Detecting BOLA Vulnerabilities with Large Language Models

EMILS JOHANSENS, University of Twente, The Netherlands

Broken Object Level Authorization (BOLA) is widely recognized as one of the most critical API security risks. Since its initial inclusion in the 2019 publication, the vulnerability has retained the number-one ranking in the OWASP 2023 API Security Top Ten. Detecting BOLA attacks manually is labour-intensive and error-prone, and the existing automated tools do not provide full coverage over every API. The paper investigates whether large language models (LLMs) can effectively identify BOLA vulnerabilities in REST APIs. The research first presents a dataset of 12 REST APIs, described in the OpenAPI 3.0 format. A prompt engineering approach is then employed by giving the LLM a context-rich and role-specific prompt and asking it to identify BOLA vulnerabilities. Four state-of-the-art LLMs are evaluated using the dataset, and their outputs are compared against the ground truth. The results show that LLMs achieve high accuracy and recall but suffer from low precision, producing many false positives. Model performance is compared against each other, and the Deepseek-R1 model achieves the best overall performance. Lastly, small-parameter LLMs are explored; however, the output shows a fundamental lack of knowledge in cybersecurity.

Additional Key Words and Phrases: Broken Object Level Authorization, BOLA, LLM, Cybersecurity

1 INTRODUCTION

Application Programming Interfaces (APIs) have become widespread and nearly unavoidable in modern web architecture, with the most frequently used architectural style for web APIs being Representational State Transfer (REST). However, increasing reliance on APIs has led to an increase in cyber-threats, with API related attacks becoming a significant security concern.[10]

In its recent publication of the Open Worldwide Application Security Project (OWASP) 2023 API Security Top 10, the foundation ranked Broken Object Level Authorization (BOLA) as the most critical vulnerability affecting modern APIs.[8] This issue is pervasive in API-based applications because the server component typically does not fully track the client's state, instead relying more on parameters such as object IDs, which are sent from the client to determine which objects to access.[9] Executing a BOLA attack typically requires knowledge of the object-identifier names within an API request. It is possible to execute a BOLA attack using brute force successfully.[2] However, a more efficient and less intrusive approach is to locate and analyze the OpenAPI specification of the API. An OpenAPI specification is a standardized format used to describe the structure and behavior of REST APIs. By successfully parsing the OpenAPI document, malicious users can gain insight into the API's design, which can lead to the identification of potential flaws, such as BOLA.

Although a person can identify BOLA vulnerabilities manually, it becomes challenging, time-consuming, and error-prone to detect across complex APIs. Furthermore, most authorization mechanisms are not specified in the OpenAPI specification, which means that a significant amount of guesswork is required to determine their implementation. Therefore, this thesis proposes utilizing recent advances in LLMs as an automated approach to assist in detecting BOLA vulnerabilities. This approach aims to reduce the need for manual effort and enable security assessments across complex and straightforward APIs. We propose the following research questions:
RQ1: How effectively can LLMs detect BOLA vulnerabilities in REST APIs?
RQ2: Which Large Language Model - Deepseek R1, GPT-o4 mini, Microsoft Copilot AI, Grok 3 performs best at detecting BOLAs?
RQ3: How effectively can LLMs parse and understand OpenAPI specifications to identify BOLAs?

Section 2 provides a more detailed explanation of how a BOLA attack can be performed against a vulnerable web application. The current automated solutions for detecting BOLA vulnerabilities will be discussed in Section 3. Subsequently, Section 4 outlines the methodology and tools employed to gather OpenAPI specifications, design LLM prompts, and discusses the interpretation of results. Section 5 presents a discussion of the results, and Section 6 considers the limitations of the research. Furthermore, Section 7 discusses future work, and finally, Section 8 concludes the study.

2 BACKGROUND: BOLA AND OPENAPI

Object-level authorization is an access control mechanism that is usually implemented at the code level to validate that a user can only access the objects that they should have permission to access. [9] This mechanism must verify the client's permissions before handling any API request that involves objects or receives object identifiers. In the absence of a proper access control mechanism, a malicious user can, in theory, gain access to sensitive data, manipulate it, or completely delete it. Two examples are provided to help clarify how a BOLA attack can be executed precisely.

Furthermore, it is essential to clarify what the OpenAPI specifications are. The OpenAPI Specification (OAS) defines a standard, language-agnostic interface to HTTP APIs, which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. [7] To find BOLA vulnerabilities, it can provide a malicious user with information such as a list of all of the endpoints and their authentication mechanisms. **Example 2** provides an example of the appearance of a vulnerable endpoint in OpenAPI 3.0 format and how it can be exploited.

