

Analysing the Security Risks of Integrating AI models in DevSecOps processes

Yoon Sae Young¹, Loh Wan Jing²,

¹Anglo-Chinese School (Independent), 121 Dover Rd, Singapore 139650,

²Defence Science and Technology Agency, 1 Depot Rd, Singapore 109679

1. Abstract

In the realm of AI-driven code generation tools, GitHub Copilot emerges as a reputable "AI pair programmer" for taking 55% less time on average to finish coding projects¹, integrated into various DevSecOps processes. This investigation addresses the inherent security implications tied to Copilot's code contributions, given its exposure to extensive unverified code from open-source GitHub repositories. A systematic exploration of the prevalence and circumstances leading to insecure code recommendations is undertaken, focusing on diverse scenarios associated with MITRE's "Top 25" Common Weakness Enumeration (CWE) list.

Our analysis delves into Copilot's efficacy along three dimensions—varieties in vulnerabilities, security hotspots and mitigation effectiveness—wherein 6 distinct project repositories were analyzed. Remarkably, these findings expose several security issues associated with the generated code. This study underscores the critical need for a thorough examination of security issues arising from Copilot's AI-driven code generation, particularly in real-world contexts. The study advocates for heightened security awareness among practitioners, emphasizing the significance of comprehensive security checks before incorporating suggested code. This underscores the pivotal role developers play in ensuring both the integrity and security of the codebase.

2. Math formulas

1. Mitigation Effectiveness of tool:

- Formula: $ME = \frac{M_n - N_n}{I_n} \times 100\%$

- Where:

ME = Mitigation Effectiveness

M_n = Number of Mitigated Security Issues of high severity

N_n = Number of New Security Issues of high severity

I_n = Number of Initially Identified Security Issues of high severity

- This metric is an accurate representation of the extent of security issues mitigated within code as it focuses on high severity issues, which have greater implications on the overall security of the code as compared to those of medium and low severity

3. Introduction

The evolution of software development methodologies has been significantly influenced by recent advancements in artificial intelligence (AI). GitHub Copilot, released in 2021, distinguished as an innovative "AI pair programmer," generates code in a variety of languages given some context such as comments, function names, and surrounding code. Trained on an extensive dataset of open-source code from GitHub repositories, Copilot exhibits a unique ability to offer contextually relevant code suggestions in real-time, redefining conventional approaches to software development.

The integration of GitHub Copilot is of particular significance in the context of DevSecOps, a methodology that integrates security practices into the software development lifecycle. As organizations increasingly embrace

¹ GitHub. (2022, September 7). Research: Quantifying GitHub Copilot's Impact on Developer Productivity and Happiness. Retrieved from <https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/>

DevSecOps principles to enhance the security posture of their applications, the incorporation of Copilot holds potential implications for both the efficiency of development processes and the consideration of security measures.

This report endeavors to explore the extent to which Copilot would recommend vulnerable or insecure code, giving developers a good idea of how much discretion and security protocols they should adhere to while using the AI model. By analyzing both the security and efficacy factors of code generated by the new AI programmer as compared to traditional manual fixing of code through research, seek to provide insights into its impact on code quality, security practices, and the evolving role of developers in the era of AI-assisted coding.

4. Literature review

4.1 GitHub Copilot's AI

GitHub Copilot is an AI-driven code generation tool that generates code in various programming languages. However, there are inherent security risks associated with Copilot's code contributions, given its exposure to extensive unverified code from open-source GitHub repositories. Thus, we quantify insecure code recommendations through metrics such as vulnerabilities and security hotspots, while also considering MITRE's "Top 25" Common Weakness Enumeration (CWE) list.

According to a study², approximately 40% of the code generated by Copilot had security vulnerabilities. The security weaknesses are diverse and related to 42 different CWEs, in which CWE-78: OS Command Injection, CWE-330: Use of Insufficiently Random Values, and CWE-703: Improper Check or Handling of Exceptional Conditions occurred the most frequently. Furthermore, another study³ analyzing 435 code snippets generated by Copilot from GitHub projects found that 35.8% of the code snippets exhibited security weaknesses, with a number of them being attributed to CWEs. The identified CWEs underscore the critical need for an examination of security issues arising from Copilot's AI-driven code generation, especially when it comes to recommending security fixes.

Therefore, this study seeks to challenge and reevaluate the conclusions drawn from these previous studies via utilizing Copilot to directly address security issues identified through security scanning, wherein valuable data about the AI model's depth to resolve security issues completely can be gathered. Overall, this study advocates for heightened security awareness among practitioners, emphasizing the significance of comprehensive security checks before incorporating suggested code. Therefore, developers should exercise caution when assimilating Copilot-generated code and ensure both the integrity and security of the codebase.

