Security Vulnerability Information Service with Natural Language Query Support

Carlos Rodriguez[1], Shayan Zamanirad[1], Reza Nouri[1,2], Kirtana Darabal[1], Boualem Benatallah[1], and Mortada Al-Banna[1]

[1] UNSW Sydney, NSW 2052, Australia
[2] QANTAS Airways, NSW 2020, Australia
{crodriguez,shayanz}@cse.unsw.edu.au, {s.nouri, kirtana.darabal}@unswalumni.com, {boualem,mortadaa}@cse.unsw.edu.au

Abstract. The huge data breaches and attacks reported in the past years (e.g., the cases of Yahoo and Equifax) have significantly raised the concerns on the security of software used and developed by companies for their day-to-day operations. In this context, becoming aware about existing security vulnerabilities and taking preventive actions is of paramount importance for security professionals to help keep software secure. The increasingly large number of vulnerabilities discovered every year and the scattered and heterogeneous nature of vulnerability-related information make this, however, a non-trivial task. This paper aims at mitigating this problem by making security vulnerability information timely available and easily searchable. We propose to enrich and index security vulnerability information collected from publicly available sources on the Web. To make this information easily queryable we propose a natural language interface that allows users to query this index using plain English. The evaluation results of our proposal demonstrate that our solution can effectively answer questions typically asked in the security vulnerability domain.

Keywords: Security Vulnerability · Indexing · Natural Language Interfaces · Information Integration

1 Introduction

In July 2017, one of the most notorious cyber security attacks was discovered in Equifax's dispute portal servers, which resulted in a breach of personal information of approximately 145 million individuals[3]. The attack was possible due to a known and unpatched vulnerability found in their servers running Apache Struts (https://struts.apache.org). Reports[4] estimate a breach-related cost of $439 million through the end of 2018.

[3] https://www.gao.gov/assets/700/694158.pdf
[4] https://www.reuters.com/article/us-equifax-cyber/equifax-breach-could-be-most-costly-in-corporate-history-idUSKCN1GE257

R. Nouri, K. Darabal - This work was done while the authors were at UNSW Sydney.

The final authenticated version is available online at https://doi.org/10.1007/978-3-030-21290-2_31

While events like these have significantly raised the concern regarding software security, the ever increasing reliance on software systems to support business operations and the rising numbers of new vulnerabilities reported every year make the task of keeping software systems secure a very difficult one [19]. Recent cybersecurity reports[5] show that in year 2017, approximately 21,000 vulnerabilities were discovered and reported. This value is 31% larger than what was discovered in the year before. Furthermore, as of mid-2018, more than 10,000 vulnerabilities were disclosed[6], of which 16.6% have high severity scores (CVSSv2[7]) ranging between 9 and 10.

In order for security professionals to become aware and informed about security vulnerabilities, an integrated access to such information is needed. However, while much of the security vulnerability information is publicly available (e.g., Vulners (https://vulners.com), NVD (https://nvd.nist.gov) and OWASP (https://owasp.org)), such information is in many cases scattered across different, heterogeneous and complex information silos that have low or no integration [7]. For example, while NVD provides a curated and uniquely identified list of vulnerabilities, finding the list of software affected, exploits and patches for a given vulnerability requires in most cases querying separately multiple, disparate sources. Moreover, accessing such information may require different forms of query mechanisms including manual keyword search, the use of domain-specific languages (DSLs), REST API calls, among other mechanisms.

This paper aims at mitigating the problems above by providing an integrated source of vulnerability information that leverages on publicly available information about security vulnerabilities. More specifically, we propose to leverage on multiple, heterogeneous and complementary sources, which we further enrich using state-of-the-art Knowledge Graphs (KGs) [17] and vector representation of words (word embeddings) [12] to enable a richer representation of vulnerability information. Such integrated and enriched information is thus capable of not only providing information about vulnerabilities alone, but also affected software and vendors, associated exploits, attacks and patches, which jointly can help understand and mitigate the risks posed by security vulnerabilities. In order to overcome the complexity of querying such heterogeneous information, we propose a Natural Language Interface (NLI) that allows security professionals to seamlessly query security vulnerability information. Such NLI does not require learning ad-hoc languages for query purposes, nor it needs familiarity with the underlying information schema. In summary, the contributions of this paper are:

- We propose an approach and architecture to (i) *collect* and *integrate* security vulnerability information from multiple, heterogeneous and disparate sources, (ii) *enrich* the integrated information with existing KGs and word embeddings, (iii) *index* such information, and (iv) *query* with support for natural language (NL) expressions.

