WENDIGO: Deep Reinforcement Learning for Denial-of-Service Query Discovery in GraphQL

Shae McFadden[†‡], Marcello Maugeri[⋆†], Chris Hicks[‡], Vasilios Mavroudis[‡], Fabio Pierazzi[†]
[†]*King's College London,* [‡]*The Alan Turing Institute,* [⋆]*University of Catania*

***Abstract*—GraphQL is a type of web API which enables a unified endpoint for an application's resources through its own query language, and is widely adopted by companies such as Meta, GitHub, X, and PayPal. The query-based structure of GraphQL is designed to reduce the over-/under-fetching typical of REST web APIs. Consequently, GraphQL allows attackers to perform Denial-of-Service (DoS) attacks through queries inducing higher server loads with fewer requests. However, with the additional complexity introduced by GraphQL, ensuring applications are not vulnerable to DoS is not trivial.**

We propose WENDIGO, a black-box Deep Reinforcement Learning (DRL) approach only requiring the GraphQL schema to discover DoS exploitable queries against target applications. For example, our approach is able to discover queries which can perform a DoS attack utilizing only two GraphQL requests per hour, as opposed to the high volume of traffic required by traditional DoS attacks. WENDIGO achieves this by building increasingly more complex queries while maximizing response time by using GraphQL features to increase the server load. The effective query discovery offered by WENDIGO, not only enables developers to test for potential DoS risk in their GraphQL applications but also showcases DRL's value in security problems such as this one.

***Index Terms*—Reinforcement Learning, Deep Neural Network, Denial-of-Service Attack, Internet Security, Web Security**

1. Introduction

GraphQL is an enticing alternative to REST APIs as it provides a unified graph abstraction to an application's web resources. GraphQL shows a widespread adoption across industry with institutions such as Meta, GitHub, X, and PayPal adopting this API [1], alongside the 33.4% of surveyed API developers who planned to utilize GraphQL in 2020 [2]. All the while, DoS attacks are a recurrent issue across multiple domains [3]. The traditional paradigm for DoS attacks involves flooding the target system with high volumes of traffic and protocol exploitation to overwhelm its resources [3]. However, as a result of the powerful nature of GraphQL, complex operations can replace high traffic volumes, making DoS a serious threat to GraphQL applications [4]. An empirical study exemplifies this stating that most commercial and large open-source GraphQL APIs may be susceptible to DoS [5]. The aforementioned study's claims are further supported by a 2022 survey stating that of the respondents only 21.7% use timeouts, 23.1% use rate limiting, 26.7% use depth limiting, and 17.5% use cost analysis [6]. With this potential risk in GraphQL, it becomes crucial that developers evaluate whether maliciously crafted queries can cause DoS. To perform this evaluation, as a result of the complex and expansive nature of the potential query space, an automated approach is necessary to search for malicious queries.

To aid in this security testing problem, we propose WENDIGO: a black-box DRL agent-based approach to discover GraphQL queries which can be leveraged to perform a DoS attack.[1] WENDIGO consists of two key components: the *environment* and the *agent*. The environment converts the GraphQL schema into the *state* and *action* spaces. The state space is a recursive expansion of the schema using GraphQL features which increase server load, thereby creating a representation of a bounded subset of the potential query space, therefore the states are representations of potential DoS queries. The action space mirrors that of the state space however it contains add and remove actions for each component of the state space. The agent chooses an action to modify the current query and receives the response time of that modified query as a reward. Through these components, WENDIGO not only discovers queries which induce large response times, but it is also able to build on these queries further maximizing their resource usage.

To the best of our knowledge, we are the first to utilize DRL to discover DoS queries in the GraphQL domain. Evolutionary algorithms have been used for white-box fuzzing of GraphQL applications, focusing on search-based testing and not the discovery of DoS queries [7]. Although they may be effective at finding bugs, we demonstrate that they are insufficient at discovering the GraphQL specific DoS risk in an application. Other research discusses the risk of DoS for GraphQL applications by utilizing specific handcrafted queries [3, 4, 5]. However, our approach eliminates the need to handcraft samples for applications. Machine learning (ML) based approaches have also been used to great success in other security testing tasks [8, 9, 10], with DRL specifically being used to search for adversarial inputs in other domains such as SQL injection [11] and cross-site

1. WENDIGO gets its name from the Algonquian legend about a malevolent spirit whose hunger and size grows the more it consumes which is analogous to how our approach grows queries the more server resources are consumed.

scripting [12]. Although these approaches cannot be directly applied as a result of the differences in domains, their past successes motivate our use of DRL for the task of DoS query discovery in GraphQL.