4.2 Evaluating Code Security

Numerous factors influence code quality. The literature on code generation often highlights the importance of ensuring that code functions correctly, which is typically assessed by compiling it and checking against unit tests. However, unlike evaluating functional correctness, determining the security of Copilot's recommendations requires various tools and techniques for conducting security analyses on software. Among these, source code analysis tools, particularly static application security testing tools, play a significant role. These tools are designed to examine source code or compiled versions for security flaws, specializing in identifying specific vulnerability classes.

In this report, we will mainly utilize automated analyses facilitated by GitHub and SonarCloud, both of which help form a critical foundation for a comprehensive security assessment.

² Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022, May). Asleep at the keyboard? assessing the security of github copilot's code contributions.

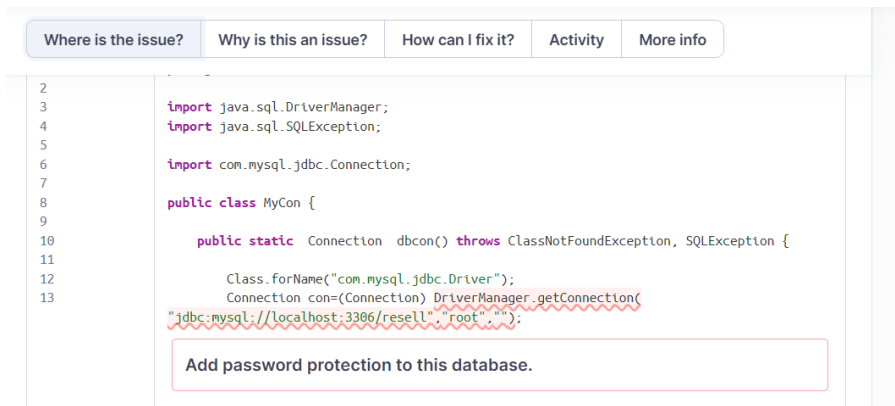
³ Fu, Y., Liang, P., Tahir, A., Li, Z., Shahin, M., & Yu, J. (2023). Security Weaknesses of Copilot Generated Code in GitHub. arXiv preprint arXiv:2310.02059.

Our approach involves utilizing SonarCloud's static code analysis that provides detailed metrics, statistics, graphs and categorization of security issues especially Common Weakness Enumeration (CWE) instances for us to compare the security of manual code fixing as compared to implementing security fixes recommended by Copilot.

4.3 Security metrics

4.3.1 Vulnerabilities

Vulnerabilities are defined as sections of code that can be exploited by hackers, with all of them resulting in one or more CWEs. Below is an example of a vulnerability where an authentication password is not set for the database, thus resulting in CWE 521: Weak password requirements, wherein an attacker could exploit this weakness once connected to access sensitive data and perform malicious modifications or deletions.



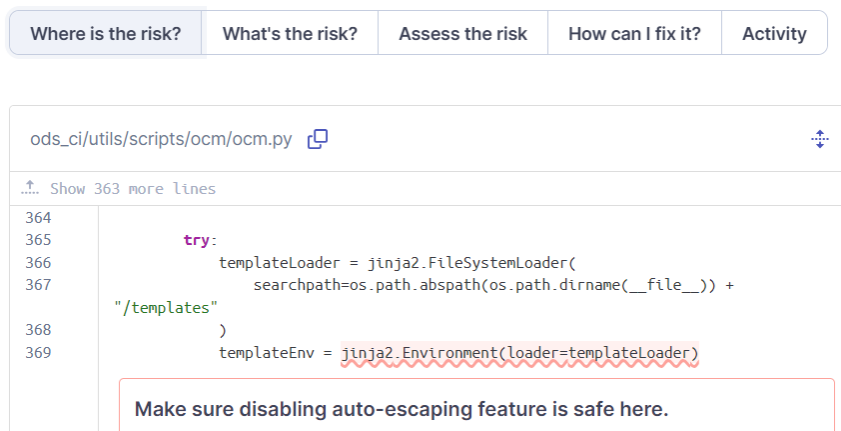
Where is the issue? Why is this an issue? How can I fix it? Activity More info

```
2
3 import java.sql.DriverManager;
4 import java.sql.SQLException;
5
6 import com.mysql.jdbc.Connection;
7
8 public class MyCon {
9
10     public static Connection dbcon() throws ClassNotFoundException, SQLException {
11
12         Class.forName("com.mysql.jdbc.Driver");
13         Connection con=(Connection) DriverManager.getConnection(
14             "jdbc:mysql://localhost:3306/resell","root","");
15     }
16 }
```

Add password protection to this database.