[5] https://pages.riskbasedsecurity.com/2017-ye-vulnerability-quickview-report
[6] https://www.riskbasedsecurity.com/2018/08/more-than-10000-vulnerabilities-disclosed-so-far-in-2018-over-3000-you-may-not-know-about/
[7] https://www.first.org/cvss/v2/guide

- We devise an NLI that is able to translate NL expressions into queries that are executable by the underlying index and search engine.
- We evaluate our proposed approach and demonstrate that our solution is able to effectively answer questions typically asked in the security vulnerability domain using NL queries.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents the security vulnerability information model used for our index. Section 4 presents our architecture and discusses the collection, enrichment and indexing of security vulnerability information. Section 5 discusses how NL queries are translated into the underlying search engine's query language. Next, Section 6 presents the evaluation of our proposal. We conclude the paper with Section 7.

2 Related work

We explore related work from two perspectives that are key to our work: Security vulnerability information sources and information querying with NL support.

Security Vulnerability Information Sources. The cybersecurity domain have traditionally relied on multiple sources when it comes to inquiring about security vulnerabilities. One of the most widely used sources is the National Vulnerability Database (NVD) (https://nvd.nist.gov), a U.S. government repository of security vulnerability information. NVD provides a list of vulnerabilities dating back to year 1988, where each vulnerability is uniquely identified by its CVE ID (Common Vulnerabilities and Exposures). Another example is the Zero Day Initiative or ZDI (https://www.zerodayinitiative.com), which allows security researchers to privately report 0-day vulnerabilities to vendors. Vulnerabilities are collaboratively made public by ZDI and the affected vendor through a joint advisory. Other useful sources of vulnerability-related information include security bulletins and advisories created and managed by vendors. Examples include Mozilla's security advisory (https://www.mozilla.org/en-US/security/advisories/), Redhat's product security center (https://access. redhat.com/security/) and the Apache Security Team (https://www.apache.org/security/). Further sources exist that provide useful archives of exploits (https://www.exploit-db.com), breach reports (https://breachlevelindex.com), vendor-specific patches (https://portal.msrc. microsoft.com) and crowd-sourced vulnerability reports (https://www.hackerone.com). Vulners (https://www.vulners.com) aims at partly mitigating the heterogeneity and complexity of security vulnerability information by normalizing and aggregating the available sources above. However, in the current version, there is no integration among the difference sources and the query interface is still at the complexity level of the DSL of the underlying indexing and search engine.

Information Querying with NL support. The use of NL for querying information sources has been largely enabled thanks to the research carried out

mainly in the intersection of areas such as Natural Language Processing (NLP) [5], database systems [2] and Semantic Web technologies [6]. Among the works that combine NLP with Semantic Web, Tablan et al. [18] propose QuestIO, an NLI that allows for querying structured information in a domain independent fashion. The work leverages both NLP techniques and semantic web technologies to help query structured information stored in ontologies. In the same line, Lopez et al. [10] propose PowerAcqua, a Q&A system for querying information stored in heterogeneous, semantic resources. Its main contribution lies in the ability to combine multiple, large and heterogeneous information sources, which helps empower their Q&A system. Kaufmann et al. [8] propose Querix, a system that allows for asking users for clarification whenever ambiguities emerge while querying ontologies using NL.
On a different front, other works looked into providing NLIs to query more traditional database systems. In this context, Li and Jagadish [9] propose NaLIR, an NLI that is able to translate NL queries written in English into SQL statements. Similarly, PRECISE [13] is a system that translates NL queries into SQL queries using a lexicon, which helps expand the NL query vocabulary. TiQi [14] proposes an NLI that leverages on previous work (PRECISE [13]) to help query traceability information in software repositories. In our paper, we do not aim at advancing the field of NLIs. However, we do leverage on some of the techniques discussed above to provide NL query support over security vulnerability information. To the best of our knowledge, our work is the first to provide unified, integrated, enriched and indexed security vulnerability information with NL query capabilities.

3 Security Vulnerability Information Model

Building an integrated source of security vulnerability requires, first of all, the identification of useful information sources that will meet the information needs of users inquiring about security vulnerability. Several such information sources exist as previously discussed in the related work section. However, *what are the concrete elements users inquire about in the context of security vulnerablities?* In an empirical study on questions asked while diagnosing security vulnerabilities, Smith et al. [16] identified a total of 78 questions typically asked in this context, which are categorized into (i) vulnerabilities, attacks and fixes (e.g., *"how can I prevent this attack"*), (ii) code and applications (e.g., *"where is this method defined?"*), (iii) individual questions (e.g., *"have I seen this before"*), and (iv) problem solving support (e.g., *"can my team members/resources provide me with more information?"*).