Example 1 - Unauthorized Data Disclosure
Consider an online clothing webshop where users are first required to create an account, add items to their cart, and then proceed to checkout. A malicious user named Bob logs in to their account and observes that their user ID is "101". Meanwhile, an unaware user, Alice, has already created an account beforehand. Upon inspecting the network traffic or by locating the OpenAPI specification of the

TScIT 43, July 4, 2025, Enschede, The Netherlands

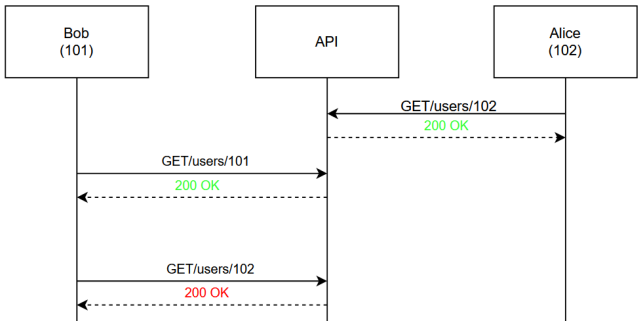

Fig. 1. Successful BOLA attack

webshop, Bob discovers that his and other users' profiles are accessible through */users/{id}*. He then issues a new API request using an API testing tool, such as Postman or Burp Suite, to the vulnerable endpoint and manually modifies the user object ID to '102'. He sends the request *"GET /users/102"*. The request goes through - the server responds with the profile data of another user, Alice. This indicates that the access control mechanism is flawed, and the BOLA attack has been successfully exploited, allowing Bob to gain unauthorized access to another person's data. Figure 2.1 provides a representation of the attack.

Example 2 - Unauthorized Data Deletion
The "Capital" digital goods store enables users to post and discuss the sold articles in the comment section by creating, editing, and deleting comments. In this scenario, the malicious user, Bob, first locates an item, such as an e-book with the ID 'book33', identifies a comment that he dislikes, and discovers it has an ID of '201'. By acquiring the OpenAPI spec (*capital.yaml*), Bob locates the endpoint */articles/slug/comments/commentId*, which has a call option named '*DELETE*'. This endpoint is illustrated in **Listing 1** as part of the API specification. Bob issues an API request, "*DELETE /articles/book33/comments/201*". The server accepts the request without verifying whether Bob was the original author of the comment, and the comment is deleted from the platform. Therefore, Bob has successfully impersonated another user and deleted their comment from the comment section. This confirms the BOLA vulnerability.

```
/articles/{slug}/comments/{commentId}:
  delete:
    tags:
      - Articles, Favorite, Comments
    summary: Delete Comment for Article
    parameters:
      - name: Content-Type
        in: header
        schema:
          type: string
        example: application/json
      - name: X-Requested-With
        in: header
        schema:
          type: string
        example: XMLHttpRequest
      - name: Authorization
        in: header
        schema:
          type: string
        example: Token {{token}}
      - name: slug
        in: path
        schema:
          type: string
        required: true
      - name: commentId
        in: path
        schema:
          type: string
        required: true
    responses:
      '200':
        description: Successful response
        content:
          application/json: {}
```

Listing 1. OpenAPI Snippet for a BOLA-vulnerable Endpoint in capital.yaml

3 RELATED WORK

Automated detection of BOLA vulnerabilities in APIs is an ongoing research area. Traditional security scanners, such as Static and Dynamic Application Security Testing (SAST/DAST), are ineffective at finding BOLAs.[5] As a result, recent work has focused on novel automated techniques that leverage either the OpenAPI specification or use known BOLA features. These techniques fall into three main categories: formal mathematical model approaches, static analysis of the API spec, and LLM-driven BOLA detection and generation.

Firstly, one line of work transforms an OpenAPI specification into a formal model for analysis. Santos Filho et al. (2025) [3] propose Links2CPN, a tool that converts an OpenAPI 3 specification into a Colored Petri Net (CPN) model. The tool then checks conformance between the model and actual API execution logs of the API to detect BOLA vulnerabilities. In the field, Links2CPN achieves high detection accuracy in case studies, reporting an accuracy of over 95 percent. However, the tool has two significant drawbacks. Firstly, it depends on careful OpenAPI documentation, where the *links* feature is present within the specification. This feature describes relationships between API operations, which can be used for modelling workflows; however, it is not always present in OpenAPI specifications. Secondly, the tool relies on dynamic traces; it analyzes server log files, which must be parsed into a JSON event log to perform conformance checking, as it only accepts JSON logs based on the Common Log Format. In summary, formal-model tools like Links2CPN, which leverage mathematical structures to analyze API behavior, can be utilized; however, they require high-quality OpenAPI documents with the *links* feature and access to runtime JSON format logs.