In summary, our main contributions are:

- We propose WENDIGO, an approach to effectively search for queries which could be exploited for DoS attacks in the GraphQL domain. As a part of this approach, we design a gymnasium environment to encapsulate GraphQL applications for DRL.
- We show that WENDIGO not only outperforms both of random-search baselines and a state-of-the-art fuzzer, EvoMaster [7], but also able to discover queries that can be exploited to cause severe delays.
- We open-source the WENDIGO code to encourage future research, and accelerate security testing of GraphQL applications: https://github.com/isneslab/Wendigo.

2. Background

This section first discusses DRL, i.e., the method that guides our discovery, and then EvoMaster [7], the fuzzer we used as our baseline. Subsequently, we present the basic concepts of GraphQL, the specific features of GraphQL used in our approach and the connection to DoS attacks.

Deep Reinforcement Learning. DRL is a type of ML in which an agent learns through interacting with an environment [13]. The three key components of a DRL problem are the: states, actions, and rewards. From a state, the agent decides on an action that results in a reward and a new state. DRL utilizes a neural network to either: determine the value of an action from a state (value-based), determine a policy (policy-gradient), or a combination of both (actor-critic) [13]. Deep Q-Network (DQN) [14], a value-based method, is often used when DRL is applied to security problems [11, 12, 15, 16, 17, 18]. However, actor-critic methods often outperform value-based or policy-gradient methods in other domains [13, 19, 20, 21].

EvoMaster. EvoMaster [10] was initially crafted as an evolutionary white-box testing tool tailored for RESTful APIs, relying on their schema provided in the OpenAPI specification format [22]. Recent developments have extended its capabilities to encompass testing for GraphQL APIs [7]. In real-world scenarios, accessing the code for white-box testing may not always be feasible. Fortunately, EvoMaster realises a black-box approach that employs a Random Search [23] on syntactically valid queries to explore the input space [7].

GraphQL. GraphQL is a query language for APIs which provides flexibility in retrieving data through a single-request declarative approach, thereby addressing both under-fetching and over-fetching problems in REST APIs. For example, consider the scenario in which a client requires a user's e-mail and the content of their posts. In REST, this would require a request to access the path *users/*{*id*}, retrieving all information regarding the user, resulting in over-fetching. However, only the first request is insufficient to retrieve the user's posts, thereby causing under-fetching. Therefore, a second access to the path *users/*{*id*}*/posts* is required to retrieve the content of the posts. In contrast, GraphQL requires a single request, called a query, to retrieve the precise data needed. An example of such a query would be:

```
query {
  users (id:{id}) {
    email
    posts{
      content
    }
  }
}
```

In a query, objects represent entities, such as users or posts, that consist of fields. Fields within an object are the individual pieces of data or references to objects. In addition, arguments pass on information for the resolution of a specific field, and aliases allow renaming fields in a query for clarity or to prevent naming conflicts. This structure is defined in the schema.

DoS Vulnerabilities in GraphQL. When the aforementioned concepts of GraphQL are used improperly, they can introduce unique attack vectors for DoS. We utilize the DoS attack vectors derived from standard GraphQL features, proposed in [24], as the set of capabilities used to derive the state and action spaces of our approach. The attack vectors are as follows:

Circular Objects are a result of a circular relationship in object references resulting in a potentially infinite amount of recursion in the query request. Querying circular objects increases server load by forcing the server to recursively retrieve objects to fulfill the request.

Field Duplication is when fields in a query are repeated so that when the server resolves the query it has to repeatedly resolve the same field thereby increasing server load.

Alias Overloading is similar to field duplication however it uses aliases to repeat fields under different names and by default there are no limits on the number of aliases.

Object Limit Overriding results when servers allow the number of objects returned from a specific query to be specified by an argument, therefore the limit can be increased to increase the load on the server.

Array-Based Query Batching allows multiple queries to be sent in a single request thereby reducing the volume of requests required to perform a DoS attack.

Denial-of-Service. Traditionally, DoS attacks and Distributed DoS (DDoS) attacks, utilize high volumes of traffic to flood networks to impact the availability of an application [25, 26]. However, DoS attacks do not require high traffic volumes, Low-rate DoS (LDoS) attacks such as *shrew* account for only 10-20% of normal traffic [27]. LDoS attacks achieve DoS through the use of periodic pulses instead of a high volume of traffic while maintaining good attack performance compared to DoS or DDoS [25]. In the case of DoS in GraphQL, we extend the intuition behind

the aforementioned LDoS attack however instead of traffic pulses we periodically use complex queries crafted from the aforementioned attack vectors.