The study above provides a useful guide into typical, vulnerability-related questions. It is worth noting, however, the wide range of questions asked in relation to security vulnerabilities, many of which go beyond inquiring strictly about security vulnerability information. For example, category (ii) focuses mostly on questions related to the programming code being analyzed, while category (iii) involves developers' self-reflection, understanding and expectation questions. In

our work, we leverage on the results of this study and exclusively focus on questions related to *inquiring about security vulnerability information* that is publicly available on the Web. Table 1 shows an excerpt of questions as emerged from [16].

Table 1. Sample of questions asked when diagnosing security vulnerabilities (extracted from [16]). Terms that are relevant for the security vulnerability domain are underlined.

Security vulnerability questions
Q1: "Is this a real vulnerability?"
Q2: "What are the possible attacks that could occur?"
Q3: "How can I prevent this attack?"
Q4: "How can I replicate an attack to exploit this vulnerability?"
Q5: "What is the problem (potential attack)?"
Q6: "What are the alternatives for fixing this?"
Q7: "How do I fix this vulnerability?"
Q8: "How serious is this vulnerability?"
Q9: "Are all these vulnerabilities the same severity?"

The questions listed in Table 1 emphasize on five main information elements (see the underline words), namely, *vulnerabilities / weaknesses*, *attacks*, *fixes* and *exploits*. Leveraging on our experience and results from previous research [1, 3] on security vulnerability discovery, exploration and understanding, we derive the model shown in Figure 1(a), which aims at capturing the information elements listed above. In this model, the *vulnerability* entity represents a reported vulnerability, which is characterized by properties such as an *id* (i.e., the CVE of the vulnerability), *publishedDate*, *description*, among other properties. A vulnerability typically exists in a *software* that is developed by a *vendor*. Moreover, a vulnerability is typically reported by a *discoverer* (e.g., a white hat hacker). Vulnerabilities are further characterized by a *weakness* that is uniquely identified by an identifier known as CWE (Common Weakness Enumeration) (https://cwe.mitre.org). Besides the information above, we also consider additional entities such as *exploits*, *attacks* and *patches*. An *exploit* is a piece of software or data that can take advantage of an existing vulnerability. An exploit can be used to *attack* a software containing a vulnerability. Finally, a *patch* is a fix to a software that can help mitigate the risks of being attacked due to a vulnerability.

The information model presented in Figure 1(a) integrates different entities that jointly provide a fuller and richer picture about security vulnerabilities. In the next section, we discuss in more details how existing, disparate information silos publicly available on the Web can be integrated, enriched and indexed for providing a unified access to security vulnerability information.

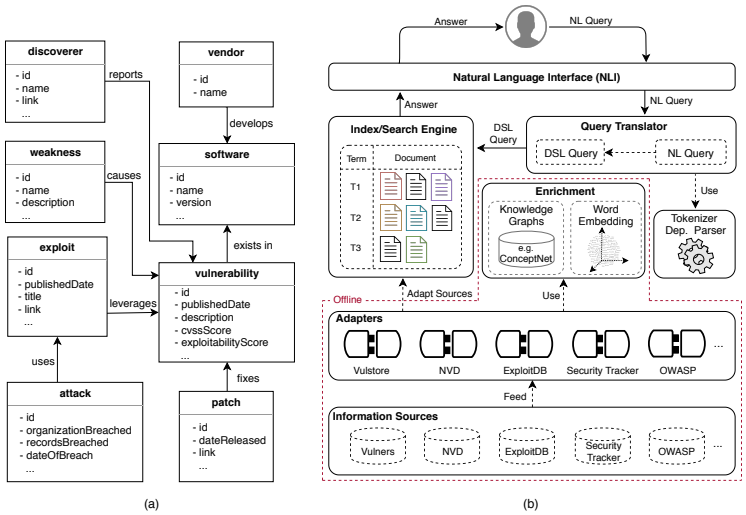

Fig. 1. (a) Security Vulnerability Information Model (we show just an excerpt of the properties for each entity). (b) Architecture for collecting, enriching, indexing and querying security vulnerability information. The bottom part of the architecture operates offline, while the upper part does it online.

4 Collecting, Enriching and Indexing Security Vulnerability Information

The approach proposed in this paper consists of three phases, namely, (i) information source collection, adaptation and enrichment, (ii) information indexing, and (iii) information querying with NL query support. This section presents the first two (we discuss the latter separately in Section 5). We show the overall architecture of our proposed solution in Figure 1(b) and use it as a reference to elaborate on each of the phases above.

