Using ChatGPT as a Static Application Security Testing Tool

Atieh Bakhshandeh

Abdasamad Keramatfar, Amir Noruzi, Mehdi Chekidehkhoun

Research Center for Development of Advanced Technologies

bakhshandeh@rcdat.com

keramatfar#@rcdat.ir

norouzi@rcdat.ac..ir

chekidekhoon@rcdat.ir

Introduction-SAST

- Software vulnerability is a technical vulnerability that can be used for violating its security policies
- Such vulnerabilities can be exploited which in turn leads to data leakage and tampering and even denial of services.
- Static source code analysis is a method for finding code vulnerabilities that is done by automatically examining the source code without having to execute the program
- Static Application Security Testing (SAST) tools analyze a piece of code or a compiled version of it in order to identify its security problems
- Covering a wide range of errors and high accuracy are two important features of SAST tools
- Most of the SAST tools often use rule-based techniques to find the vulnerable patterns of code
- However, these tools have shown to have their own flaws, including a high rate of false positives and false negatives

Introduction-ML

- In recent years, Machine Learning (ML) and deep learning have been of great attention in the field of code processing tasks such as vulnerability detection.
- Machine learning models can automatically learn the patterns of software vulnerabilities based on datasets
- Research indicates that ML models have fewer false positives compared to SAST tools [1]

Introduction-ChatGPT

- Recently, ChatGPT, an AI-powered chatbot tool has drawn a lot of attention
- ChatGPT uses Natural Language Processing (NLP) and machine learning algorithms to understand and respond to customer inquiries
- ChatGPT has been trained with a huge amount of data till 2021 so that it can be a great help in finding known patterns in thousands of packages in automated way.
- The model is also trained on a large amount of code and is thus able recognize common patterns
- In this paper, we evaluate the performance of ChatGPT in identifying security vulnerabilities of Python codes and compare the results with three well-known SAST tools for Python vulnerability detection (Bandit, Semgrep and SonarQube)
- In this study, we used the GPT-3.5-turbo model
- The GPT-3.5-Turbo model is a superior option compared to the GPT-3 model, as it offers better performance across all aspects while being 10 times more cost-effective per token.

Datasets Description

- Our dataset consists of 156 Python code files

1. 130 files of the securityEval dataset [1] these 130 files cover 75 vulnerability types that are mapped to Common Weakness Enumeration (CWE) .
2. 26 files belong to a project called PyT in which the author developed a tool for Python code vulnerability detection[2]

- A security expert of our team rechecked the data and specified the vulnerable line

Working with ChatGPT API

- To use the model for our experiments, we put all the vulnerable python codes of our dataset in a directory and called GPT-3.5 API with an optimized prompt for each of the vulnerable python files.
- The choice of prompt is the most challenging task in this process, as it has direct effect on the results the model provides.
- According to [3], some of the key points to create an optimized prompt are:
  - Using delimiters such as triple quotes or triple backticks to specify the piece of code
  - Asking for structured output such as JSON, HTML, etc.
  - Specifying the steps to complete a task in a clear way
  - Instructing the model to work out its own solutions before rushing to a conclusion

Working with ChatGPT API

- The temperature can take the values between 0 to 6 with 6 giving the most random output
- We did four kinds of experiments.

Experiment 1

1. In our first experiment, we give the model the vulnerable files and ask it whether they contain any security vulnerabilities or not, without specifying the corresponding CWEs. We ask the model to just return the line number of the vulnerability if it contains any. Then, we compare these lines with ground truth labels. In effect, this experiment is a binary classification

Experiment 1

- Prompt

> You will be provided with a python code delimited by triple backticks. If it contains any security vulnerability, identify the lines of vulnerable code and only write the line in quotation. If the code does not contain a vulnerability, then simply write None. Python code: ’’’{Vul_code}’’’

- Sample response

> Line 117: "self.login(password='password1') "
>
> Line 120: "response = self.client.get(reverse('login'))"

Experiment 2

2. In our second experiment, we provide the list of the corresponding CWEs and ask the model to find the vulnerabilities from the labels' list in the Python vulnerable file. In this experiment, we ask the model to respond in JSON format like [{"label": "CWE-X", "line of Code": "line no."}] so we can compare our results with those of SAST tools

Experiment 2

- Prompt

> Which of the following vulnerabilities from list of vulnerabilities exist in the python code which is delimited with triple backticks. Also give the line of the vulnerability in the code. Python code:‴{Vul_code}‴ list of vulnerabilities: {", ".join(labels1)}. Format your response as a list of JSON objects with "label" and "line of Code" as the keys for each element. only answer with JSON.