3. WENDIGO

We propose WENDIGO, a black-box DRL agent-based approach for DoS exploitable query discovery in GraphQL applications. We first discuss the motivation behind a black-box approach. Then, we present the environment which consists of the state, action, and rewards utilized to encapsulate interactions with a GraphQL application. Subsequently, we discuss the design of the agent and random search approaches. A flow chart showing the connection between the concepts described in this section is presented in Figure 1.

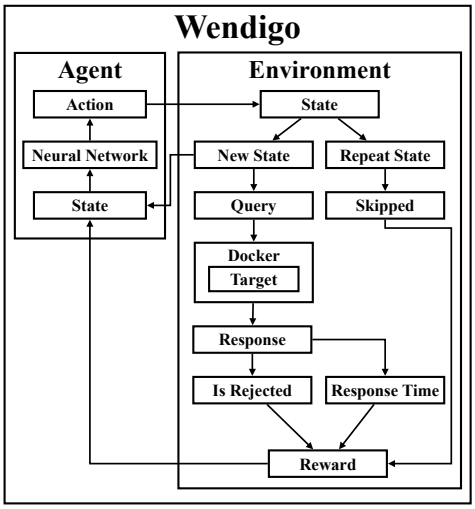


Figure 1: Flowchart of WENDIGO query discovery.

Black-Box. The motivation behind the design of a black-box approach is that it makes a more realistic threat model, utilizing only the URL and schema which would be available to a realistic adversary as the schema is often publicly available or can be reconstructed. Furthermore, black-box approaches can make security testing accessible to product developers who otherwise lack deep security expertise. WENDIGO does not utilize any internal information from, or code of, the target application and only relies on the response time and queries generated. Our experiments and released code utilize Docker containers instead of a URL to ensure evaluations are conducted safely in a controlled environment.

Environment. The environment was designed following the specifications of Gymnasium [28], a standardized agent-environment communication API, allowing compatibility between different agents and environments. The state, action, and rewards of the environment are defined as follows.

State. We design the current state to be a representation of the current query, therefore the state transitions imposed by actions reflect query modifications. To represent the GraphQL query space as the state space, we use the settings of max_depth and max_height to bound the state space. max_depth represents the maximum recursion when constructing the state space from the query space, and max_height represents the maximum of a single value in the state space. In the state space, each valid field/object location has values for *field duplication* and *alias overloading*. Additionally, objects have values for *object limit overriding* arguments, with *circular objects* being implicitly included by the recursive construction of the state space. Finally, *Array-based query batching* has its own value to specify the batch size for queries; if not specified, only a single query is sent. An example of the state to query conversion, pruned to only present values for readability, is as follows:

```
State: [1, 2, 1, 2]

State-to-Query Mapping:
 [users-DUP, users_email-DUP,
  users_posts-DUP,
  users_posts_content-ALIAS]

Query: query {
          users { Depth = 1 & Height = 1
              email |  Height = 2
              email |
              posts {  Depth = 2 & Height = 1
                  C1:content | Height = 2
                  C2:content |
```

Actions. The action space mirrors that of the state space, however, it has an *add*, and a *remove* action for each value in the state space. These actions result in the increase or decrease of the value at a location in the state. The $multiplier$ setting is the quantity by which a value is increased or decreased when a action is performed. *add* enables the agent to expand or further exploit a fruitful section of the query. Conversely, *remove* allows the agent to remove or reduce a disadvantageous section of the query. If a performed action is invalid in the current state then no modification occurs and the query is skipped.

Rewards. The reward for a state-action pair is the response time of the application in seconds. This implicitly handles rejected queries as a result of their marginal response times. If the state-action pair does not produce a new query then the query is skipped and a response time of zero seconds is returned.

Agents. The agent decides the action to take given the current state. The DRL algorithm chosen for our approach is Proximal Policy Optimization (PPO) [29], an actor-critic method, utilizing the original settings and implementation of the algorithm from *cleanrl* [30]. PPO was chosen as it shows superior performance and training time when compared other algorithms in [29].