4.1 Security Vulnerability Information Collection, Adaptation and Enrichment

At the bottom of Figure 1(b), we show examples of various, publicly available *information sources* that can be used to collect security vulnerability information. The collection of such information can be done through various mechanisms, including REST API calls and web data extraction. For example, while the information provided by Vulners is represented as JSON documents and accessible through REST APIs, SecurityTracker's information is available mainly as HTML web pages, which requires for web data extraction techniques [4]. The tasks of accessing, collecting, adapting and integrating the information sources to the target representation (which is presented in the next section) require, therefore, the creation of dedicated adapters for each of the information sources of interest. A list of adapters is exemplified in the *Adapters* component at the bottom of Figure 1(b).

In order to have a richer representation of the integrated information resulting from the previous step, we propose to enrich such information with semantics from the security vulnerability domain (see the *Enrichment* component in Figure 1(b)). Such enrichment will allow us to provide more flexibility in expressing NL queries. Consider, for example, an NL query such as *what vulnerabilities are there in Internet Explorer?*. In this query, a user may choose to ask the same question using different mentions for *Internet Explorer*, such as *IE* or simply *Explorer*. The enrichment of the original security vulnerability information aims at enabling the possibility of using alternative mentions for such entities, thus, allowing for more flexibility in NL expressions.

Security vulnerability information enrichment is performed at two levels. First, at the ***attribute level***, we store the various mentions that can be used to refer to an attribute in our information model (see Figure 1(a)). For example, for the attribute *vulnerability.publishedDate* (we use the dot notation, *entity.attribute*, to refer to attributes of an entity), we store other mentions such as {*publication date*, *release date*, *announcement date*, ...}. In this way, whenever a user uses any of these mentions in an NL query (e.g., *release date*), we can associate it to the target attribute (i.e., *vulnerability.publicationDate*). Second, at the ***value level***, we store mentions of attribute values for relevant attributes within this domain. For example, one relevant attribute is *weakness.name*. A possible weakness (i.e., attribute value) in the context of security vulnerabilities is *Improper Neutralization of Input During Web Page Generation*. This weakness is, however, also commonly referred to as *CWE-79*, *XSS* and *Cross-Site Scripting*[8]. With this enrichment we can therefore refer to weakness *CWE-79* by using any of its alternative mentions.

The enrichments above are performed by leveraging on named-entity recognition [11], KGs [17] and word embeddings [12] (see the *Enrichment* component in Figure 1(b)). Named-entity recognition is used to recognize named-entities appearing in attributes of the security vulnerability information. In this work, we focus on three main entity types that are relevant in this domain, namely, *software*, *weakness* and *vendor*. Examples of such named-entities include *Internet Explorer* (software), *Microsoft* (vendor) and *XSS* (weakness). The named-entity recognizers (we use Stanford's NER [11]) are trained using a combination of publicly available lists of named-entities (e.g., for software, NVD's Common Platform Enumeration (CPE) list[9]), which are extended with alternative mentions using existing KG. Such mentions are obtained from ConceptNet[17] through its APIs, by leveraging its *synonym relation*. In addition, for enrichment at the schema level, we use word embeddings [12] trained on data from Information Security Stack Exchange (https://security.stackexchange.com), which allows us to enrich attribute mentions with semantically-related terms from the security domain. Next, we show how we represent this information for indexing purposes and how we extend it with the enrichments discussed in this section.

[8] https://cwe.mitre.org/data/definitions/79.html
[9] https://nvd.nist.gov/products/cpe

4.2 Security Vulnerability Information Indexing

The majority of security vulnerability information available on the Web is of unstructured or semi-structure nature [7]. Much of such information consists of textual descriptions of vulnerability-related artifacts such as exploits, patches, breaches and security advisory bulletins. In order to efficiently query this information, we therefore propose to rely on existing indexing and searching technologies that are capable of efficiently dealing with such unstructured and semi-structured information. More specifically, in this work we rely on ElasticSearch (https://www.elastic.co) (see the *Index / Search Engine* component in Figure 1(b)), an open-source index and search engine based on Apache Lucene (https://lucene.apache.org).

In ElasticSearch, indexes are flat collections of documents repesented as JSON. In order to represent our information model shown in Figure 1(a), we therefore need to translate that model into JSON documents while still keeping the relationships among the different entities of the model[10]. We do so by denormalizing (flattening) the model in Figure 1(a). The result is shown in Figure 2(a). In this representation, we have one document per each vulnerability. This document is self-contained (and vulnerability-centric) meaning that all *related* entities (as shown in Figure 1(a)) are contained within the same document. As an example, Figure 2(b) shows a document for vulnerability *CVE-2009-1295*, which also includes related entities such as software affected (e.g., *Ubuntu*) and weaknesses (e.g., *Configuration*) involved. This representation will allow us to effectively retrieve documents from our index to answer queries such as *Vulnerabilities in Ubuntu, with weakness CWE-16*, which involves relationships (encoded in our JSON representation) found in our security vulnerability information model (Figure 1(a)).

