PS-1: Development of a software tool using LLMs or similar AI based techniques for Detection of Vulnerabilities (Malicious Code) in Source Code of Software (especially Open-Source software) and suggest Mitigation Measures

I. <u>Short Summary</u>

1. In the fast-evolving world of software development, one thing remains constant, the need for security. As the use of open-source software continues to soar, vulnerabilities lurking in the codebase pose significant risks, that could be exploited by malicious actors.

2. Open-source software, while offering tremendous innovation, is also a double-edged sword. Developers worldwide collaborate to build these projects, but they don't always have the resources to identify and fix security flaws quickly. And as more organizations rely on these systems, the threat becomes even more critical.

3. This is where the challenge lies and we need a faster, more efficient way to detect these vulnerabilities as the traditional methods of vulnerability detection rely heavily on manual reviews and outdated tools.

4. Through the AI Grand Challenge, we are looking forward to getting a cutting-edge software tool developed that harnesses the power of Large Language Models (LLMs) and similar AI techniques to detect vulnerabilities in open-source software and also suggest mitigation measures to address these risks.

5. With this tool, we are hoping to empower developers to create secure, reliable, and resilient open-source software at scale.

6. The AI Grand Challenge is just the beginning. Together, we can create the next generation of security tools for open-source software. Join us in this mission, and let’s build safer, more secure open-source software for everyone.

"Together, we can make open-source software safer, one line of code at a time."

II. Detailed Description

1. The development of a large language model (LLM) or similar AI techniques-based tool for detection of vulnerabilities (Malicious Code) in source code of software (especially open-source software) and suggest mitigation measures, could provide a range of capabilities that assist developers in identifying, mitigating, and preventing security vulnerabilities and malicious behaviour. Here’s a breakdown of what such a product would be able to do:

2. **Malicious Code Detection**

a. **Identify Malicious Patterns**: The tool should be able to analyse source code and identify patterns indicative of malicious behaviour. This could include detecting backdoors, trojans, spyware, or any type of code that could be used for unauthorized access or control over the system.

b. **Suspicious Code Snippets**: It should flag suspicious or non-idiomatic code that might be indicative of attempts to conceal malicious actions, such as encoded payloads, obfuscated code, or unusual use of libraries.

c. **Code Behaviour Analysis**: Instead of just static pattern recognition, the tool should preferably employ dynamic analysis to evaluate how certain parts of the code behave when executed, identifying whether they perform any suspicious actions like network communication, privilege escalation, or data exfiltration.

3. Vulnerability Detection

a. **Common Vulnerabilities and Exposures (CVEs)**: The tool should be able to detect and list down the known CVEs (such as buffer overflows, SQL injections, cross-site scripting (XSS), etc.) by scanning the code for patterns that match or resemble vulnerable code structures.

b. **Unknown / Zero-Day Vulnerabilities**: The tool should also attempt to detect and list potential unknown/ zero-day vulnerabilities that are not yet publicly known but may be exposed in certain code patterns or misconfigurations.

c. **Dependency Vulnerabilities**: The tool should have the functionality to check the open-source dependencies in the code for known vulnerabilities, helping to secure the entire software ecosystem, not just the user’s custom code.

d. **Severity Ranking**: The tool should allow the user to filter and prioritize vulnerabilities based on severity, so they can focus on the most critical issues first.

e. **Risk Assessment**: Based on the detected issues, the tool should be able to provide a risk assessment that helps prioritize which vulnerabilities should be addressed first, taking into account factors like exploitability and potential impact.

4. Code Quality and Best Practices Enforcement

a. **Code Review and Suggest Improvements**: In addition to detecting malicious code and vulnerabilities, the tool should enforce best practices by suggesting code quality improvements and safer alternatives, such as avoiding deprecated functions or ensuring proper validation of inputs.

b. **Automated Code Audits**: The tool should help in automating code audits, providing developers with reports on potential security flaws and providing suggestions on how to mitigate them.

c. **Compliance Assistance**: By further fine tuning, the tool should be able to assist organizations in ensuring compliance with relevant regulations and standards (e.g., OWASP, PCI DSS) by flagging violations and adhering to the standard best practices.

5. Mitigation Measures and Recommendations

a. **Automated Patches and Fixes**: Upon identifying vulnerabilities or malicious code, the tool should suggest or even attempt to automate fixes based on established mitigation patterns. For example, replacing unsafe functions with more secure alternatives or patching known vulnerabilities.

b. **Code Hardening**: It should be able suggest security hardening techniques (e.g., input validation, encryption, least privilege principle) that would mitigate the risk of exploiting vulnerabilities.

c. **Customizable Security Rules**: Developers could customize the tool to enforce specific security policies or coding standards relevant to their project or organization.