We design two random search-based techniques in our environment to show the value of an agent-based approach. First, *Random* uses random action selection and secondly, *Random-Greedy* uses a random action selection which only updates the current state if the new response time is greater than the state's response time. We utilize random search for

these approaches following the lead of EvoMaster's black-box setting as mentioned in Section 2.

4. Experimental Settings

In this section we present our target, baseline, the two settings used for query discovery, and our DoS attack design used to show the value of our discovery.

Target. For evaluation safety we use a docker container of the target application. We utilize *Damn Vulnerable GraphQL Application* (*DVGA*) [31], as a proof of concept to show the potential of our approach on a vulnerable application. We do not evaluate WENDIGO on real applications in this initial work because we want to ensure the evaluation is conducted safely in cooperation with the developers.

Baselines. The motivation behind our selection of a baseline from existing approaches is twofold. Firstly, to the best of our knowledge, there are no other approaches designed to discover DoS queries in the GraphQL domain, barring simple test suites which evaluate only the presence of specific attack vectors. Secondly, without alternatives, developers would have to rely on state-of-the-art fuzzers. Therefore utilizing a fuzzer as a baseline evaluates if a GraphQL fuzzer is sufficient to effectively search for DoS queries in an application. For the aforementioned reasons we have decided to utilize ***EvoMaster***, a state-of-the-art GraphQL fuzzer, as our main baseline.

Additionally, we also utilize ***Random*** and ***Random-Greedy*** in our environment as additional baselines. For all evaluations the two random agents use the same random seed value therefore the only difference is the aforementioned state transition criteria in Section 3.

Settings. We consider two settings for evaluating query discovery: UNPROTECTED and PROTECTED.

The UNPROTECTED setting for query discovery evaluates an application with no mitigations in place to protect against DoS. Although we are not enforcing protections through the use of constraints in this case, for practical reasons the trained and random agents' state and action spaces require bounds. Therefore, we utilize the settings $max\_depth = 10$, and $max\_height = 100$. We utilize $multiplier = 10$ to speed up the query discovery since we have a large bound on max_height. Additionally, we set *EvoMaster*'s $max\_depth = 10$. For the UNPROTECTED setting query discovery, we present the first 384 steps (3 PPO updates) as the agent sufficiently exploits the application in this time frame.

The PROTECTED setting for query discovery evaluates an application with depth and field height limiting for DoS mitigations to reduce the max complexity of requested queries. To model this basic protection we utilize the built-in constraints of our environment. Therefore, we set the trained and random agents to $max\_depth = 5$, and $max\_height = 5$. We use a $multiplier = 1$ so that the approaches do not rapidly over-step the constraints. Additionally, we set *EvoMaster*'s $max\_depth = 5$. For the PROTECTED setting query discovery, we present the first $1,280$ steps (10 PPO updates).

DoS Attack Design. We take the highest response time (HRT) queries from both of the aforementioned settings and utilize them to perform a DoS attack on the application. We conduct this attack by sending the query to the application and waiting for the full response before re-sending the query. This results in pulses of activity by the attacker, akin to that described by the LDoS attack in Section 2. While the attacker is sending its queries we have a benign user sending a basic query in the same fashion. The benign query has an average response time, over $100,000$ queries, of 0.00783 seconds when the application is not under attack. We send queries from both the attacker and benign user for an hour ($3,600$ seconds) and wait until the last query from both agents is received. We then use the number of attacker queries required to successfully perform the DoS to compare the attack efficacy across settings and approaches.

Metric 4.1 ($\%_{Denial}$)**.** To calibrate the DoS attack to ensure a fair comparison, we use the percentage of benign user response time which is a result of the attack. Formally, we define $\%_{Denial}$ as follows:

$$\%_{Denial} = \frac{\sum(\hat{\tau} - \bar{\tau})}{\sum(\tau)} \tag{1}$$

Where: $\bar{\tau}$ is the mean of the pre-attack response times for benign queries; σ is the standard deviation of the pre-attack response times for benign queries; B is the set of the benign user's response times during the attack, with $\tau \in \mathrm{B}$; Impacted response times $\hat{\tau} \in \{\beta | \beta \in \mathrm{B}, \beta > (\bar{\tau} + 2\sigma)\}$ uses threshold $\bar{\tau} + 2\sigma$ to determine the out of distribution response times with 2σ to reduce the threshold's sensitivity to noise.

5. Results

This section presents the results for our evaluation of WENDIGO on *DVGA*.

