Relevance Score Attacks on Elasticsearch

Noah Hirsch Rohan Dandavati Esme Bajo

ABSTRACT

This paper attempts to expand on the Search Text Relevance Score Side-channel attack described in Wang et al. [1]. We seek to observe the side channel attack, precisely as described in [1], in both a controlled Kibana environment and on GitHub. Then, we examine the potential of repository name attacks on GitHub, evaluating the likelihood of character-by-character attacks on private repository names via a statistical approach and a shard mapping technique.

1. INTRODUCTION

We present a series of attempted attacks on Elasticsearch based on the STRESS attack detailed in [1]. The basic idea is that Elasticsearch results include relevance scores that are computed with respect to *all* documents, not just public ones. Therefore, if one were to query a term that appeared in no other documents and a term of the same length that appeared in many documents (even private ones), the term that appeared with a high frequency might yield a lower relevance score due to its presence in other documents. A more detailed discussion of the computation of relevance scores can be found in Section 2.1.

In this paper, we focus primarily on repository name attacks. The relevance score attack still works on repository names, but some noise is reduced; almost every 8 character string will appear in the text of some document, but not necessarily as the name of some repository. To further eliminate noise and focus on the general concepts, some of our attacks were done in a controlled Kibana environment. This methodology is detailed in Section 4.1. This allowed us to control additional variables, including the number of shards and the method of tokenizing. Most of these points of attack will serve as warnings of how not to implement Elasticsearch rather than attacks that can be realistically weaponized.

2. RELATED WORK

2.1 TF-IDF and the STRESS Attack

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

Our attack is enabled by the use of the Term Frequency-Inverse Document Frequency (TF-IDF) statistic that is the basis of Elasticsearch [3]. The TF-IDF of a document for a given term is a function both of how often the term appears in the document, and how often the term appears in the entire set of documents. Essentially, TF-IDF requires the use of two functions: tf and idf. The function tf takes in a term t and a document d and returns some measure of t's frequency within d. The function idf takes as input a term t and a set of documents D and measures t's inverse frequency within the set D. Thus, the more frequently t appears within D, the lower its idf score is. The tf function can simply return the number of times t appears in d, but a more complicated function may also be used, depending on what aspects of frequency one wants to emphasize. For example, Elastic uses the square root of the number of counts as a measure of frequency. This means that all documents with relatively large counts of a given term will all display relatively small differences in frequency of that term, as the frequency is the square root of the count.

The idf function used by Elastic is computed as the expression

$$1 + \log\left(\frac{numDocs}{docFreq + 1}\right) \tag{1}$$

where $numDocs$ is the total number of documents in the index, and $docFreq$ is the number of documents containing the term. Given the two functions tf and idf, the TF-IDF value of a given document d in an index D and a term t is given by

$$tf(t, d) \times idf(t, D).$$

The use of the TF-IDF measure means that the relevance score of a document, as returned by Elastic, is a function of other documents contained within the index. Therefore, one can strategically use relevance scores to infer information about other (potentially private) documents within an index. This document frequency side channel was discovered by Büttcher and Clarke and described in [4] in 2005, but was only successfully tested and exploited in active search engine deployments by Wang et. al in 2015 [1].

The attack works as follows. Suppose there is some term t that an attacker believes could be contained in a document owned by the victim and further suppose that t is of sufficient entropy that it is not contained in any other document. The attacker can then create a document containing this same term, enter it into the index, and search the term. Assuming that the victim's document is not visible to the attacker (there would be no need for an attack if it were),

the victim's document will be removed from the list of documents containing t and ranked by its relevance score, before it is returned to the attacker. However, as the inverse document frequency is taken into account while calculating the relevance score, the relevance score of the attacker's document will be lower than it would have been t was not contained in the victim's document. This is referred to as *score-dipping*. To see if this is the case, the attacker can create an additional document containing another high entropy term, t', and observe the score of the document using t' as the search term. If there is a significant difference in these scores, the attacker knows that t must be contained in some other document in the index, even though they themselves cannot see the document.

Note that the experiments we perform are in Kibana and on GitHub, both of which invoke *sharding*. Essentially, different documents are placed on one of several "shards" and the index D with respect to which the relevance scores are computed only contains documents on the same shard as the returned document.

3. RESEARCH QUESTIONS

After reading the results obtained in [1], we were interested in first recreating and then attempting to expand upon the attack. As proof of concept, we first aim to implement an exact-match attack both in the Kibana environment and on GitHub private repository names. From here, we want to explore the effects of different variables in Elasticsearch, including wildcards, tokenizing, and sharding. How might we employ wildcard scoring to gain information about the presence of a term starting with a given prefix? Can we use this to implement a character-by-character attack on a string, thereby reducing the time complexity of retrieving a string of length n from around 26^n (in the exact-match scenario) to something on the order of $26n$ (in the character-by-character scenario)?