- Sample response

```
[
 {
   "label": "CWE-79",
   "line of Code": "username = request.args.get('username')"
 },
 {"label": "CWE-94",
  "line of Code": "response = make_response(username)"
 }
]
```

Experiment 3

3. In the third experiment, for each of vulnerable files, we give the model all the labels returned from Bandit, Semgrep and SonarQube tools for the Python code, as the classes that ChatGPT should use. We then ask the model whether each vulnerable file contains any of those vulnerabilities or not? Here, the main difference with our second experiment is that we specify the classes per vulnerable file separately. In other words, we use the model as an assistant for the SAST tools to verify the detected vulnerabilities by them. In this experiment, we use the same JSON format as the second experiment for the responses. Note that in this experiment, although we provide the labels' list beforehand for each vulnerable file, in some cases the model has returned a new CWE which is not among its input labels. This is a natural behavior seen from a language model and in order to address this issue in our evaluation, we consider two cases:

   - In one case, we ignore the new labels and calculate the metrics without considering them. This policy can reduce the number of false positives of SAST tools.
   - In another case, we consider them as well and this time the number of false negatives may decrease.

Experiment 3

- Prompt

Which of the following vulnerabilities from list of vulnerabilities exist in the python code which is delimited with triple backticks. Also give the line of the vulnerability in the code. Python code:‴{Vul_code}‴ list of vulnerabilities: {”, ”.join(labels1)}. Format your response as a list of JSON objects with ”label” and ”line of Code” as the keys for each element. only answer with JSON.

- Sample response

```
[
 {
  "label": "CWE-78",
  "line of Code": "command = 'echo ' + param + ' >> ' + 'menu.txt'"
 },
 {
  "label": "CWE-94",
  "line of Code": "subprocess.call(command, shell=True)"
 },
 {
  "label": "CWE-489",
  "line of Code": "param = request.form['suggestion']"
 }
]
```

Experiment 4

4. In our fourth experiment, we do not provide any label list for the model and ask it to detect the vulnerabilities in the files and determine their corresponding CWEs from its own trained knowledge. Here, the format of the responses is the same JSON structure in the previous experiments.

Experiment 4

- Prompt

Your task is to determine whether the following python code which is delimited with triple backticks,is vulnerable or not? identify the following items:
- CWE of its vulnerabilities.
- Lines of vulnerable code.

Format your response as a list of JSON objects with ”label” and ”line of Code” as the keys for each vulnerability. If the information isn’t present, use ”unknown” as the value. Make your response as short as possible and only answer with JSON. python code:‴{Vul_code}‴

- Sample response

```
[{"label": "CWE-352", "line of Code": "data = yaml.load(f)"}]
```

Experiment Setup

- We give a dataset of 156 vulnerable python codes to Bandit, Semgrep and SonarQube SAST tools and we also query the ChatGPT model with our dataset using the appropriate prompts
- We then calculate the precision, recall and F1-score metrics for each of the tools' results and the model result based on our ground truth labels.

$$precision = \frac{TP}{TP + FP} \qquad recall = \frac{TP}{TP + FN} \qquad F = 2 \times \frac{precision \times recall}{precision + recall}$$

- Note that in the experiments, the order of the given labels to GPT-3.5 model has high impact in the generated results from the model.
- Therefore, we gave the labels in a random order

Results-Exp.1

	Precision	Recall	F1
Semgrep	0.6694	**0.1504**	**0.2457**
Bandit	0.7450	0.1447	0.2424
SonarQube	**0.9104**	0.1161	0.2060
GPT-3.5	0.7413	0.0819	0.1475

Results of Experiment 1 (Binary classification)

- The precision for the model in this experiment is not better than other three tools
- The low recall suggests that using this model for only detecting vulnerable lines of a code does not give any better results than SAST tools

Results-Exp.2

	Precision	Recall	F1
Semgrep	**0.4682**	**0.1123**	**0.1812**
Bandit	0.3168	0.0609	0.1022
SonarQube	0.3283	0.0419	0.0743
GPT-3.5	0.1659	0.0761	0.1044

Results of Experiment 2 (Selecting from the list)

- These results also indicate that using GPT-3.5 model with all the classes given as labels, does not provide superior results in comparison with the SAST tools.