In addition to the attributes of our original information model (Figure 1(a)), we add to each document the *enrichments* discussed in the previous section (see the shaded attributes in Figure 2(a) and (b)) . More specifically, we extend the original information model with additional attributes that contain mentions of named-entities that are relevant in this domain. As explained before, in this paper we focus on the entity types *software*, *vendor* and *weakness*. Figure 2(b) shows examples of alternative mentions for the named-entities *Ubuntu OS* (the software), *Ubuntu* (the vendor) and *CWE-16* (weakness).

Finally, besides indexing security vulnerability information, we also keep a separate index in which we store different mentions of the attributes of our information model. The mentions are stored using the schema shown in Figure 2(c), where each JSON document stores the name of an attribute (as used in the main index schema) and its alternative mentions. Figure 2(d) shows an example JSON document for the attribute *publishedDate*. In the next section, we show how the enrichments (i.e., entity and attribute mentions) discussed above are used by our NLI.

[10] https://www.elastic.co/guide/en/elasticsearch/guide/current/relations.html

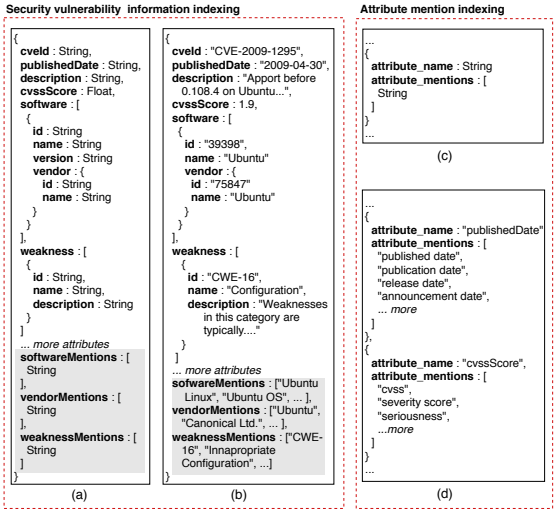

Fig. 2. Index-ready JSON representation of our security vulnerability information model (introduced in Figure 1(a)): (a) JSON schema of our information model (shaded attributes correspond to enrichments), (b) Example of a single document containing vulnerability information, (c) JSON schema for storing attribute mentions, (d) example of two attributes and their possible mentions

5 Security Vulnerability Information Querying with NL Support

In order to be able to answer NL queries on top of the index introduced in the previous section, we first need to understand the intent of users as expressed in their NL queries. In this section we discuss how this translation takes place (see the *Query Translator* component in Figure 1(b)). Since this work uses ElasticSearch for indexing and searching, we show how we do this translation into ElasticSearch's own DSL[11]. Figure 3 summarizes the steps of the translation process. For illustration purposes, we use in this section the following exemplary NL query: *What vulnerabilities are there in Ubuntu Linux, with a severity score of 10?*

Our translation operates on a subset of ElasticSearch's DSL to support attribute-based queries. In order to do this, we focus on two features from this DSL: Attribute selection and attribute-based filtering. *Attribute selection* is used to select attributes that will appear in the result set. Attribute-based filtering is used to specify query conditions that will help filter the entries to be retrieved from the index. Here, we support query conditions of the form *attribute:value*, where the semantics is that *value* must be contained in *attribute* in order for the condition to be satisfied. Next, we will discuss how NL queries are translated into this subset of DSL query.

[11] https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html

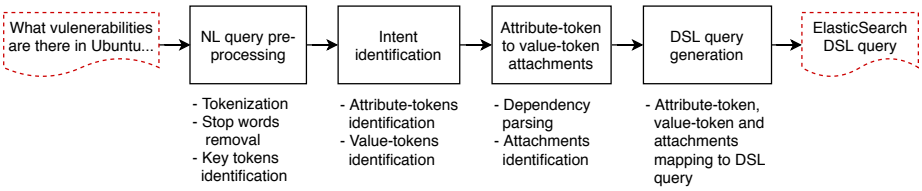

Fig. 3. Steps for NL to ElasticSearch's DSL query translation.

NL query pre-processing. The first step toward translating our exemplary NL query consists in dividing the query into tokens that carry the semantics of the user's NL expression. We use Stanford Core NLP Parser [11], which helps us identify tokens in the query. Once tokenization is performed, we proceed with removing stop words that do not contribute to the semantics of the NL expression. In our exemplary query, we remove the tokens *What*, *are*, *there*, *in*, *with*, *a* and *of*. This leaves us with the tokens *vulnerablities*, *Ubuntu Linux*, *severity score* and *10*, which we call *key tokens* (see Figure 4). These key tokens are used for identifying the intent expressed in the NL query.