4.3.2 Security Hotspots

Security Hotspots are defined as sections of security-sensitive code that require manual review to assess whether a vulnerability exists. These sections of code may not necessarily have CWEs, however the programmer must consider the context and content of the code and its exploitability, thereafter he must make the decision to either mark it as safe or alter the code to ensure that there are no potential risks, of which the latter is preferred as it eliminates any potential attacks and is more maintainable in general. Below is an example of a security hotspot regarding Cross-Site Scripting (XSS) where the new Jinja2 environment has autoescape disabled by default, hence the code section may contain CWE-79: Improper Neutralization of Input During Web Page Generation, wherein hackers may use cross-scripting attacks on browsers to execute malicious scripts, which may result in exposure of sensitive data.



Where is the risk? What's the risk? Assess the risk How can I fix it? Activity

```
ods_ci/utlils/scripts/ocm/ocm.py
... Show 363 more lines
364
365     try:
366         templateLoader = jinja2.FileSystemLoader(
367             searchpath=os.path.abspath(os.path.dirname(__file__)) +
368             "/templates"
369         )
370         templateEnv = jinja2.Environment(loader=templateLoader)
```

Make sure disabling auto-escaping feature is safe here.

4.3.3 Common Weakness Enumerations (CWEs)

The CWE is a category system for software weaknesses and vulnerabilities. Outside of vulnerabilities, CWEs may also occur in bugs and code smells wherein the reliability and maintainability of the code

may be impacted, which may lead to security issues if not addressed accordingly. Below is an example of a bug where the resource is not closed properly, which may lead to resource leaks, thus compromising code security.



```
32     String end=request.getParameter("des");
33
34     try {
35
36
37         Class.forName("com.mysql.jdbc.Driver");
38
39         ServletContext ct=getServletContext();
40
41         Connection con=(Connection) DriverManager.getConnection(ct.getInitParameter(
"path"),ct.getInitParameter("user"),ct.getInitParameter("pass"));

```

Use try-with-resources or close this "Connection" in a "finally" clause.

4.4 Hypothesis

Our hypothesis is that Copilot when given security-tailored prompts with specific instructions will be able to perform on par with most human programmers when it comes to resolving security issues, however we also hypothesize that Copilot may end up trading security issues for one another when resolving them as it may prioritize functional correctness over other potential security risks, thus potentially introducing new security issues in the code commit.

5 Methodology

After thorough searching on GitHub for readily available open-source projects with enough security issues to conduct a scan, 6 projects that were found suitable to conduct targeted security analysis were selected. These projects were mainly web interfaces and software projects.

1. Project Configuration:

- Established version control using GitHub, subsequently cloning the repositories thrice to VSCode so that first may be used as control, the second may be used to test for manual security fixes and the last for Copilot security fixes.
- Configured Visual Studio Code (VSCode) for the chosen repositories.

2. SonarCloud Integration:

- Integrated chosen repositories with SonarCloud
- Configured SonarCloud to perform static code analysis with a focus on identifying and assessing security issues in the code.
- Identified security issues within repositories using SonarCloud

3. First wave - Manual Inspection and Remediation:

- Manually fixed code with security issues as identified by SonarCloud's security analysis in VSCode with non-compliant and compliant code suggestions examples by SonarCloud in addition to research.
- Scanned the Project after committing the changes to compare the difference between the new commit and the main branch and within the main branch

4. Second wave - GitHub Copilot Integration:

- Integrated GitHub Copilot and Copilot Chatbot into VSCode for code generation and recommended security fixes for the cloned repository
- Utilized security tailored prompts corresponding to the type of vulnerability identified by SonarCloud in

Copilot Chatbot along with command “/fix” to implement security fixes to code

- Scanned the Project after committing the changes to compare the difference between the new commit and the main branch and within the main branch

5. Comparative Analysis Framework:

- Conducted a comparative analysis of security fixes implemented manually and those suggested by GitHub Copilot using data imported from SonarCloud presented in tables and discussion
- Utilize SonarCloud’s branch analysis to analyze security issues that were introduced when using Copilot to inspect case examples in greater detail

6 Results

Results were mainly compiled from data derived from SonarCloud security analysis; with table 6.1 covering the security comparison based on the number of vulnerabilities detected; table 6.2 covering the security comparison based on the number of security hotspots detected; table 6.3 covering the security comparison based on the number of CWEs; and with table 6.4 evaluating the Mitigation Effectiveness (ME) of both fixing method

Table. 6.1. Comparison of number of vulnerabilities per 1000 lines of code (V_n), and percentage change across 6 projects before and after both methods of fixing