Another approach is a purely static analysis of the API specification. Barbanov et al. (2022) [2] develop a rule-based algorithm that parses an OpenAPI specification to identify potential BOLA vulnerabilities. Their methodology consists of two phases: firstly,

the tool annotates each endpoint with 'BOLA properties' (e.g., identifying which parameters resemble object IDs); secondly, it applies a rule-based 'attack analyzer' to the annotated specification to generate possible attack vectors. The authors of the paper implemented a Python algorithm that successfully identified vulnerabilities in specifications. In practice, the authors note that their analyzer requires trusted, well-annotated specifications, such as correct 'securityScheme' annotations and 'required' flags. Nevertheless, this is among the first to use only an OpenAPI specification to predict BOLA vulnerabilities. It is noteworthy to mention that their tool is not open-sourced, and the authors are impossible to contact, making it unfeasible to recreate their results.

Finally, emerging AI-based methods have already been applied to automate BOLA testing; however, there is a lack of documentation of the results. Most notably, Palo Alto Networks' "BOLABuster" [5] leverages LLMs to interpret OpenAPI specifications and generate test plans. First, it identifies potentially vulnerable endpoints; then, it utilizes the LLM to uncover endpoint dependencies and constructs workflows that demonstrate how endpoints can be accessed. Next, it generates and executes tests against a live API, simulating two users interacting with these endpoints, and analyzes responses for authorization failures. Their tool can also dynamically generate tests from the outputted endpoint dependencies and the specification. Furthermore, the authors have proof of finding BOLAs in real-world services such as Grafana or Harbor. However, their blog post does not entail either the prompt design or the specific LLMs used, and it does not provide any empirical evidence to substantiate their findings.

In addition to these specific tools, other automated API security frameworks exist, such as stateful fuzzers like RESTler or 42Crunch's audit tool. However, they do not specifically target BOLAs and are outside the scope. Moreover, traditional methodologies like fuzzing and static analysis are ineffective in detecting BOLAs, making manual detection the standard approach.[5]

4 METHODOLOGY

4.1 Dataset Collection and Validation

We collected OpenAPI 3.0 specifications, formerly known as Swagger, from open-source projects known to contain BOLA vulnerabilities. The collection primarily consisted of intentionally vulnerable APIs designed to assist people with learning about cybersecurity and testing various tools; however, a real-world vulnerability example is also used. GitHub was our primary source for finding the specifications; we used the built-in search tool to locate JSON and YAML API definition files. For example, the VAmPI[1] vulnerable API provides an openapi3.yml file in its repository. In addition to GitHub, we consulted Google searches and scientific research databases to find other candidates. In total, we obtained 12 distinct API specifications, encompassing 241 unique endpoints. The full dataset is available on our GitHub page. [4] Furthermore, every API in our dataset had to be manually examined and confirmed for BOLA instances. For example, a published writeup on the VAmPI API [2]describes a BOLA in the `/books/v1/{title}` endpoint. Every API was hosted locally to confirm that the BOLA vulnerabilities truly exist. The majority of the APIs included Docker configuration files, which enabled efficient deployment through Docker containers. Postman, together with the imported OpenAPI specification or a Postman Collection, was used to probe specific endpoints, create users, and execute BOLA attacks. Burp Suite Community Edition was a tool also used for issuing API requests and confirming BOLAs. These manual verifications ensured that all 19 cases in our dataset were true positive BOLA vulnerabilities.

4.2 Prompt Engineering

Following practices committed in security-focused research [1] [11], we crafted prompts that specify the task, role, and output format. The final prompt is present in **Appendix A**. We experimented with multiple templates (zero-shot and few-shot) and contents, where the LLM was either given a single piece of context regarding the definition of BOLAs or various pieces of context, along with explicit role instructions and vulnerability examples. The final prompt instructs the LLM to "act as a security analyst." Such role-based and domain-specific prompts have been shown to significantly improve LLM accuracy in security tasks. [1] It asks for a structured JSON response, listing any endpoints susceptible to BOLA. Furthermore, it asks what not to give in the prompt response. Research has shown that providing negative class examples can significantly enhance the model's performance and reduce false negatives [1]. Overall, the prompt offers as much context as possible to the LLM and follows the principles of prompting for security-related tasks by utilising an appropriate expert role and focused content instructions to guide the model's analysis, thereby reducing false negatives and other irrelevant input.