Intent identification. Once key tokens are identified, the next task consists in recognizing the intent of the NL query. We consider two types of intents: (i) attribute selection, and (ii) attribute-value based filtering. In the first case, the user expresses his information needs by indicating the information item (attribute) he/she is interested in. For example, in our exemplary NL query, the user is asking about *vulnerabilities* in a software, which can be thought of as the list of *CVEs* (see the *cveId* attribute in Figure 2(a) and (b)) of vulnerabilities affecting such software. We address this intent by matching key tokens to attributes in our index. In the second case, we consider intents related to filter conditions that a user may express in an NL query. For instance, in our exemplary query, the user does not want just any vulnerability, but only the ones affecting the software *Ubuntu Linux*. We address this intent by identifying key tokens that refer to values stored in our index, which can be used as filtering conditions.

We identify the intents above by first focusing on intent type (i). We take each key token and try to match it to each of the atributes' mentions stored in our index (see, e.g., Figure 2(d)). If we find a match between a key token and an attribute mention (e.g., the key token *severity score* matches one of the mentions for the attribute *cvssScore*), we save the corresponding attribute name (e.g., *cvssScore*) for latter use and designate that key token as an *attribute-token* [13].

When a key token cannot be matched to any attribute mention, we try to match it to an attribute value of the index (thus, focusing on intent type (ii)). For example, the key token *Ubuntu Linux* does not match any attribute mention. However, it does match one of the mentions of the (enrichment) attribute *softwareMention* (see, e.g., Figure 2(b)). Since a mapping could be found to one of the attribute values, we designate *Ubuntu Linux* as a *value-token*, and we associate it to the corresponding attribute (*softwareMention*). As a result, the

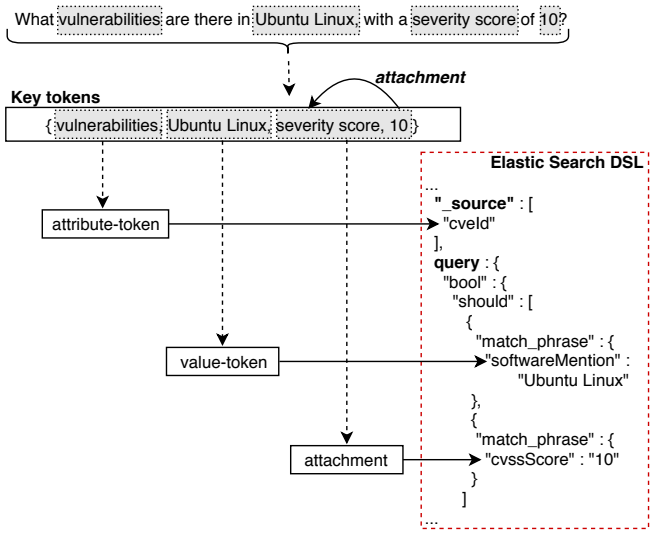

Fig. 4. Translating NL queries into ElasticSearch's DSL query.

overall intent identification task will yield a mapping of attribute-tokens to attributes names (e.g., *severity score* → *cvssScore*) and a mapping of value-tokens to attributes names (e.g., *Ubuntu Linux*→*softwareMentions*).

Attribute-token to value-token attachments. In some cases, users may indicate the intent of finding items that satisfy a condition explicitly expressed in the NL query. In our exemplary query, the user is asking about vulnerabilities in Ubuntu Linux, where such vulnerabilities have a given severity score (*severity score of 10*). In order to identify such conditions, we need to detect if any of the attribute-tokens is related to a value-token [13]. To do so, we use dependency parsing [11] techniques to detect such relations (also referred to as *attachments* [13]). In our exemplary NL query, the dependency parsing results indicate that the attribute-token *severity score* is attached to the value-token *10*. From this attachment, we infer that the attribute corresponding to the attribute-token *severity score* (the attribute is *cvssScore* in this case) should contain the value-token *10* (i.e., *cvssScore:10*). We use this attachment next to build query conditions using the search engine's DSL query.