V_n (2 d.p.)	Project 1	Project 2	Project 3	Project 4	Project 5	Project 6
Initial V_n	1.67	0.83	0.78	13.89	1.36	10.44
V_n after manual fix	0.00	0.00	0.00	0.00	0.00	0.00
% Change	- 100%	- 100%	- 100%	- 100%	- 100%	- 100%
V_n after Copilot fix	0.00	0.27	0.00	0.00	0.43	0.00
% Change	- 100%	- 67.5%	- 100%	- 100%	- 68%	- 100%

Table. 6.2. Comparison of number of security hotspots per 1000 lines of code (SH_n), and percentage change across the 6 projects before and after both methods of fixing

SH_n (2 d.p.)	Project 1	Project 2	Project 3	Project 4	Project 5	Project 6
Initial SH_n	1.33	2.78	1.56	18.52	0.91	20.89
SH_n after manual fix	0.00	0.27	0.00	0.00	0.00	2.61
% Change	- 100%	- 90%	- 100%	- 100%	- 100%	- 88%
SH_n after Copilot fix	0.00	1.08	0.00	0.00	0.43	7.83
% Change	- 100%	- 61%	- 100%	- 100%	- 53%	- 63%

Table. 6.3. Comparison of number of CWEs per 1000 lines of code (CWE_n), and percentage change across the 6 projects before and after both methods of fixing

CWE_n (2 d.p.)	Project 1	Project 2	Project 3	Project 4	Project 5	Project 6
Initial CWE_n	1.67	2.5	2.47	32.41	1.82	20.89
CWE_n after manual fix	0.00	0.00	0.00	0.00	0.00	0.00
% Change	- 100%	- 100%	- 100%	- 100%	- 100%	- 100%
CWE_n after Copilot fix	0.00	0.28	0.39	0.00	0.43	0.00
% Change	- 100%	- 89%	- 84%	- 100%	- 76%	- 100%

Table. 6.4. Comparison of Mitigation Effectiveness (ME) for both methods across the 6 projects

ME (3 s.f.)	Project 1	Project 2	Project 3	Project 4	Project 5	Project 6
Manual fixing ME	100%	100%	100%	100%	100%	100%
Copilot fixing ME	100%	67%	100%	100%	75%	100%

7 Discussion

7.1 Analysis

Firstly, on the basis of vulnerabilities with reference to table 6.1, manual code fixing outperforms Copilot code recommendations by a significant margin, with manual code fixes resolving 100% of vulnerabilities across all 6 projects; while Copilot code recommendations resolved 100% of vulnerabilities across only 4 projects, with it resolving 67.5% and 68% of vulnerabilities across projects 2 and 5 respectively.

Secondly, on the basis of security hotspots with reference to table 6.2, manual code fixing outperforms Copilot code recommendations by a significant margin again, albeit with comparatively lower values than that of vulnerabilities, with manual code fixes resolving 100% of security hotspots across 4 projects, with it resolving 90% of that of Project 2's security hotspots and 88% of that of Project 6's; while Copilot code recommendations resolved 100% of security hotspots across only 3 projects, with it resolving 61%, 53% and 63% of security hotspots in Projects 2,5 and 6 respectively.

Next, on the basis of CWEs with reference to table 6.3, manual code fixing still outperforms Copilot code recommendations by a significant margin, albeit with comparatively lower values than that of vulnerabilities, with manual code fixes resolving 100% of security hotspots across all 6 projects; while Copilot code recommendations resolved 100% of security hotspots across only 3 projects, with it resolving 89%, 84% and 76% of security hotspots in Projects 2,3 and 5 respectively. The reason as to why the CWE values do not seem to align with those of the vulnerabilities and security hotspots can be accounted to the fact that some CWEs are

neither vulnerabilities nor security hotspots but instead could be bugs or code smells that may potentially be exploited by malicious hackers.

Finally, on the basis of Mitigation Effectiveness (ME) with reference to table 6.4, it is clear that manual code fixing is more effective than Copilot-recommended fixing in mitigating security issues, with manual code fixing mitigating **100%** of all high severity security issues across all projects as compared to Copilot-recommended fixing's average ME of **90.3%** .

7.2 Examples and Explanation

Projects 2 and 5, which notably encountered a greater margin of error when applying Copilot-recommended security fixes, will now have their specific case examples of security issues of **high severity** seemingly not being resolved analyzed with greater detail.