Without proper guidance, the LLMs often gave far too many false positives and other unnecessary information, such as instructions on how to mitigate the vulnerabilities. These issues were present across all models, and proper actions were taken to address them. For example, reducing the false positive rate was addressed by providing the models with two separate examples of BOLA vulnerabilities and 11 specified points on how an endpoint with the issue might appear, for example, "*The parameter is likely to be a UUID, GUID, JWT, session token, email address, or any string with high entropy.*"

What is not present in the final prompt is that other prompts were used to make sure the results of the prompt were presented within reason. To clarify, this meant asking the LLM for specific reasoning behind every outputted endpoint and ensuring it was not an uninformed guess but a thoughtful and justifiable outcome. Additionally, the prompts were enhanced with questions regarding the dependencies of the outputted endpoints and the authorization mechanisms, if applicable. Lastly, the LLMs were tasked with describing the properties of the endpoint in terms of what it was intended to do. The features mentioned above would help in discerning the capability of LLMs in understanding the OpenAPI specifications altogether, therefore answering **RQ2**.

[1]VaMPi: https://github.com/erev0s/VAmPI

[2]Writeup on VAmPI: https://medium.com/@josegpach/vampi-vulnerable-api-a-beginners-guide-to-api-security-testing-ed3b0302eeef

4.3 LLM Selection

We selected four state-of-the-art LLMs for evaluation: Grok 3, Microsoft Copilot, GPT-o4-mini, and Deepseek R1. These models were chosen because they represent leading AI capabilities and are ranked among the top few on LMArena. It is worth mentioning that Copilot AI is based chiefly on GPT-4 variants; however, it is specifically tuned for tasks such as data analysis and code writing. Three additional models were also tested, namely Deepseek R1-7b, Deepseek R1-14b, and LLama 3.1-8b. Although these models contain significantly fewer parameters — less than one-tenth of those in their largest counterparts — they were selected for their feasibility in terms of local runtime on standard desktop hardware.

4.4 Prompting

During the prompting phase, each LLM was given the exact prompt found in **Appendix A** and specification five separate times. The reason why the models were prompted five times is that the outputs of the prompts showed very similar results, and almost four-fifths of the time, the yielded output matched the previous one. The prompting itself was done either through the LLMs' available graphical user interface online or through a Python script in Visual Studio Code. Each prompt was provided independently, meaning no prior context was given to the LLM, and the session was reset to a blank state. That way, no dialogue is carried over from a previous state or conversation. The resulting output was compiled into a comprehensive spreadsheet to gain insights into the results.

4.5 Result Interpretation

To evaluate LLM performance, we computed standard classification metrics from the confusion matrix, consisting of true positive (TP), false positive (FP), true negative (TN), and false negative (FN) rates. The rates mentioned above were labeled accordingly:

- **TP**: The model correctly identified BOLA
- **TN**: The model correctly ignored a safe endpoint
- **FP**: The model incorrectly identified a safe endpoint as BOLA
- **FN**: The model did not flag a BOLA

Accuracy is calculated as the fraction of correctly classified examples: $(TP + TN)/(TP + TN + FP + FN)$. Precision and recall are computed as $TP/(TP + FP)$ and $TP/(TP + FN)$, respectively. Lastly, the F1 score, which is the harmonic mean between the precision and recall, is calculated as $2 \cdot (\text{precision} \cdot \text{recall})/(\text{precision} + \text{recall})$. These metrics enable the measurement of model performance and help identify the best overall model. Using all four metrics also gives a more complete picture of the overall performance. For example, a model could have high accuracy and low recall, meaning it missed many true positives, or high recall and low precision, meaning it overfitted the findings and produced a large number of false positives, or vice versa.

5 RESULTS AND DISCUSSION

5.1 Analysis of Metrics

To address **RQ1**, tables were created for visual and quantitative analysis. **Table 1** reports the comprehensive gathering of the said metrics for each LLM by summing TP, FP, TN, and FN over all of the OpenAPI specifications. Furthermore, the table in **Appendix B** provides a more comprehensive representation of each LLM, displaying the classification metrics for each specification.

Overall, all four models achieved a moderate accuracy (between 0.82 and 0.86. The recall is also relatively high (between 0.75 and 0.83), indicating that the models detected most of the BOLA vulnerabilities. It remained high across most of the API specifications, with 8 out of 12 specs showing a recall of 1 (all of the vulnerabilities were found) for all LLMs. With that said, the precision turned out to be relatively low (between 0.28 and 0.33), meaning that many false positives were flagged. Furthermore, the F1 scores yielded by the research were only moderate at best (between 0.42 and 0.48).