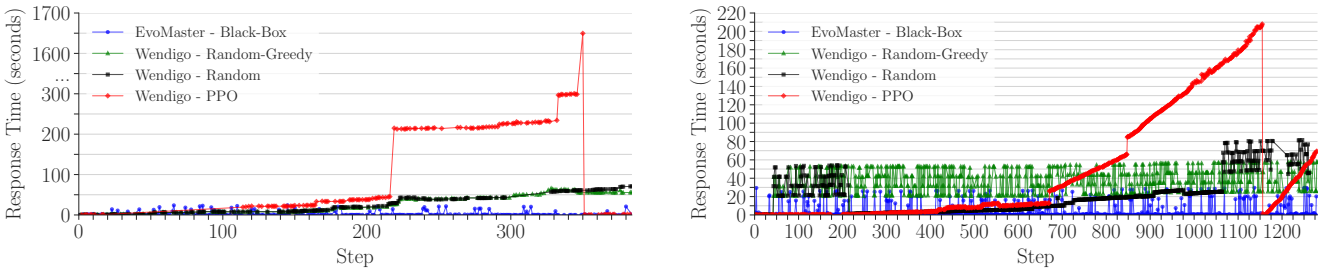
For readability: seconds have been abbreviated to 's'; rejected and skipped queries have been excluded from Figure 2, but can be found in Table 1.

UNPROTECTED Setting. The UNPROTECTED results are presented in Figure 2a and consist of four separate runs: *PPO*, *Random*, *Random-Greedy* and *EvoMaster*.

Random and *Random-Greedy* have comparable performance under this setting as a result of their shared selection strategy. *EvoMaster* achieves a maximum response time of 23.96s throughout the evaluation, however, only achieves these response times sporadically.

Finally, the *PPO* agent is able to surpass both the random agents in 114 steps as well as *EvoMaster* in 144 steps. At 350 steps the agent achieves a response time of $1,649.57$s before the application docker is terminated because of memory over-consumption and then the environment and state are restarted to continue evaluation.

For the UNPROTECTED setting, the query with the HRT of all queries is produced by *PPO* and induces a response time of $1,649.57$s. When we utilize this query to perform the attack described in Section 4, we get an average response



(a) Results for the UNPROTECTED setting: $max\_depth = 10$, $max\_height = 100$, and $multipler = 10$.

(b) Results for the PROTECTED setting: $max\_depth = 5$, $max\_height = 5$, and $multipler = 1$.

Figure 2: The results of the UNPROTECTED and PROTECTED settings on *PPO*, *Random-Greedy*, *Random*, and *EvoMaster*. The plots present the response time y of a query at step x. Note the different scales of the y-axis for readability.

time of $1,824.18$s over $3,648.36$s for the attacker and 456.00s over $3,648.04$s for the benign user. The attacker is able to successfully perform a DoS with a $99.998\%_{Denial}$ utilizing only 2 queries, the least amount of attack queries required for this setting. The remainder of the attack results for the UNPROTECTED setting are presented in Table 1.

PROTECTED Setting. The PROTECTED results are presented in Figure 2b and consist of four separate runs: *PPO*, *Random*, *Random-Greedy* and *EvoMaster*.

Random performs comparable to *Random-Greedy* for the first 215 steps. However, subsequently drops off in performance as a result of randomly removing valuable components of the query. *Random-Greedy* stays within the 20.64s to 57.77s range throughout the evaluation as a result of maintaining sub-optimal states as its baseline. After *Random*'s initial decline, it is able to slowly increase the response time, eventually surpassing *Random-Greedy*. As a result of *Random* not being tied to a baseline state, it can explore more freely. *EvoMaster* achieves a maximum response time of 29.68s throughout the evaluation. However, it finds such response times sporadically and not consistently like our methods.

Finally, the *PPO* agent takes 650 steps to surpass *EvoMaster* and 681 steps to surpass both of the random agents. *PPO* continues to increase the response time up to 208s at $1,155$ step in which the application docker is terminated because of memory over-consumption and then the environment is restarted to continue evaluation. In comparison, it took the UNPROTECTED *PPO* agent only 219 steps to achieve a 208s response time, thereby showing the impact of constraints on our agents.

For the PROTECTED setting, the query with the HRT of all queries is produced by *PPO* and induces a response time of 208s. When we utilize this query to perform the attack described in Section 4, we get an average response time of 20.28s over $3,609.65$s for the attacker and 5.49s over $3,603.81$s for the benign user. The attacker is able to successfully perform a DoS with a $99.852\%_{Denial}$ utilizing 178 queries, the least amount of attack queries required for this setting. The remainder of the attack results for the UNPROTECTED setting are presented in Table 1.