Case Example 1 – Project 2:

a. Original Code with CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

```
49         command.CommandText = sql;
50         _context.Database.OpenConnection();
51
52         using var dataReader = command.ExecuteReader();
53         dataReader.Read();
54         order.OrderId = Convert.ToInt32(dataReader[0]);
55     }
56
57     sql = ";\\nINSERT INTO OrderDetails (" +
58         "OrderId, ProductId, UnitPrice, Quantity, Discount" +
59         ") VALUES (@OrderId, @ProductId, @UnitPrice, @Quantity, @Discount)";
60     foreach (var (orderDetails, i) in order.OrderDetails.WithIndex())
61     {
```

b. Change Copilot Implemented:

```
49         command.CommandText = sql;
50         _context.Database.OpenConnection();
51
52         using var dataReader = command.ExecuteReader();
53         dataReader.Read();
54         order.OrderId = Convert.ToInt32(dataReader[0]);
55     }
56
57     sql = "INSERT INTO OrderDetails (" +
58         "OrderId, ProductId, UnitPrice, Quantity, Discount" +
59         ") VALUES (@OrderId, @ProductId, @UnitPrice, @Quantity, @Discount)";
60     command.CommandText = sql;
61     _context.Database.OpenConnection();
62
63     using var dataReader = command.ExecuteReader();
64     dataReader.Read();
65     order.OrderId = Convert.ToInt32(dataReader[0]);
66 }
67
68 sql = ";\\nINSERT INTO OrderDetails (" +
69     "OrderId, ProductId, UnitPrice, Quantity, Discount" +
70     ") VALUES (@OrderId, @ProductId, @UnitPrice, @Quantity, @Discount)";
71 foreach (var (orderDetails, i) in order.OrderDetails.WithIndex())
72 {
```

c. Copilot Issue:

Copilot's change involves removing the direct construction of SQL queries from user-controlled data, which is a positive security improvement. However, Copilot's subsequent modification reintroduces a potential SQL injection vulnerability by not using parameterized queries for the **'INSERT INTO OrderDetails'** operation. Concatenating user-controlled data directly into SQL queries opens up the possibility of SQL injection attacks,

which enable attackers to manipulate and delete data.

d. Secure Alternative and Explanation:

```
43     command.CommandText = sql;
44     command.Parameters.AddWithValue("@CustomerId", order.CustomerId);
45     command.Parameters.AddWithValue("@EmployeeId", order.EmployeeId);
46     command.Parameters.AddWithValue("@OrderDate", order.OrderDate);
47     command.Parameters.AddWithValue("@RequiredDate", order.RequiredDate);
48     command.Parameters.AddWithValue("@ShipVia", order.ShipVia);
49     command.Parameters.AddWithValue("@Freight", order.Freight);
50     command.Parameters.AddWithValue("@ShipName", order.ShipName);
51     command.Parameters.AddWithValue("@ShipAddress", order.ShipAddress);
52     command.Parameters.AddWithValue("@ShipCity", order.ShipCity);
53     command.Parameters.AddWithValue("@ShipRegion", order.ShipRegion);
54     command.Parameters.AddWithValue("@ShipPostalCode", order.ShipPostalCode);
55     command.Parameters.AddWithValue("@ShipCountry", order.ShipCountry);
56
57     _context.Database.OpenConnection();
58
59     using var dataReader = command.ExecuteReader();
60     dataReader.Read();
61     order.OrderId = Convert.ToInt32(dataReader[0]);
62 }
63
64 sql = "INSERT INTO OrderDetails (" +
65     "OrderId, ProductId, UnitPrice, Quantity, Discount" +
66     ") VALUES (@OrderId, @ProductId, @UnitPrice, @Quantity, @Discount)";
67 foreach (var orderDetails, i) in order.OrderDetails.WithIndex()
68 {
69     orderDetails.OrderId = order.OrderId;
70     sql += (i > 0 ? ", " : "") +
71         $"(@OrderId, @ProductId{i}, @UnitPrice{i}, @Quantity{i}, @Discount{i})";
72
73     command.CommandText = sql;
74     command.Parameters.AddWithValue($"@ProductId{i}", orderDetails.ProductId);
75     command.Parameters.AddWithValue($"@UnitPrice{i}", orderDetails.UnitPrice);
76     command.Parameters.AddWithValue($"@Quantity{i}", orderDetails.Quantity);
77     command.Parameters.AddWithValue($"@Discount{i}", orderDetails.Discount);
78 }
79
80 // Execute the final SQL statement with all the parameters
81 using (var command = _context.Database.GetDbConnection().CreateCommand())
82 {
```

This alternative replaces direct concatenations of user-controlled data in SQL queries with placeholders and provides the values through parameters, thus by treating user input as data and not executable code, parameterized queries prevent attackers from injecting malicious SQL code.

e. Reason for Copilot's Choice:

Copilot's initial change to avoid constructing SQL queries directly from user-controlled data aligns with secure coding practices. However, its subsequent modification overlooked the need for parameterized queries, reintroducing a vulnerability that the initial change sought to address. This highlights the importance of considering the entire context and ensuring comprehensive security measures in code changes.