With regard to searches on GitHub repository names, the GitHub RESTful API returns public repository names sorted by relevance score. We wish to expand on the STRESS attack by extracting information about private repository names with this score. More specifically, GitHub claims to implement a Google-like search optimization [2]. This suggests the possibility of a partial-match repository name attack, in which a search query for a portion of a private repository name may exhibit a score-dipping side channel. We wish to investigate this possibility by attempting partial-match attacks on private repository names.

Finally, is there any way we can transform sharding from an obstacle in the attack to a weapon? For example, if we can accurately map out the shards containing GitHub repository names, can we use information about the shard that a repository name is on to narrow down the user's identity? Thorough explanations of how these questions were addressed and the varying successes and failures of these approaches follow in the next few sections.

4. TECHNICAL SETUP

Here, we discuss the Kibana/Elasticsearch and GitHub settings we set when performing our attacks, the reasoning behind these choices, and the implications.

4.1 Kibana Settings

We downloaded Elasticsearch (version 6.4.2) on a local machine, as well as an instance of Kibana to adjust different settings and make queries. The role of Kibana in our project was to provide a controlled setting, where we could propose and initially test potential attacks. Therefore, we made our setup as simple as possible; before uploading any victim and attack documents, we forced the number of shards to be 1, so that all existing documents would be used to compute the relevance score any document returned in a query. We uploaded the victim and attacker documents using .json files and performed queries within the Kibana console.

In our setting of Elasticsearch via Kibana, queries were tokenized based on whitespace and hyphens. Therefore, the query "abc-123" would result in two tokens, "abc" and "123." We left the scoring on the default setting, except when we attempted wildcard attacks. The default setting for wildcard queries is to return 1 if the prefix is present and 0 otherwise, thus preventing any character-by-character attack based on wildcard searching. In the Kibana environment, we changed the wildcard scoring method to be scoring_boolean in an attempt to retrieve more informative (and less secure) scores.

4.2 GitHub API

In order to test the DF side-channel on GitHub's instance of Elastic, we needed a way to automate our attacks. As we had decided to perform our attack on the repository names, this required that we be able to automatically perform tasks including creating and deleting repositories. We also needed to be able to submit a query and return a list of repositories with their raw relevance scores.

Before we could automate these attacks, we first needed to configure GitHub accounts to facilitate the attacks themselves. We created GitHub accounts which we could use to create public repositories. Because of the way free GitHub accounts are structured, we were not able to create private repositories on these throw-away accounts. We instead had to use our personal accounts (which were enabled with student developer privileges) to create private repositories.

Once we had our GitHub accounts set up, we relied on both the PyGithub package, as well as code written as part of Wang et al (and provided to us by Professor David Cash, a coauthor of the paper) to automate the necessary tasks. We chose to use PyGithub because it had simple commands for creating and deleting repositories. The other code was used for its search functionality, which was implemented through queries to the GitHub API using the requests library. These packages allowed us to create a recipe for our first attack, described in Section 5.3, and we continued to use these tools for the remainder of our attacks on GitHub.

While the combination of these two code bases was sufficient for our initial requirements, we soon ran into issues once we tried to expand on our range of attacks. One of the most persistent problems we ran into was the issue of rate limiting on the GitHub search API. To get around this we each ran our own tests, each person using their own throw-away account. We also put a wrapper around our search function that would attempt a search and sleep for a given amount of time if the search failed due to rate limiting (we usually set this sleep time at 5 seconds). This ensured that our tests did not fail due to rate limiting errors, but also meant some of our tests took several minutes to execute. If we were to continue with this type of research, we would

need to create additional dummy accounts to get around rate limiting. We ran into a further issue with the GitHub API when we attempted to create a shard map for the repository name data. Neither of the libraries we used supported changing repository names, which was necessary in order to create a shard map (as we need to be able to test if a repository name score-dipping attack is possible on any two given repositories). We attempted to automate repository name changing through the REQUESTS package, but had little luck.

Overall, using the code from Wang et. al and the PyGithub package was sufficient for our initial tests. However, for further testing, we will need a better understanding of the GitHub API in order to expand our capabilities.