DSL query generation. Given the mappings and attachments identified in the previous steps, we are now ready to generate the DSL query to be executed by the search engine. Figure 4 shows how we generate ElasticSearch's DSL query for our exemplary NL query. We follow three mapping rules for generating such DSL. In mapping rule *MR1* (attribute-tokens are used for attribute selection in ElasticSearch's DSL), we map the attribute names corresponding to attribute-tokens (without attachments) to the **_source** attribute of the DSL (see the mapping of *vulnerabilities* to *cveId* in Figure 4). Attribute names listed here are going to appear in the result set returned by ElasticSearch (this can be thought of as

the *projection* operation in relational databases). In mapping rule *MR2* (value-tokens *without* attachment are used as query conditions), we map value-tokens to conditions under the **query** attribute of ElasticSearch's DSL. For example, Figure 4 shows that *Ubuntu Linux* is translated to the condition *softwareMention*:"Ubuntu Linux" (because *"Ubuntu Linux"* was found as one of the values of attribute *softwareMention*). In mapping rule *MR3* (attachments are used as query conditions), we map attribute-tokens (*with* attachments) to the DSL's **query** attribute by creating a condition where the attribute (corresponding to the attribute-token) should contain the attached value-token. In Figure 4, this results in the condition *cvssScore:10*.

6 Evaluation

We conducted experiments with the objective of evaluating the feasibility of our approach to effectively answer NL queries related to security vulnerabilities. We present the details of the experimental setting, evaluation mechanism and results below.

Questions used in the evaluation. We use the questions from Smith et al. [16] for our evaluation. We adapted the questions listed in Table 1 to questions that are more contextualized to the information model supported by our solution (see the examples in Table 2). This adaptation is necessary in order to turn questions that make references to generic or vague information items into more concrete questions that can be answered with the information we support. For example, the reference to *"this vulnerability"* in question Q8 cannot be answered with our solution without providing a reference to a vulnerability (e.g., by using a CVE identifier or similar). We created variants for each of the AQs (e.g., using references to different vulnerabilities and software), which gave us a total of 65 variants (a mean of approximately 7 variants per AQ). We use these variants for the purpose of our evaluation.

Dataset. We collected security vulnerability information from Vulners and NVD (for vulnerabilities, weaknesses, discoverer, software and vendors), ExploitDB (for exploits), Breach Level Index (for attacks), and SecurityTracker (for patches). The dataset we use for evaluation consisted of a sample of approximately 102K vulnerabilities, 25K exploits, 33K patches, 21K software and 12K vendors affected, and 124 weaknesses. These sources were integrated and enriched as discussed in previous sections.

Implementation. The proposed solution was implemented based on the architecture shown in Figure 1(b). The adapters for the sources listed above were implemented using Python 2.7. For enrichment, we leveraged on ConceptNet (as explained earlier) and word embeddings trained on Information Security Stack Exchange. We trained the model using Word2Vec [12] and a skip-gram model with negative sampling (sampling rate of 10 words), 300 dimensions and context word window of 5. We used ElasticSearch 5.5.2 as our index and search engine. Tokenization, dependency parsing and named-entity recognition were done using

Table 2. Examples of adapted questions. The questions in bold font are the original questions (Q) from [16], while questions in regular font are examples of adapted questions (AQ). We used a total of 65 variants of these AQs for the evaluation.

Examples of adapted security vulnerability questions
Q1: "Is this a real vulnerability?"
AQ1: "What are the details of vulnerability CVE-2004-1305?"
Q2: "What are the possible attacks that could occur?"
AQ2: "What are the possible attacks on Firefox?"
Q3: "How can I prevent this attack?"
AQ3: "How can I prevent attacks exploiting shellshock?"
Q4: "How can I replicate an attack to exploit this vulnerability?"
AQ4: "Is there any exploit for vulnerability CVE-2004-1305?"
Q5: "What is the problem (potential attack)?"
AQ5: "What's the weakness in HeartBleed?"
Q6: "What are the alternatives for fixing this?"
AQ6: "Is there a workaround to protect against Dirty COW?"
Q7: "How do I fix this vulnerability?"
AQ7: "What are the patches to remediate CVE-2017-3561?"
Q8: "How serious is this vulnerability?"
AQ8: "What's the severity of shellshock?"
Q9: "Are all these vulnerabilities the same severity?"
AQ9: "Whats the severity of vulnerabilities CVE-2017-3561 and CVE-2017-3563?"

Stanford Core NLP 3.8. The NLI and query translator were implemented using Python 2.7.

Expert evaluation. We fed the 65 AQs discussed before to our solution and obtained the corresponding answers. Such answers were evaluated by a domain expert in the area of cybersecurity who judged whether the answers provided by our solution satisfy the NL query in input. Since the results returned by ElasticSearch are ranked using a TF/IDF-based scoring system, we use the metrics R-Precision and Precision@n (or precision at level n), which are typically used in Information Retrieval [15]. R-precision is computed as $TP/|Rel|$, where TP is the number of true positives and $|Rel|$ is the total number of relevant results for a given query (here, TP is computed only for the top $|Rel|$ answers). For queries with a potentially large number of results we employ Precision@n. This metric is used to compute relevant results on the top n answers, which is useful in scenarios where end-users are typically interested only on the top results (e.g., web search) [15]. Precision@n is computed as $P@n = TP/n$, where TP is the true

Table 3. Evaluation results. We report on average values for $|Rel|$, R-Precision and P@10. P@10 is computed only for questions with a potentially large $|Rel|$ [15].