Case Example 2 – Project 5:

a. Original Code with CWE-798: Use of Hard-coded Credentials

```
{
  "type": "service_account",
  "project_id": "gymworkddisclosedoauth",
  "private_key_id": "b7e94828d2b93de6332d8b47aec03459faf10efe",
  "private_key": "-----BEGIN PRIVATE KEY-----\nMIIIEvwIBADANBgkqhkiG9w0BAQFEASCBKkwggSlAgEAAoIBAQDVVcOmOIztk+vy\n+x0RyC5KncUOKEhWOC\n",
  "client_email": "firebase-adminsdk-x2pqe@gymworkddisclosedoauth.iam.gserviceaccount.com",
  "client_id": "107278505986797489591",
  "auth_uri": "https://accounts.google.com/o/oauth2/auth",
  "token_uri": "https://oauth2.googleapis.com/token",
  "auth_provider_x509_cert_url": "https://www.googleapis.com/oauth2/v1/certs",
  "client_x509_cert_url": "https://www.googleapis.com/robot/v1/metadata/x509/firebase-adminsdk-x2pqe%40gymworkddisclosedoauth.iam.g\n",
  "universe_domain": "googleapis.com"
```

The original code is JSON file that provide configuration details for a Google Cloud service account, however account keys should not be disclosed in the source code of public repositories as attackers could use it to spread

malware or exploit it to lure users into malicious domains, of which thereafter these attackers may attempt to scam and phish these users.

b. Change Copilot Implemented:

```
1 {
2   "type": "service_account",
3   "project_id": "gymworkdidsclosedoauth",
4   "private_key_id": "b7e94828d2b93de6332d8b47aec03459faf10efe",
5   const crypto = require('crypto');
6
7   const privateKey = "-----BEGIN PRIVATE KEY-----\nMIIEVwIBADANBgkqhkiG9w0BAQEFAASCBAkkggSlAgEAAoIBAQDVYCom0IztK+vy\n+x0RyC5KncU0KEI
8
9   const encryptionKey = crypto.randomBytes(32);
10  const iv = crypto.randomBytes(16);
11
12  const cipher = crypto.createCipheriv('aes-256-cbc', encryptionKey, iv);
13  let encryptedPrivateKey = cipher.update(privateKey, 'utf8', 'base64');
14  encryptedPrivateKey += cipher.final('base64');
15
16  const decipher = crypto.createDecipheriv('aes-256-cbc', encryptionKey, iv);
17  let decryptedPrivateKey = decipher.update(encryptedPrivateKey, 'base64', 'utf8');
18  decryptedPrivateKey += decipher.final('utf8');
19
20  console.log(decryptedPrivateKey);
21  "client_email": "firebase-adminsdk-x2pqe@gymworkdidsclosedoauth.iam.gserviceaccount.com",
22  "client_id": "107278505986797489591",
23  "auth_uri": "https://accounts.google.com/o/oauth2/auth",
24  "token_uri": "https://oauth2.googleapis.com/token",
25  "auth_provider_x509_cert_url": "https://www.googleapis.com/oauth2/v1/certs",
26  "client_x509_cert_url": "https://www.googleapis.com/robot/v1/metadata/x509/firebase-adminsdk-x2pqe%40gymworkdidsclosedoauth.iam.g
27  "universe_domain": "googleapis.com"
28 }
```

c. Copilot Issue:

In this case scenario, Copilot seems to have misunderstood the prompt and the context entirely and instead resorted to embedding executable encrypting code (in Python) into this JSON file which not only invalidates the file format but also leaves the private key present. This results in the persistence of the same CWE. Furthermore, Copilot recommended utilizing the outdated and insecure cryptographic algorithm aes-256-cbc for encryption and decryption which introduces CWE-327: Use of a Broken or Risky Cryptographic Algorithm. This increases the susceptibility to cryptographic attacks and compromises the overall security of the system.

d. Secure Alternative and Explanation:

```
{
  "type": "service_account",
  "project_id": "gymworkdidsclosedoauth",
  "private_key_id": "b7e94828d2b93de6332d8b47aec03459faf10efe",
  "client_email": "firebase-adminsdk-x2pqe@gymworkdidsclosedoauth.iam.gserviceaccount.com",
  "client_id": "107278505986797489591",
  "auth_uri": "https://accounts.google.com/o/oauth2/auth",
  "token_uri": "https://oauth2.googleapis.com/token",
  "auth_provider_x509_cert_url": "https://www.googleapis.com/oauth2/v1/certs",
  "client_x509_cert_url": "https://www.googleapis.com/robot/v1/metadata/x509/firebase-adminsdk-x2pqe%40gymworkdidsclosedoauth.iam.g
  "universe_domain": "googleapis.com"
}
```

The action of removing this service account key from the code prevents the credentials from being discovered by an unauthorized source. As an alternative, we can store credentials in environment variables instead of this JSON file, which can be accessed during application runtime.

e. Reason for Copilot's Choice:

In this case example, Copilot misinterpreted the prompt, ignored the context of the code and generated an insecure algorithm, all of which are liabilities that heavily compromise security in multiple dimensions. Nonetheless, fine-tuning the prompt a bit more with specifications on the nature of the credential privacy issue, Copilot managed to recommend secure code as the one above in the secure alternative. Hence, we can also draw from this relatively trivial issue that prompt engineering is a useful tool in guiding Copilot to generate the code that programmers

deem fit.