6. Reporting and Documentation

a. **Detailed Security Reports**: The tool should be able to generate detailed reports that explain the vulnerabilities or malicious code identified, their severity, and how to fix them. This would be helpful both for developers as well as for security auditors.

7. **User-friendly Interface**

a. **Real-time Analysis and Feedback**: The tool should be able to integrate with popular IDEs (e.g., Visual Studio Code, IntelliJ IDEA) to provide developers with real-time feedback as they write or review code, if required.

b. **Collaboration Tools**: The tool should allow user teams to collaborate on security issues, leaving comments on detected vulnerabilities and tracking their resolution progress when used in collaborative testing mode, if required.

8. **Adaptive Learning and Customisation**

a. **Learning from False Positives/Negatives**: Over time, the tool should learn from its false positives and false negatives, becoming more accurate as it receives feedback and adapts to specific projects, environments, and development patterns.

b. **Customisable Sensitivity**: It should allow users to adjust the sensitivity of vulnerability detection based on project needs—for example, tightening detection in high-risk applications (e.g., banking software) while relaxing rules.

10. Ultimately, the developed tool would combine the power of LLMs or similar AI based techniques for natural language processing and machine learning with traditional vulnerability detection and mitigation techniques, making it a highly valuable tool for open-source software development teams.

III. <u>Evaluation Parameters and Criteria:</u>

Ser No.	**Evaluation Parameters**	**Remarks**	**Weightage (%)**
STAGE – I [Shortlisting 15-20 from all entries]			
1	Languages Supported (for standalone software, web applications)*	How many out of Java, Python, C/C++/C#, PHP.	30

2	No. of Vulnerabilities detected by the tool and whether it is able to map the vulnerabilities with their CVEs, CWEs (if available). **(a) Critical & High (weightage 60%)** **(b) Medium (weightage 30 %)** **(c) Low (weightage 10%)**	Like OWASP Top 10, CWE-Top 25, memory safety, injection, misconfiguration, etc.	40
3	Detection Accuracy	This will be evaluated based on the F1 score.	30
STAGE – I [Physical Evaluation of Shortlisted Participants to Select Top 6 at the end of Stage 1]			
1	Languages Supported (for standalone software, web applications)*	How many out of Java, Python, C/C++/C#, PHP.	20
2	No. of Vulnerabilities detected by the tool and whether it is able to map the vulnerabilities with their CVEs, CWEs (if available). **(a) Critical & High (weightage 60%)** **(b) Medium (weightage 30 %)** **(c) Low (weightage 10%)**	Like OWASP Top 10, CWE-Top 25, memory safety, injection, misconfiguration, etc.	30

3	Detection Accuracy	This will be evaluated based on the F1 score.	20
4	Approach	Start-up need to present Solution based on (a) Methodologies used (b) Architecture (c) Scalability (d) Resources Utilisation	30
	STAGE – II		
Ser No.	**Evaluation Parameters**	**Remarks**	**Weightage (%)**
1.	Languages Supported (for mobile applications, standalone software, web applications)*	How many out of Ruby, Rust, Kotlin, Swift, HTML, Javascript, Go (Golang) and the languages mentioned in Stage I covered.	20
2.	• No. of Vulnerabilities detected by the tool and whether it is able to map the vulnerabilities with their CVEs, CWEs (if available). **(a) Critical & High (weightage 60%)** **(b) Medium (weightage 30 %)** **(c) Low (weightage 10%)** • Mitigation measures suggested	Like OWASP Top 10, CWE-Top 25, memory safety, injection, misconfiguration, etc. Mitigation measures (Yes or No). If yes, whether they can be implemented without hampering the functionality and security of the software itself?	50

	• Granularity of Detection	Whether the tool is able to locate and mark the exact code segment, line or not?	
3.	Performance of the tool based on the Processing Time for scanning and analyzing the software code	This would be measured in terms of either how much average processing time per line of code or per KB/ MB of code.	10
4.	Explainability of Decisions taken by the tool (Proof to verify the output)	Can the model explain why a piece of code is vulnerable or how the fix helps? Whether the tool supports or provides security annotations or traceability to CVE references	10
5.	Approach	Start-up need to present Solution based on (a) Methodologies used (b) Architecture (c) Scalability (d) Resources Utilisation	10
STAGE - III			
Ser No.	**Evaluation Parameters**	**Remarks**	**Weightage (%)**
1.	No of vulnerabilities detected **(a) Critical & High (weightage 60%)** **(b) Medium (weightage 30 %)** **(c) Low (weightage 10%)**	Three categories of applications - Standalone, Mobile and Web applications	40

2.	Languages supported (for standalone software, web applications)*	Whether the solution supports all languages mentioned in Stage I and II	25
3.	Mitigation Measures Suggested and functionality of automated code correction.	Yes or No. If yes, whether they can be implemented without hampering the functionality and security of the software itself?	20
4.	Scalability of the tool Documentation of the tool Usability of the tool	Whether the tool is scalable in terms of deployment across enterprises or used for bigger open-source software or it is just a prototype and cannot be scaled up? What is the quality of the tool documentation in terms of its user manual etc. Whether the tool has a user-friendly UI so that the learning curve to use the tool is minimal and it can be easily installed and run to achieve the end objective?	15

The tool should meet minimum threshold set for the particular language.