Temporal Vulnerabilities. The reason for the deviation in response time, between the query discovery and DoS attack of *PPO* with the PROTECTED setting, is a result of the audit field which requests the application's logs. Therefore, the time to process the audit field is dependent on the size of the application's logs which grow throughout the execution of the experiment. Although the audit field is less effective initially on a clean application, our agent was able to learn to exploit it through the temporal nature of the discovery.

This shows the value of our approach which utilizes no preconceived knowledge about the application besides the schema and learns through interacting with and exploiting an application.

6. Discussion

Importance of an Agent. The advantage of utilizing an agent-based approach is evident when comparing across the results. The *Random* approach suffers from a lack of information resulting from the completely random action selection, thereby discovering good states purely on chance. The *Random-Greedy* approach is more stable as a result of the state transition criteria maintaining the best state seen so far, however, the approach suffers when the state maintained is sub-optimal. Although both of the random agents perform better than the agent initially, once the agent begins to learn the environment, both random agents are greatly surpassed.

Designed for Purpose. State-of-the-art fuzzers, such as EvoMaster, may be sufficient for finding surface-level DoS queries in GraphQL as can be seen across the results, however, it only scratches the surface. We observe a 68.85x and 7x increase in the max response time for discovery using the UNPROTECTED and PROTECTED settings respectively when compared to EvoMaster. It is essential to take into consideration that *EvoMaster* was designed for comprehensive coverage of the input space and not to discover DoS queries. Therefore, intrinsically in its structure it lacks the capabilities for array-based query batching, field duplication, and circular object exploitation which are features crucial for DoS query generation in GraphQL. These findings show the importance of having an approach designed for the problem

TABLE 1: Table displaying the percentage of queries skipped, rejected queries, and highest response time (HRT) for discovery as well as denial percentage for attack calibration and the number of queries required to perform the attack. Queries are skipped if an invalid action is chosen and queries are rejected by the application if they are malformed. Note that EvoMaster does not perform query skipping.

Approach	Setting	Query Discovery			DoS Attack	
		Queries Skipped	Rejected Queries	HRT	%Denial	Attack Queries
PPO	UNPROTECTED	43.35%	14.06%	1649.57s	99.998%	2 Queries
	PROTECTED	36.88%	1.25%	208.00s	99.852%	178 Queries
Random	UNPROTECTED	47.92%	3.91%	70.74s	99.956%	52 Queries
	PROTECTED	38.98%	1.95%	81.61s	99.847%	594 Queries
Random Greedy	UNPROTECTED	48.70%	0.26%	65.75s	99.962%	65 Queries
	PROTECTED	47.03%	0.39%	57.77s	99.726%	1169 Queries
EvoMaster	UNPROTECTED	-	27.34%	23.96s	99.729%	1222 Queries
	PROTECTED		28.59%	29.68	99.674%	1434 Queries

at hand, especially with the high-risk and complex nature of the GraphQL query space.

Impact of DoS in GraphQL. GraphQL's ability to provide a unified interface for applications, all the while reducing under- and over-fetching, can be very beneficial to benign users and attackers alike. This is exemplified in our *DoS Attack* results in Table 1, in which we achieve successful DoS attacks with queries discovered by our approach utilizing only 2 and 178 query requests for the UNPROTECTED and PROTECTED settings respectively. This demonstrates the importance of having an approach able to discover valid queries that pose a DoS threat for the purpose of application testing.

7. Availability

We release WENDIGO's Python prototype to the research community to promote its use in finding potential DoS queries for GraphQL applications, as well as future work in this direction. The WENDIGO prototype can be found at:
https://github.com/isneslab/Wendigo

8. Related Work

The context of GraphQL application fuzzing, besides EvoMaster [7], also been explored as part of a microservices architecture fuzzer [32], primarily focusing on detecting inconsistencies between the expected and actual returned types. However, these tools generally aim to cover the schema, maximizing coverage without being explicitly designed to uncover more than simplistic bugs.

For payload generation, mutational fuzzing has been applied for evading web application firewalls in WAF-A-MoLE [33] and DRL has been applied to SQL injection in SQiRL [11] as well as cross-site scripting in HAXSS [12]. Although these approaches successfully perform security testing in their respective domains, as a result of the fundamental differences in the syntax of domains as well as objective of the testing, would be incompatible with DoS security testing of GraphQL applications.