After a finer analysis in **Appendix B**, it becomes clear what causes the precision and F1 scores to plummet. Simple APIs such as *vampi*, *dvws* or *Vulnerable-rest-api* specs typically yielded F1 scores between (0.67-0.8). On the contrary, certain APIs with complex logic, such as *capital* and *crapi*, which also had a large number of endpoints — 34 and 40, respectively — achieved exceptionally low F1 scores, with most of them being approximately 0.25. An outlier with only 17 endpoints and 1 BOLA vulnerability was the *RESTaurant* specification from the Damn Vulnerable RESTaurant Game. The explanation for why this could be the case may relate to the fact that the BOLA vulnerability did not match the prompt, and there was nothing wrong with the specification on the surface. The vulnerability itself was embedded in the application logic, not the OpenAPI specification, meaning that even with sufficient context, it would be challenging for an LLM to identify this vulnerability.

5.2 Reliability of LLM Output

In general, the output produced by the LLMs was satisfactory in the end, although it required some refinement within the prompts. Additionally, the definition of *satisfactory* in the research context meant that the model should output only a list of possibly vulnerable endpoints, and the list should match the instructions provided in the prompt. This was also addressed in Section 4.2 by asking the LLMs for specific reasoning behind each endpoint in their output before conducting the final tests. This also helps with answering **RQ3** regarding the parsing of OpenAPI 3.0 specifications. It became evident early on that the LLMs were fully capable of comprehending the OpenAPI specifications. Firstly, they demonstrated the capability to extract all endpoints within the context of a single specification. Secondly, the models could correctly identify the methods and tasks each endpoint was intended to perform in most cases. Thirdly, the LLMs demonstrated their ability to properly analyze and discern the dependencies within the endpoints, as well as parse the token and authorization mechanisms evident in the specifications.

5.3 Model Comparison

Both tables - **Table 1** and the table in **Appendix B** can be used to analyze how well each LLM performed on its own and to help answer **RQ2**. It becomes evident that the models exhibited very similar performance across all key metrics, with Deepseek R1 outperforming other models by a slight margin. Although it performed as the best overall, the major drawback of its free-to-use model is slow responses and server timeouts. In practice, it would only allow

Table 1. Performance Metrics Comparison

LLM/Metric	TP	FP	TN	FN	Accuracy	Precision	Recall	F1-Score
Deepseek R1	**79**	**158**	**952**	**16**	**0.8556**	**0.3333**	**0.8316**	**0.4759**
Grok 3	75	188	922	20	0.8274	0.28517	0.7895	0.419
GPT o4-mini	71	170	940	24	0.839	0.2946	0.7474	0.4226
Copilot AI	75	192	918	20	0.824	0.2809	0.7895	0.4144

for a few inputs before timing out, at which point it was necessary to use the model through a different account. The server response time was deliberately degraded for the user, and it became quite cumbersome to interact with after a while.

Furthermore, as mentioned in the subsection above, all the models demonstrated the capability to reason with an understanding of the OpenAPI specification and with the prompt. However, no specific outperformer can be identified from this conclusion, as it would become quite strenuous to compare the reasoning behind the thought processes of the models.

5.4 Smaller Models

We want to address the three additional models (Deepseek R1 7b/14b and LLama 3.1 8b) that were tested. As highlighted in **Section 4.3**, these models were run on a local machine where the OpenAPI specification, along with the prompt, was provided in plain text format through a Python script. The minor 7b and 8b models responded in approximately 30 seconds, while the 14b model took around a minute to process, which made the process quite time-consuming. Despite exhaustive testing (each model was tested around 30 times with different prompt structures), none of them were able to obtain any relevant or meaningful results. The models were unable to understand the given problem. Despite all the context provided, the models demonstrated a fundamental lack of understanding of what a BOLA attack entails. They often hallucinated by outputting unrelated data, for example, security advice on how to avoid SQL injection and cross-site scripting (XSS) attacks. Taking into account these limitations, the outputs from these models were deemed to be too unreliable and were excluded from the quantitative analysis of different LLMs.

5.5 General Implications

The results suggest that LLMs can reason about API logic and classical BOLA flaws to identify potentially vulnerable endpoints by analyzing patterns and structure and being effectively guided by carefully engineered prompts. Furthermore, the findings suggest that LLMs possess a strong ability to reason when presented with an appropriate and context-rich prompt. In practice, such automated analysis could be incorporated into tools used by professionals, such as security specialists, by utilising LLM-based scanners during system reviews. Another possibility for LLM-based detection is to complement existing BOLA detection methods, such as formal analysis [3], which is constrained by logging but achieves very high accuracy. However, the lack of F-1 scores and precision suggests that the models overgeneralized the given rules, which is reinforced by every LLM suffering from similar issues. This suggests that fully automatic LLM-based tools should not be used as the sole remedy for automatic BOLA detection, and that further manual labour or intervention is required.