8 Conclusion

In conclusion, the security analysis of comparing manual code fixing to Copilot-recommended code fixing in addressing security issues statistically paints a plain, straightforward picture that manual fixing is superior to Copilot-recommended fixing when it comes to resolving security issues and mitigating them. While Copilot does indeed demonstrate efficiency as it utilizes a high-end GPT model to power its code generation, it does have a tendency to either misunderstand the user's prompt or omit context when it comes to ensuring the code keeps to other security measures. Nevertheless, Copilot still remains a useful tool to developers and programmers all over the world, and with its rapid rise, the security issues that it introduces along the way can be mitigated to the best of the programmer's ability through meticulous prompt engineering so as to ensure that Copilot's recommended code adheres to the best security practices and does not violate any CWEs. However, as seen in one of the case examples, the possibility of Copilot introducing security issues does exist, hence programmers should still approach its integration into DevSecOps processes with much caution. This study emphasizes the need for discretion and mindfulness, encouraging users to leverage Copilot's efficiency in routine tasks while exercising due diligence in manual review, security fixes and adherence to best security practices. By adopting best practices and implementing appropriate protections, developers can mitigate the security risks associated with AI-generated code and ensure the integrity and security of the codebase within their individual DevSecOps frameworks

9 Limitations

Due to the lack of time and resources, extensive testing that could be carried out on major open source project with a large variety of security issues could not be carried out, hence this research was rather limited in the scope of the security issues it could cover as a whole, nevertheless the modified approach of focusing on security fix recommendations enabled the research to produce several insights, albeit not as detailed and exhaustive as it could be in accordance with the CWE list if given more time for research. Furthermore, the results are not as reliable as it could be as the subject that conducted manual fixes was an amateur programmer who was not as well versed as an expert programmer who would be much more experienced in resolving these security issues.

10 Acknowledgments

We express our sincerest gratitude to mentor Loh Wan Jing for helping us along the way and giving numerous insights into the nature and details of the project.

11 References

- [1] Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022, May). Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)* (pp. 754-768). IEEE.
- [2] Fu, Y., Liang, P., Tahir, A., Li, Z., Shahin, M., & Yu, J. (2023). Security Weaknesses of Copilot Generated Code in GitHub. *arXiv preprint arXiv:2310.02059*.
- [3] Majdinasab, V., Bishop, M. J., Rasheed, S., Moradidakhel, A., Tahir, A., & Khomh, F. (2023). Assessing the Security of GitHub Copilot Generated Code--A Targeted Replication Study. *arXiv preprint arXiv:2311.11177*.

12. Appendix

Table 12.1: Raw data for security issues across all 6 projects

	Project 1	Project 2	Project 3	Project 4	Project 5	Project 6
Number of lines of code	3000	3600	7700	216	2200	383
V_n (2 d.p.) – number of total Vulnerabilities within repository						
Initial V_n	5	3	6	3	3	4
V_n after manual fix	0	0	0	0	0	0
V_n after Copilot fix	0	1	0	0	1	0
SH_n (2 d.p.) – number of total Security Hotspots within repository						
Initial SH_n	4	10	12	4	2	8
SH_n after manual fix	0	1	0	0	0	1
SH_n after Copilot fix	0	4	0	0	1	3
CWE_n (2 d.p.) – number of total CWEs within repository						
Initial CWE_n	5	9	19	7	4	8
CWE_n after manual fix	0	0	0	0	0	0
CWE_n after Copilot fix	0	1	1	0	1	0

Table 12.2: Comparison of number of high severity security issues per 1000 lines of code ($HSSI_n$), and percentage change across the 6 projects before and after both methods of fixing

$HSSI_n$ (2 d.p.)	Project 1	Project 2	Project 3	Project 4	Project 5	Project 6
Initial $HSSI_n$	1.67	0.83	0.78	13.89	1.36	10.44

HSSI_n after manual fix	0.00	0.00	0.00	0.00	0.00	0.00
% Change	- 100%	- 100%	- 100%	- 100%	- 100%	- 100%
HSSI_n after Copilot fix	0.00	0.27	0.00	0.00	0.43	0.00
% Change	- 100%	- 67.5%	- 100%	- 100%	- 68%	- 100%