5. RESULTS

5.1 Exact Queries and Exact Tokens

To get started, we made sure the exact-match attack was still working. In the Kibana environment, after setting the number of shards to be one, we uploaded three documents: a victim file "catdogcat" and two attacker files "catdogcat" and "catdogcar." From the attacker's point of view, we can only see the files "catdogcat" and "catdogcar" but the relevance scores when we queried these terms revealed that there was another file with an exact match for "catdogcat." When we queried the string "catdogcat," our known document "catdogcat" returned with a relevance score of 0.47000363. But when we queried the string "catdogcar," our known document "catdogcar" returned with a relevance score of 0.9808292. The lower score of 0.47000363 would indicate the presence of a document entitled "catdogcat" to our attacker.

In addition, we tried the same attack in Kibana, but token by token. If we can get an exact-match attack on n tokens of length $\ell_1, ..., \ell_n$, then we could retrieve the full term in time

$$\sum_{i=1}^{n} 26^{\ell_i} < 26^{\sum_{i=1}^{n} \ell_i}.$$

We inserted a victim document "123-45-6789" and two attacker documents "122-00-0000" and "123-00-0000." The query "122" returned the document "122-00-0000" with a score of 0.9808292 and the query "123" returned the document "123-00-0000" with a score of 0.47000363, due to the presence of the victim "123-45-6789" document. In this idealistic scenario, i.e. a scenario with one shard and no noise, we can thus recover the entire document "123-45-6789" with no more than $10^3 + 10^2 + 10^4$ queries.

5.2 Wildcards

Next, we attempted to extend this term-by-term attack to a character-by-character attack by using wildcard queries. This attempt failed, so this attempt will serve as a warning against bad Elasticsearch implementations rather than a practical attack. The default wildcard scoring in Kibana is to return a relevance score of 1 if a document matches the given query. That is, if we add the document "catdog" and query "cat*," the document "catdog" will yield a relevance score of 1, regardless of the number of other documents with the prefix "cat" on that shard. This defeats a relevance score attack. To combat this, we set the variable "rewrite" to "scoring_boolean" in the Kibana console. Now, despite there being documents "123456789," "123450000," and "123460000" on one shard and no noise, the queries "12345*" and "12346*" return the documents "123450000" and "123460000," respectively, with the same score: 0.9808292. This is the desired behavior, as it defeats a character-by-character attack. It is important that sites like GitHub that deploy Elasticsearch adopt wildcard scoring systems that do not suggest the presence of victim documents with matching prefixes.

5.3 Repository Name Attack on Github

When searching for a repository name, the Github API returns a list of possible results ordered by score value. We investigated this score for various vulnerabilities related to the TF-IDF side channel and private repository names. We briefly looked into the viability of an exact-match attack on a private repository name, which seemed promising. However, we quickly moved past this attack as its threat model is relatively weak. This is because the attack presumes a large amount of a priori information. The attack goal is to query the exact private repository name to observe score dipping, so the attacker must know the entire name beforehand. It is possible for an attacker to search for the existence of an exact known repository, such as determining whether SoftwareV1.2.3 is in development. Otherwise, if they do not know n characters, their target repository name is one of 26^n exact possibilities (which is likely to be prohibitively large given GitHub's rate limiting). Furthermore, we ran into issues where the exact match search did not consistently exhibit score dipping, possibly due to our attack repository being on a different shard than the victim repository.

We found the possibility of a seed-based character-by-character side channel to be more interesting. The idea with this attack would be to start with a realistic known seed query that matches some substring of the victim repository. Then, we would create and search for 26 repositories, one for each alphabetical character appended to the seed, and look for score dipping. The attack algorithm would append the score-dipping character to the seed on each iteration, and look for the next character. Fortunately, we were not able to identify any effective method for such a character-by-character attack. Using a random 32-character repository name, we tried seeds of length 6-31, and tried appending characters to the start and the end of the seed. So, it does not seem feasible to use the TF-IDF side channel to build out a private repository name from a known seed. It is important to note that our inability to carry out this attack may have been due to the lack of a shard map for the repository name indices. So, we were unable to determine if and when our attacking repositories were on the same shard as the victim repository. To create a shard map, the ability to rename repositories is required. We could not reliably rename our repository names, and so we could not confirm that sharding was the reason for the seed attack's failure. Furthermore, given Github's search query rate limiting, exact match attacks are not feasible either. So, we have not identified any legitimate vulnerabilities with Github's repository search system.

```
SUMS_OF_SCORES = np.zeros(26)
for n ∈ range(NUM_TRIALS) do
    for r ∈ get_repos() do
        r.delete()
    end for
    for c ∈ {a,b,...,z} do
        create_repo_safe(QUERY_PREFIX + c)
    end for
    for i, c ∈ enumerate({a,b,...,z}) do
        SCORE = make_query_safe(QUERY_PREFIX + c)
        SUM_OF_SCORES[i] += SCORE
    end for
end for
return chr(min_index(SUM_OF_SCORES) + 97)
```