6 LIMITATIONS

Firstly, we acknowledge that we were unable to quantitatively compare the output from the experiments with that of some related methods. For example, the Links2CPN [3] tool requires JSON-formatted server logs as one of the inputs. Unfortunately, the GitHub projects the team took for the OpenAPI specifications did not include such logging methods. The only possibility would be to implement such logging techniques manually into the projects; however, this would be too time-consuming since the codebases have different structures and implementation methods. Likewise, the tool developed by Barabanov et al. (2022) [2] was not publicly accessible, and the contacts they had provided in the paper were unreachable.

Secondly, more LLMs could have been tested for a more comprehensive comparison, for example, Claude Opus 4 or GPT-o3, which are only available for PRO subscriptions or are paid by the number of tokens. As the prompts used in the research are quite significant (around 2000-4000 tokens per prompt), the costs would be as well. Furthermore, local models such as LLama 3.1 70b, qwen3-235b, or Deepseek R1-71b could not be tested due to hardware constraints, as approximately 168 GB of GPU memory would be required to serve the LLaMA model with 70 billion parameters.[6] In contrast, the test setup used by the team had only 12 GB of GPU memory.

Thirdly, the number of documented and available OpenAPI specifications is limited, which constrains our dataset. While massive datasets of OpenAPI 3.0 specifications already exist, those with known BOLAs are scarce.

7 FUTURE WORK

The following key step addresses the limitations identified in the study. First, a tool needs to be found or developed for reliable benchmarking against LLMs and other methods concerning BOLAs. Second, a more exploratory analysis of model performance would require a broader range of LLMs, including not only commercial but also open-source types that can give reliable output. Lastly, the dataset of OpenAPI specifications with BOLAs could be enlarged, enabling even more generalizable conclusions regarding LLM effectiveness.

In addition to addressing the limitations, other avenues could be explored to approach this research. For example, fine-tuning local, open-source models could produce more reliable results. This could be done by specifically training the models on BOLA-related data and changing the input parameters of the models themselves. Furthermore, LLM-dependent prompts yield higher metric scores by adjusting the prompts based on the LLMs.

Another option to enhance the research would be to explore more advanced reasoning techniques, such as chain-of-thought reasoning, which can assist the model in reasoning with greater accuracy by generating intermediate reasoning steps.

Lastly, a fully automated pipeline for test generation and vulnerability confirmation could be explored using LLMs, leveraging the metrics extracted from the research to further establish the applicability of LLMs. This could mean detecting vulnerabilities with no human intervention whatsoever.

8 CONCLUSION

This study aimed to investigate the ability of LLMs to detect BOLA vulnerabilities in REST APIs by utilizing their OpenAPI specifications. The research intended to assess the overall effectiveness of models, compare their performance, and evaluate their capability to parse OpenAPI specifications. At the beginning of the study, three research questions were addressed:

RQ1: How effectively can LLMs detect BOLA vulnerabilities in REST APIs?

Overall, LLMs were shown to be partially effective. They achieved high accuracy (0.82-0.86) and recall (0.75-0.83), identifying all BOLA vulnerabilities in 66 percent of the cases. However, their precision was low (0.28-0.33), meaning that many false positives were reported, especially in more complex specifications. This limits their overall reliability but suggests potential as a supportive tool in BOLA detection.

RQ2: Which Large Language Model - Deepseek R1, GPT-o4 mini, Microsoft Copilot AI, Grok 3 performs best at detecting BOLAs?

Among the tested models, Deepseek R1 performed the best across most metrics. Although the rest of the competition achieved very similar results, with the differences being marginal, Deepseek consistently outperformed them.

RQ3: How effectively can LLMs parse and understand OpenAPI specifications to identify BOLAs?

Larger LLMs demonstrated reasonable capability in parsing OpenAPI specifications and identifying potentially vulnerable endpoints. They consistently produced the correct type of data as the output, with no hallucinations or errors in understanding the task at hand. However, smaller models such as Deepseek R1-7b and Llama 3.1-8b showed a lack of fundamental understanding of the assignment.

In conclusion, LLMs demonstrate significant promise for the automated detection of BOLA vulnerabilities in APIs, particularly when reviewing well-documented and straightforward API specifications. Precision issues in large APIs and the limited ability to reason over complex logic are among the most critical unresolved problems, indicating the need for human intervention. Further research into LLMs, model fine-tuning, and prompt engineering enhancements may mitigate such limitations and improve the overall real-world applicability of LLMs to API security analysis.

