–21 June 2018; pp. 1–6.
40. *ISO/IEC 27002:2022*; Information security, cybersecurity and privacy protection. International Organization for Standardization: Geneva, Switzerland, 2022.

41. Iqbal, A.; Horie, D.; Goto, Y.; Cheng, J. A Database System for Effective Utilization of ISO/IEC 27002. In Proceedings of the Fourth International Conference on Frontier of Computer Science and Technology, Washington, DC, USA, 17–19 December 2009; pp. 607–612.
42. Fraser, B. Request for Comments: 2196. Network Working Group 1997. Available online: https://datatracker.ietf.org/doc/html/rfc2196 (accessed on 11 April 2024).
43. Brundage, M.; Avin, S.; Wang, J.; Belfield, H.; Krueger, G.; Hadfield, G.; Khlaaf, H.; Yang, J.; Toner, H.; Fong, R.; et al. Toward Trustworthy AI Development: Mechanisms for Supporting Verifiable Claims. *arXiv* **2020**, arXiv:2004.07213.
44. Mateo Sanguino, T.J.; Lozano Domínguez, J.M.; Carvalho Baptista, P. Cybersecurity Certification & Auditing of Automotive Industry. In *Advances in Transport Policy and Planning*; Elsevier: Amsterdam, The Netherlands, 2020; Volume 5, pp. 1–17.
45. Rahmawati, T.; Shiddiq, R.W.; Sumpena, M.R.; Setiawan, S.; Karna, N.; Hertiana, S.N. Web Application Firewall Using Proxy and Security Information and Event Management (SIEM) for OWASP Cyber Attack Detection. In Proceedings of the 2023 IEEE International Conference on Internet of Things and Intelligence Systems (IoTaIS), Bali, Indonesia, 28–30 November 2023; pp. 280–285.
46. Vimala, K.; Fugkeaw, S. VAPE-BRIDGE: Bridging OpenVAS Results for Automating Metasploit Framework. In Proceedings of the 2022 14th International Conference on Knowledge and Smart Technology (KST), Chon Buri, Thailand, 26–29 January 2022; pp. 69–74.
47. Bouafia, R.; Benbrahim, H.; Amine, A. Automatic Protection of Web Applications Against SQL Injections: An Approach Based On Acunetix, Burp Suite and SQLMAP. In Proceedings of the 2023 9th International Conference on Optimization and Applications (ICOA), Abu Dhabi, United Arab Emirates, 5–6 October 2023; pp. 1–6.
48. Sönmez, F.Ö.; Kiliç, B.G. Holistic Web Application Security Visualization for Multi-Project and Multi-Phase Dynamic Application Security Test Results. *IEEE Access* **2021**, *9*, 25858–25884. https://doi.org/10.1109/access.2021.3057044.
49. Pikulin, V.; Kubo, D.; Bandara, S.; Shamsiemon, M.A.; Yasmin, A.; Jayatilaka, A.; Madugalla, A.; Kanij, T. Towards Developer-Centered Secure Coding Training. In Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), Luxembourg, 11–15 September 2023; pp. 24–31.