Finally, the presence and basic exploitation of DoS and other vulnerabilities in GraphQL has been covered in [24]. Static and dynamic tools such as InQL [34], GraphQL Cop [35] and BatchQL [36] test for these vulnerabilities. However, they focus on detecting each attack vector by performing simple checks. Therefore, they do not focus on the discovery of more complex queries utilizing a combination of these vulnerabilities. To the best of our knowledge, we are the first to utilize DRL to discover DoS exploitable GraphQL queries through the combination of vulnerabilities.

9. Conclusion

As GraphQL becomes increasingly prevalent and the cybersecurity landscape continues to be challenging, it is essential to recognize how DoS attacks affect GraphQL applications. The potential for significant disruption through the misuse of specific GraphQL features, which can lead to powerful attacks with minimal traffic, necessitates thorough testing. Therefore, this paper introduces WENDIGO, a novel agent-based black-box security testing method aimed at discovering DoS vulnerabilities in GraphQL queries. WENDIGO offers an automated solution that replaces the need for manually crafted test queries, facilitating developers in actively safeguarding their GraphQL applications against such threats.

As future work, we first intend to extend the capabilities of our approach by including more pathways for DoS exploitation and investigate the impact that different DRL algorithms and approaches (e.g., hierarchical DRL) have on our query discovery. Subsequently, we plan to test the most common GraphQL defenses utilized in the industry to evaluate their robustness to our approach, and identify if these defenses can be improved. Finally, we would like to collaborate with developers of real-world applications to be able to safely evaluate our approach in the wild, in addition to aiding in the improvement of DoS robustness for the evaluated application through responsible disclosure.

Acknowledgments

Research partially funded by: the Defence Science and Technology Laboratory (DSTL), an executive agency of the UK Ministry of Defence, supporting the Autonomous Resilient Cyber Defence (ARCD) project within the DSTL Cyber Defence Enhancement programme; UK EPSRC Grant no. EP/X015971/1; the SEND Mobility Consortium via an Erasmus+ Grant. Experiments received partial cloud service support from GARR Consortium.

References

[1] GraphQL Foundation, "Graphql landscape," https://landscape.graphql.org, accessed: January 2024.

[2] Statista, "Most exciting technologies for people working with application programming interfaces (apis) worldwide as of 2020," https://www.statista.com/statistics/1083227/worldwide-api-important-technologies/, accessed: January 2024.

[3] T. Mahjabin, Y. Xiao, G. Sun, and W. Jiang, "A survey of distributed denial-of-service attack, prevention, and mitigation techniques," *International Journal of Distributed Sensor Networks*, vol. 13, no. 12, p. 1550147717741463, 2017.

[4] G. Brito, T. Mombach, and M. T. Valente, "Migrating to graphql: A practical assessment," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 140–150.

[5] E. Wittern, A. Cha, J. C. Davis, G. Baudart, and L. Mandel, "An empirical study of graphql schemas," in *Service-Oriented Computing: 17th International Conference, ICSOC 2019, Toulouse, France, October 28–31, 2019, Proceedings 17*. Springer, 2019, pp. 3–19.

[6] State of GraphQL, "The state of graphql 2022: Features," https://2022.stateofgraphql.com/en-US/features/, accessed: January 2024.

[7] A. Belhadi, M. Zhang, and A. Arcuri, "Random testing and evolutionary testing for fuzzing graphql apis," *ACM Trans. Web*, vol. 18, no. 1, jan 2024. [Online]. Available: https://doi.org/10.1145/3609427

[8] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A neural network language {Model-Guided}{JavaScript} engine fuzzer," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2613–2630.

[9] C. Lyu, S. Ji, Y. Li, J. Zhou, J. Chen, and J. Chen, "Smartseed: Smart seed generation for efficient fuzzing," *arXiv preprint arXiv:1807.02606*, 2018.

[10] A. Arcuri, "Restful api automated test case generation," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2017, pp. 9–20.

[11] S. Al Wahaibi, M. Foley, and S. Maffeis, "SQIRL: Grey-Box Detection of SQL Injection Vulnerabilities Using Reinforcement Learning," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6097–6114.

[12] M. Foley and S. Maffeis, "HAXSS: Hierarchical Reinforcement Learning for XSS Payload Generation," in *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2022, pp. 147–158.