6. DISCUSSION

6.1 Future Work

As mentioned previously, if we were to further our research, we would plan to create a shard map for the repository name data. We believe that our findings for the repository name attack were influenced by the use of shards in GitHub's instance of Elasticsearch. In order to improve on our attack, and determine whether or not our issues were caused by repositories/repository names being stored on different shards, it would be necessary to create a mapping of all possible shards onto which we could insert data. Pseudocode for creating this shard map was developed in [1]. We include a version of this pseudocode adapted to our repository name attack below.

```
Input: n, a maximum number of iterations
Output S, a set of repositories on different shards
x ← create_repo(generate_repo_name())
S ← {x}
for i ∈ {1, ..., n} do
    y ← create_repo(generate_repo_name())
    if y is on the same shard as any repo in S then
        continue
    else
        S ← S ∪ y
    end if
end for
return S
```

In the above code, generate_repo_name() refers to a function that returns a random repository name of sufficient length (for our attempts we used 32 characters). We attempted to implement this algorithm in order to create a shard map, but ran into issues with testing if two repository names are kept on the same shard. To do so, one must first create two repositories. Suppose these are named "r1" and "r2". If both of these names were stored on the same shard, we would expect to observe score dipping if we changed the name of the second repository so it was also "r1" — searching for "r1" would return a lower relevance score than searching for "r2". If the repository names were not on the same shard, no such score dipping would occur. Therefore, it would be possible to create such a shard map given sufficient time (as it is sometimes necessary to sleep during the shard test in order to ensure that the new repository has been indexed and that its name change is properly reflected on its respective shard) and the ability to rename repositories. As we recently attempted to create a shard map, we ran into issues when trying to change repository names through queries, as described in [2]. We unfortunately did not have enough time to resolve these issues and still take the time to actually create a shard map, but we believe this to be a useful next step.

There are many things that could be learned from creating a shard map. At first, we believed that all repository names were stored on a single shard, in order to enable quick and complete searching on repository names. We thought this would be possible as it takes very little data to store the name of a repository in comparison to all the terms contained in documents within the repository and their respective frequencies. As we continued with our attacks, we soon inferred that this was unlikely, as we were not consistently able to observe score dipping on repository name searches. However, it is still to be seen how many shards are used by GitHub to store this data. In [1], the authors found 191 shards while creating a shard map of GitHub's instance of Elasticsearch. While it seems as if repository names are not all stored on one shard, is it necessarily true that they are distributed among the entire set of shards? If the repository names are stored on a subset of these shards, this would enable a score dipping attack that could be executed more quickly and with fewer queries to the GitHub API, as opposed to a score dipping attack of the type described in [1].

6.2 Weaponizing the Work from Wang et al.

Future research could also involve weaponizing the work from Wang et al. As score dipping attacks also inherently indicate when two repositories are stored on the same shard, these types of attacks reveal information about the basic shard structure underlying GitHub's instance of Elastic. With these attacks, one can infer which shard holds the data relating to a specific repository. We believe that this type of information could be relevant if an adversary wants to perform a negative SEO attack. Negative SEO attacks involve attempting to lower an victim's ranking or relevance score for a given search engine. It is possible that an adversary could determine the shard/node where a victim's documents are indexed, and use it to target an attack towards that specific node in order to negatively impact the victim's relevance scores for GitHub searches. The targeted attack could involve inserting many documents onto the same shard with similar terms as the victim's documents, in order to drive down relevancy scores for the documents under searches for the given term. An alternative attack could involve finding some method to take the node offline, so no relevance scores are calculated from the documents on the affiliated shard. However, this type of attack could be prevented by keeping backup shards in case of a node failure.

7. ACKNOWLEDGMENTS

We would like to thank Professor Cash for introducing us to this work and for his constant support and guidance throughout this project. In a show of gratitude, we plan to pay him a significant sum of DCash (hopefully before it takes off and becomes too valuable for us to afford).

8. REFERENCES

[1] L. Wang, P. Grubbs, J. Lu, V. Bindschaedler, D. Cash, and T. Ristenpart. "Side-Channel Attacks on Shared

Search Indexes," 2017 IEEE Symposium on Security and Privacy (SP), San Jose, California, USA, 2017, pp. 673-692. doi:10.1109/SP.2017.50

[2] Github REST API v3 Documentation. https://developer.github.com/v3/search/search-repositories

[3] Theory Behind Relevance Scoring. https://www.elastic.co/guide/en/elasticsearch/guide/current/scoring-theory.htmlidf

[4] S Büttcher and C. L. A. Clarke. A Security Model for Full-Text File System Search in Multi-User Environments. *FAST '05: 4th USENIX Conference on File and Storage Technologies.*