REFERENCES

[1] Weiheng Bai, Qiushi Wu, Kefu Wu, and Kangjie Lu. 2024. Exploring the Influence of Prompts in LLMs for Security-Related Tasks. https://www.ndss-symposium.org/wp-content/uploads/aiscc2024-15-paper.pdf

[2] A. Barabanov, D. Dergunov, D. Makrushin, and A. Teplov. 2022. Automatic detection of access control vulnerabilities via API specification processing. *Voprosy Kiberbezopasnosti* 1, 47 (Jan. 2022), 49–65. https://arxiv.org/abs/2201.10833

[3] A. S. Filho, R. J. Rodriguez, and E. L. Feitosa. 2025. Automated broken object-level authorization attack detection in REST APIs through OpenAPI to Colored Petri nets transformation. *International Journal of Information Security* 24 (Feb. 2025). https://link.springer.com/article/10.1007/s10207-024-00970-5

[4] Emils Johansens. 2025. Vulnerable OpenAPI specifications with BOLA. https://github.com/ejohansens/Vulnerable-APIs

[5] R. Mazon and J. Chen. 2024. Harnessing LLMs for automating BOLA detection. https://unit42.paloaltonetworks.com/automated-bola-detection-and-ai/. Unit 42, Palo Alto Networks.

[6] Medium. 2024. How Much GPU Memory is Needed to Serve a Large Language Model (LLM)? https://masteringllm.medium.com/how-much-gpu-memory-is-needed-to-serve-a-large-languagemodel-llm-b1899bb2ab5d.

[7] OpenAPI Initiative. 2023. OpenAPI Specification. https://swagger.io/specification/. https://swagger.io/specification/ Version 3.1.0.

[8] OWASP. 2023. API Security Top 10 – 2023 Edition. https://owasp.org/API-Security/editions/2023/en/0x11-t10/.

[9] OWASP. 2023. Broken Object Level Authorization (BOLA) – 2023 Edition. https://owasp.org/API-Security/editions/2023/en/0xa1-broken-object-level-authorization/.

[10] OWASP. 2023. Modern API Security Frameworks for Enterprise Systems. https://www.researchgate.net/publication/389380711_Modern_API_Security_Frameworks_for_Enterprise_Systems.

[11] Xinyu Wang, Yuxian Gu, Ruoxi Sun, Zihang Song, Haoyu Zhai, Fan Nie, Xiang Ren, Cho-Jui Hsieh, and Sijia Liu. 2025. Everything You Wanted to Know About LLM-based Vulnerability Detection But Were Afraid to Ask. https://arxiv.org/abs/2504.13474

A APPENDIX A - FULL PROMPT

You must act as a security analyst looking for BOLA (Broken Object Level Authorization) vulnerabilities within an OpenAPI 3.0 specification. You will be given an OpenAPI 3 specification. Your task is to parse it and identify potentially vulnerable endpoints based on patterns commonly associated with BOLA flaws.

Here are a few examples of how a BOLA vulnerability might look like:

(1) Accessing another user's data:
For example, an API endpoint might look like /users/{userId}, where {userId} is a path parameter. If the API does not verify that the user making the request is authorized to access the specific userId, a malicious user could change the {userId} value to retrieve another user's data.
(2) Manipulating resources:
Another example is when an API endpoint like /orders/{orderId} allows users to view or update their orders, but does not enforce that the user is the creator or owner of the order. A malicious user could access or manipulate another user's order by tampering with the {orderId} path parameter.

Identify and return all endpoints that could have BOLA vulnerabilities and that meet *any* of the following criteria:

(1) The endpoint includes a path or query parameter whose name or description suggests it identifies a user, session, organization, or resource (e.g., userId, accountId, orgId, orderId, postId, token).
(2) The parameter is likely to be a UUID, GUID, JWT, session token, email address, or any string with high entropy.
(3) The endpoint includes a path or query parameter whose name matches identifier patterns: Ends with id, ID, _id, Id, uuid, UUID, guid, GUID, is exactly id, uuid, guid, token, key OR matches known identifier terms: name, email, phone, group, account, user, project, team, tenant, bucket, session, profile, etc.
(4) The parameter may reference a group of users or resources (e.g., teamId, projectId, tenantId).
(5) The parameter appears in HTTP methods PUT, PATCH, DELETE, or GET (indicating data access or mutation).
(6) The OpenAPI spec lacks explicit authorization mechanisms on the operation (e.g., missing security requirements or 403 response codes).
(7) The parameter is reused across multiple endpoints, suggesting object reuse or shared access scope.
(8) Parameter is in the path and the path contains a matching noun prefix (e.g., /orders/{order} where order and orders align), or, the path pattern contains action verbs and resource-like identifiers (e.g., /profileInfo/edit/{profile} or /{collection}/action/{item}).
(9) The parameter value corresponds to a resource created or returned in another endpoint (e.g., /files creates fileId, used in /files/{fileId}).
(10) The 'tags' parameter has words like UUID, ID, GUID, id.
(11) The HTTP method is one that accesses or modifies data (GET, PUT, PATCH, DELETE).