[13] T. T. Nguyen and V. J. Reddi, "Deep reinforcement learning for cyber security," *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[15] L. Xiao, X. Wan, C. Dai, X. Du, X. Chen, and M. Guizani, "Security in mobile edge caching with reinforcement learning," *IEEE Wireless Communications*, vol. 25, no. 3, pp. 116–122, 2018.

[16] G. Han, L. Xiao, and H. V. Poor, "Two-dimensional anti-jamming communication based on deep reinforcement learning," in *2017 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 2017, pp. 2087–2091.

[17] L. Xiao, Y. Li, G. Han, H. Dai, and H. V. Poor, "A secure mobile crowdsensing game with deep reinforcement learning," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 1, pp. 35–47, 2017.

[18] M. Chatterjee and A.-S. Namin, "Detecting phishing websites through deep reinforcement learning," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2. IEEE, 2019, pp. 227–232.

[19] C. Hicks, V. Mavroudis, M. Foley, T. Davies, K. Highnam, and T. Watson, "Canaries and Whistles: Resilient Drone Communication Networks with (or without) Deep Reinforcement Learning," in *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, ser. AISec '23. Association for Computing Machinery, 2023, pp. 91–101.

[20] M. Foley, M. Wang, Z. M, C. Hicks, and V. Mavroudis, "Inroads into Autonomous Network Defence using Explained Reinforcement Learning," in *Conference on Applied Machine Learning in Information Security (CAMLIS)*, 2022.

[21] M. Foley, C. Hicks, K. Highnam, and V. Mavroudis, "Autonomous Network Defence Using Reinforcement Learning," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '22. Association for Computing Machinery, 2022, pp. 1252–1254. [Online]. Available: https://doi.org/10.1145/3488932.3527286

[22] The Linux Foundation, "Openapi specification," https://github.com/OAI/OpenAPI-Specification, 2017, release: July 2017, Accessed: January 2024.

[23] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications,"

IEEE Transactions on Software Engineering, vol. 38, no. 2, pp. 258–277, 2012.

[24] N. Aleks, D. Farhi, and O. Chan, *Black Hat GraphQL: Attacking Next Generation APIs*. No Starch Press, 2023.

[25] W. Zhijun, L. Wenjing, L. Liang, and Y. Meng, "Low-rate dos attacks, detection, defense, and challenges: A survey," *IEEE access*, vol. 8, pp. 43 920–43 943, 2020.

[26] Y. Xiang, K. Li, and W. Zhou, "Low-rate ddos attacks detection and traceback by using new information metrics," *IEEE transactions on information forensics and security*, vol. 6, no. 2, pp. 426–437, 2011.

[27] A. Kuzmanovic and E. W. Knightly, "Low-rate tcp-targeted denial of service attacks: the shrew vs. the mice and elephants," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, 2003, pp. 75–86.

[28] M. Towers, J. K. Terry, A. Kwiatkowski, J. U. Balis, G. d. Cola, T. Deleu, M. Goulão, A. Kallinteris, A. KG, M. Krimmel, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai, A. T. J. Shen, and O. G. Younis, "Gymnasium," Mar. 2023. [Online]. Available: https://zenodo.org/record/8127025

[29] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[30] S. Huang, R. F. J. Dossa, C. Ye, J. Braga, D. Chakraborty, K. Mehta, and J. G. Araújo, "Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms," *Journal of Machine Learning Research*, vol. 23, no. 274, pp. 1–18, 2022. [Online]. Available: http://jmlr.org/papers/v23/21-1342.html

[31] Dolev Farhi, "Damn vulnerable graphql application," https://github.com/dolevf/Damn-Vulnerable-GraphQL-Application, 2021, release: February 2021, Accessed: January 2024.

[32] R. Mahmood, J. Pennington, D. Tsang, T. Tran, and A. Bogle, "A framework for automated api fuzzing at enterprise scale," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2022, pp. 377–388.

[33] L. Demetrio, A. Valenza, G. Costa, and G. Lagorio, "Waf-a-mole: evading web application firewalls through adversarial machine learning," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020, pp. 1745–1752.

[34] Doyensec, "Inql v5.0 - burp extension for advanced graphql testing," https://github.com/doyensec/inql, 2020, release: March 2020, Accessed: January 2024.

[35] Dolev Farhi, "https://github.com/dolevf/graphql-cop," https://github.com/dolevf/graphql-cop, 2022, release: February 2022, Accessed: January 2024.

[36] Assetnote, "Batchql," https://github.com/assetnote/batchql, 2019, release: August 2019, Accessed: January 2024.