Results-Exp.3

	Precision	Recall	F1
Semgrep	0.4682	0.1123	0.1812
Bandit	0.3168	0.0609	0.1022
SonarQube	0.3283	0.0419	0.0743
Experiment3,GPT-3.5-Case 1	**0.7807**	**0.2781**	**0.4101**
Experiment3,GPT-3.5-Case 2	0.333	0.1542	0.2109

Table 5. Results of Experiment 3 (SAST assistant)

- Case 1, in which we do not accept the new labels returned from the model, has produced better results than case 2.
- Case 1 results are even significantly better than those of SAST tools.
- This behavior shows that using ChatGPT as an assistant along with SAST tools can be a good idea.

Results-Exp.4

	Precision	Recall	F1
Semgrep	**0.4682**	0.1123	**0.1812**
Bandit	0.3168	0.0609	0.1022
SonarQube	0.3283	0.0419	0.0743
GPT-3.5	0.3350	**0.1238**	0.1808

. Results of Experiment 4 (Free Classification)

- If we do not provide any labels for the model and ask it to return the CWEs of the vulnerable codes from its own knowledge, as we did in experiment 4, we obtain the above results which are comparable to the SAST tools.

Threats to Validity

- Our biggest challenge was the choice of the prompts of ChatGPT. We tried to choose the most efficient prompts in terms of naturalness and expressiveness and based on [3]. However, it is possible that a more careful selection of the prompt can affect the results.
- Another factor which may also affect the results is the size of the dataset and its accessibility on the Internet.
- The distribution of the CWEs of the dataset is of great importance. To overcome this threat, we chose two different datasets for better generalization of the vulnerabilities they cover, but there may be still few coverages of the vulnerabilities.
- We only compare this model with three SAST tools for Python language. Perhaps, further SAST tools affect the results.
- We only test GPT-3.5 model of ChatGPT and it is possible that the new billable version (GPT-4) performs better than this version.

Conclusion

- In this paper, we did four types of experiments with ChatGPT model to detect the security vulnerabilities of Python codes. We compared this model with Bandit, Semgrep and SonarQube that are popular SAST tools for Python codes.
- We concluded that using GPT-3.5 model for vulnerability detection of codes in some especial manners gives promising results. Specifically, if we use it as SAST tool assistant, it will produce results that can help to improve the returned results of SAST tools, since it reduces false positive and false negative rates.

Conclusion

- We admit that this study is not general from all aspects and provides primary steps toward this path.
- In future studies:
  - The behavior of the latest model of ChatGPT (GPT-4) which is more powerful than GPT-3.5 model, can be examined
  - The Temperature parameter of the model can be set to values other than zero
  - Some innovative rules can be passed to decide for the most efficient obtained results
  - Using of one-shot learning
- One of the security concerns of using ChatGPT as SAST tools' assistant is that, it is required to upload the source code on the OpenAI servers.

Our GitHub Repository URL

https://github.com/abakhshandeh/ChatGPTasSAST.git

References

1. Perl H, Dechand S, Smith M, Arp D, Yamaguchi, Rieck K, and et al. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015.*
2. Mohammed Latif Siddiq and Joanna CS Santos.Securityeval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*, pages 29–33,2022.
3. Bruno Thalmann Stefan Micheelsen. Pyt: A static analysis tool for detecting security vulnerabilities in python web applications, 2016.
4. Isa Fulford Andrew Ng. Chatgpt prompt engineering for developers. https://www.deeplearning.ai/short-courses/chatgptprompt-engineering-for-developers, April 2023. Accessed: 2023-04-27.