Submission Format

1. The participants are required to submit the findings of first stage in excel sheet in following format:-

SNo	**Primary Language of Benchmark**	**Vulnerability**	**CVE ID**	**Severity**	**Common Weakness Enumeration (CWE) Id**	**file name with path**	**line number**	**Code Snippet at the line**
1	Java/PHP/ C++	Denial-of-service (DoS) XXXXXXXXXXXXXXXXXXXXXXXXXX	CVE-XXXXXXXX	High	CWE-XXX			

2. The file name should be GC_PS_01_Startup_name.

IV. Indicative Datasets for Training and Testing and Evaluation

1. It may be noted that since this problem statement focusses on the detection of vulnerabilities (malicious code) in source code of software (especially open source), the participating teams are free to choose the datasets available in open domain for training their respective models for improving the functionality of the tool.

2. It is pertinent to mention here that three types (Standalone software, web application and mobile applications) of software codes are envisaged to be tested using the tool and the performance of the tool would be evaluated based on the output generated by the tool and fulfilment of evaluation parameters stage wise.

3. The common programming languages on which the performance of the tool would be evaluated are mentioned in the evaluation parameters matrix. The categories of applications have also been listed viz. Mobile Applications, Standalone Software Applications and Web based Applications.

4. For the ease of participating startups to train their models and test the performance, some illustrative datasets available in the open domain for all categories of applications are mentioned below:

Ser No.	Dataset Name	Description/ Link/ Remarks
	Software Assurance Reference Dataset (SARD)	Real and synthetic vulnerable code samples https://samate.nist.gov/SARD/
	Devign	GitHub-based dataset with vulnerable/non-vulnerable labels https://github.com/epicosy/devign
	CodeXGLUE (Defect)	Dataset for defect prediction and repair https://github.com/microsoft/CodeXGLUE
	Multi-language dataset (Oct 2024): supports C, C++, Java, JS, Go, PHP, Ruby, Python with CWE/CVE labels and patches (Zenodo)	https://zenodo.org/records/13870382
	MegaVul (C/C++)	Vulnerabilities from repositories, CVE-linked, JSON format, ideal for detection and severity tasks

		https://github.com/Icyrockton/MegaVul
	DiverseVul	Vulnerable functions across CWE types plus nonvulnerable examples, excellent for fine-grained CWE classification https://github.com/wagner-group/diversevul
	GITHUB Vulnerability Dataset Open Source	https://github.com/CAE-Vuldataset/CAE-Vuldataset
	GitHub – Vulnerability Dataset: A dataset for vulnerability detection and program analysis	https://github.com/ppakshad/VulnerabilityDataset
	Real CVE Patches	From GitHub projects or National Vulnerability Database (NVD)

5. For evaluation of the tools submitted by the startups, datasets would be selected either from the above collections or similar collections of open-source software source codes. The evaluation across the three stages would be limited to the parameters and languages mentioned in the evaluation parameters and criteria.

6. For shortlisting, the startups would be given datasets 4 days prior (on, 28 Oct 2025 at 10:00am) to the Stage I submission deadline (i.e., 31 Oct 2025, Midnight) and the startups are required to submit their results of their respective tools based on these datasets. A private leaderboard will be made and at most, top 15-20 startups would be selected for final evaluation of Stage I. The shortlisted participants will be published along with the cutoff score as per the evaluation criteria. Participants individual scores will be shared over the email. The number may vary based on the overall performance at the discretion of theJury for this Problem Statement.

7. The shortlisted startups would be called for demonstration of the practical capabilities of their tool either in person or through VC and the performance of the respective tools would be evaluated using the ‘Holdout’ datasets for this purpose to select the final 6 winners (Max) of Stage I. The Holdout datasets would be released at the end of Stage I just before this evaluation.

8. The demonstration and evaluation of stage II and stage III would be in physical mode only in accordance with the Evaluation Matrix of stage II and III. The test datasets for these would be released during evaluation.

Note - The startups using these datasets are required to adhere to the terms and conditions of usage of these datasets as mentioned on their websites.