Only return a plain list of the endpoint paths that match the criteria. Do NOT include any explanations, formatting, or reasoning. Output ONLY the list of endpoints.

Do not include any explanation, description, extra text, or any markdown formatting.

Do not include any other vulnerabilities, such as SQL injection, XSS, etc.

Only focus on BOLA vulnerabilities (Broken Object Level Authorization), not other vulnerabilities such as Broken Object Property Level Authorization, Broken Access Control, or Broken Authentication.

Example output:
["/api/v1/patients/{patientId}", "/api/v1/medical-records/{recordId}", "/api/v1/prescriptions/{prescriptionId}"]

Here is the OpenAPI 3 specification:(openapi3.yaml)

B APPENDIX B

Table 2. Performance Metrics Comparison between specifications

Specification/LLM		Deepseek R1	Grok 3	Gpt 4o-mini	Copilot AI
Vapi	Accuracy	0.9	**0.9545**	**0.9545**	**0.9545**
	Precision	0.3125	**0.5**	**0.5**	**0.5**
	Recall	**1**	**1**	**1**	**1**
	F1	0.4762	**0.6667**	**0.6667**	**0.6667**
Capital	Accuracy	**0.8606**	0.8182	0.8424	0.8121
	Precision	**0.1785**	0.14286	0.1613	0.1389
	Recall	**1**	**1**	**1**	**1**
	F1	**0.303**	0.25	0.2778	0.2439
Vuln-Bank	Accuracy	0.7	0.625	**0.9167**	0.6333
	Precision	0.12195	0.1	**0.3333**	0.10204
	Recall	**1**	**1**	**1**	**1**
	F1	0.2174	0.1818	**0.5**	0.1852
dvapi	Accuracy	**0.9286**	0.8571	**0.9286**	0.7571
	Precision	**0.5**	0.3333	**0.5**	0.22727
	Recall	**1**	**1**	**1**	**1**
	F1	**0.6667**	0.5	**0.6667**	0.3704
crapi	Accuracy	**0.9063**	0.8421	0.8256	0.8462
	Precision	**0.34615**	0.25	0.14706	0.16667
	Recall	0.9	**1**	0.5	0.5
	F1	**0.5**	0.4	0.22727	0.25
vampi	Accuracy	**0.8571**	**0.8571**	**0.8571**	**0.8571**
	Precision	**1**	**1**	**1**	**1**
	Recall	**0.6667**	**0.6667**	**0.6667**	**0.6667**
	F1	**0.8**	**0.8**	**0.8**	**0.8**
RESTaurant	Accuracy	**0.8333**	0.8235	**0.8333**	0.8235
	Precision	**0.5**	0	0.0909	0
	Recall	**0.3333**	0	0.2	0
	F1	**0.4**	0	0.125	0
Vulnerable-rest-api	Accuracy	0.88	**0.9**	0.8182	**0.9**
	Precision	0.625	**0.6667**	0.5	**0.6667**
	Recall	**1**	**1**	0.5	**1**
	F1	0.7692	**0.8**	0.5	**0.8**
dvws	Accuracy	0.9167	**0.95**	0.9167	0.9167
	Precision	0.5	**0.625**	0.5	0.5
	Recall	1	**1**	1	1
	F1	0.6667	**0.7692**	0.6667	0.6667
OWASP Juice Shop V1	Accuracy	**1**	**1**	**1**	**1**
	Precision	**1**	**1**	**1**	**1**
	Recall	**1**	**1**	**1**	**1**
	F1	**1**	**1**	**1**	**1**
OWASP Juice Shop V2	Accuracy	0.5	0.5	**1**	**1**
	Precision	0	0	**1**	**1**
	Recall	0	0	**1**	**1**
	F1	0	0	**1**	**1**
Memos	Accuracy	0.7368	0.7368	**0.7632**	0.7368
	Precision	0.0909	0.0909	**0.1**	0.0909
	Recall	**1**	**1**	**1**	**1**
	F1	0.16667	0.16667	**0.1818**	0.16667