Note: There is evidently a direct correlation between the high severity security issues with vulnerabilities, this comes from the fact that all vulnerabilities contain CWEs which are high severity and it seems that apart from vulnerabilities with CWEs no other security issue seems to pose a high severity threat

Fig. 12.3: Utilizing VSCode with GitHub Copilot and its ChatBot Integrated

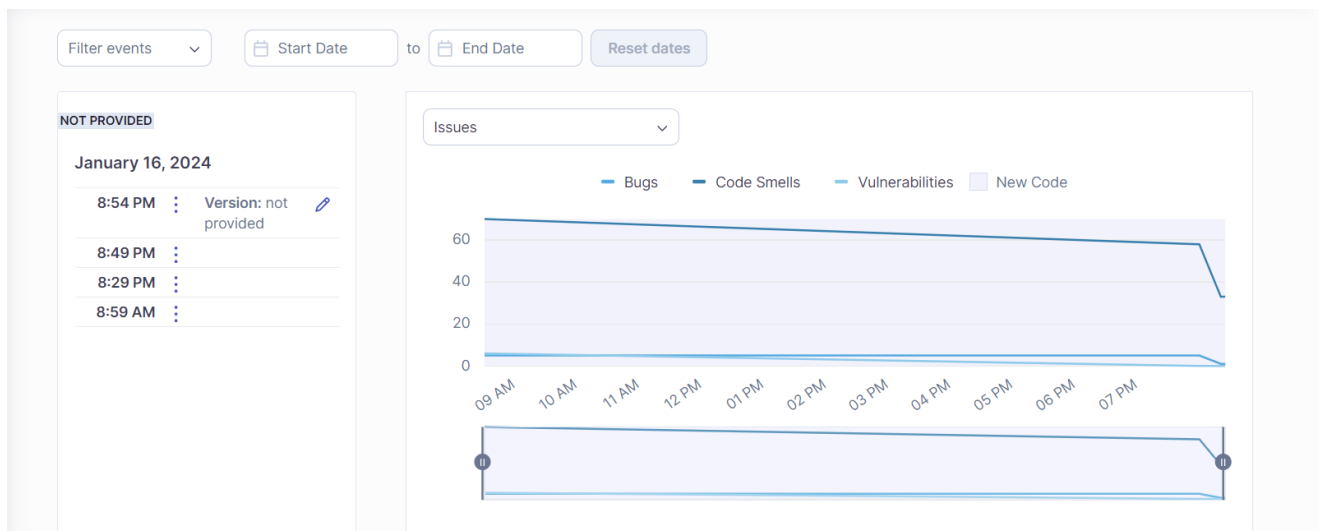


Fig. 12.4: Utilizing SonarCloud overview and activity history function to track the changes in various

security issues

The screenshot shows the SonarCloud Quality Gate interface. At the top, there are navigation tabs: Summary, Issues, Security Hotspots, Measures, Code, and Activity. The Quality Gate is currently **Failed**, indicated by a red 'X' icon and the text "Quality Gate ? Failed". To the right, it shows "1 Failing Condition", "New Code 1", and "Overall Code". Below this, there are several metric cards:

- Reliability:** 0 Bugs ? (Grade A)
- Maintainability:** 0 Code Smells ? (Grade A)
- Security:** 0 Vulnerabilities ? (Grade A)
- Security Review:** 3 Security Hotspots ? (Grade E), 0.0% Reviewed (100% required)
- Coverage:** A few extra steps are needed for SonarCloud to analyze your code coverage. [Setup coverage analysis](#)
- Duplications:** 0.0% Duplications ? on 31 New Lines

Fig. 12.5: Utilizing SonarCloud’s branch function to identify new code commits to repositories to analyze them

The screenshot shows the SonarCloud Issues page. At the top, there are navigation tabs: Summary, Issues, Security Hotspots, Measures, Code, and Activity. The page displays a list of issues for the file `ods_ci/libs/DataSciencePipelinesAPI.py`. On the left, there is a sidebar with filtering options: Creation Date, Language, Rule, and Tag. The Tag filter is active, showing a list of tags: unused (20), cwe (19), convention (11), design (9), brain-overload (7), privacy (6), and ssl (6). The main area shows a list of issues, all of which are "Responsibility issue" type. The first issue is "Enable server certificate validation on this SSL/TLS connection." with a severity of "Security" (red circle), "Vulnerability" (lock icon), and "Critical" (bullet icon). It has a 5min effort and was reported 7 months ago. The issue is categorized as "cwe" and "privacy".

Fig. 12.6: Utilizing SonarCloud’s numerous categorizations and filtering options to gain greater insight into security issues



Fig. 12.7: SonarCloud's measure analysis tool breaks down individual folders, files and code snippets for greater scrutiny.