Question	**|Rel|**	**R-Precision**	**P@10**
AQ1	1.30	1.00	
AQ2	265.40	0.93	1.00
AQ3	1.60	0.85	
AQ4	1.00	1.00	
AQ5	1.17	1.00	
AQ6	1.60	0.85	
AQ7	1.10	1.00	
AQ8	1.17	1.00	
AQ9	2.00	1.00	
Avg.		**0.96**	

positives and n corresponds to the number of top results to be considered (here, TP is computed only for the top n results). In this evaluation we consider the top-10 results and therefore compute Precision@10.

Results and discussion. The results of our evaluation is presented in Table 3. Most questions have a low $|Rel|$, except for question Q2. High $|Rel|$ values were observed whenever an NL query contained value-tokens that can appear among the values of attributes inside entries that are not relevant to an NL query. In AQ2 the expert expected to get vulnerabilities related to Firefox browser only. The system returned, however, also vulnerabilities affecting Mozilla Firefox OS. Yet, Table 3 reports a Precision@10 of 1.00. Questions AQ1, AQ4-5 and AQ7-9 report high R-Precision values. This stems from the nature of the corresponding questions, which inquire information about specific vulnerabilities, attacks and exploits (e.g., *CVE-2004-1305* and *ShellShock*) without much ambiguity (as opposed to AQ2). Questions AQ3 and AQ6 ask, essentially, the same question, but using different terminology. We obtained the same results and thus we report the same $|Rel|$ and R-precision values. The R-precision have in both cases the lowest values accross the various AQs.

The proposed solution and evaluation come with their own ***limitations***. Range queries (e.g., *vulnerabilities with severity between 7 and 10*) and questions that imply *Yes/No* answers are not supported in the current version. While our solution cannot provide answers for the former, in the latter case the answer provided is either an empty result set (for "No" answers) or a list of results (for "Yes" answers). In addition, questions involving aggregate functions such maximum / minimum values (e.g., *what's the latest vulnerability in Ubuntu?*) and sums / counts (e.g., *how many vulnerabilities are there in FreeBSD?*) are not currently supported. The same applies for comparison operators such as $>$ and $<$. In addition, our evaluation focuses only on the set of questions obtained from [16] and involves only one expert evaluator. More thorough evaluations are needed with a larger and more varied set of questions, involving also pilot users and additional evaluators. We plan to expand our evaluation in this direction.

7 Conclusion and Future Work

This paper proposes an approach and architecture for supporting the exploration and understanding of security vulnerabilities. Our approach stems from the pressing needs for a unified, integrated and easy-to-query security vulnerability information platform that helps businesses mitigate the threats from the growing number of security vulnerabilities. The NL query capability of the proposed solution makes it a good candidate for integration into productivity tools used in software development and devops environments (e.g., through chatbots), which can help bring security vulnerability information seamlessly into context and while performing core development and devops tasks.

Directions for future work include the development of domain-specific ontologies and knowledge graphs for supporting more complex queries beyond attribute-based queries (e.g., relationship-based queries), further enrichments

using intelligent information taggers that leverage on advancements in NLP and AI, and the use of alternative sources (e.g., Twitter) for obtaining updates on the latest cybersecurity developments (e.g., 0-day vulnerabilities and attacks).

Acknowledgement. We acknowledge Data to Decisions CRC (D2D-CRC) for funding this research.

References

1. Al-Banna, M.: Crowdsourcing Software Vulnerability Discovery: Expertise Indicators, Organizations Perceptions and Quality Control. Ph.D. Thesis, Computer Science and Engineering, Faculty of Engineering, UNSW (2018)
2. Coronel, C., Morris, S.: Database systems: design, implementation, & management. Cengage Learning (2016)
3. Darabal, K.: Vulnerability Exploration and Understanding Services. Master Thesis, Computer Science and Engineering, Faculty of Engineering, UNSW (2018)
4. Ferrara, E., De Meo, P., Fiumara, G., Baumgartner, R.: Web data extraction, applications and techniques: A survey. Knowledge-based systems **70**, 301–323 (2014)
5. Hirschberg, J., Manning, C.D.: Advances in natural language processing. Science **349**(6245), 261–266 (2015)
6. Hitzler, P., Krotzsch, M., Rudolph, S.: Foundations of semantic web technologies. CRC press (2009)
7. Kampanakis, P.: Security automation and threat information-sharing options. IEEE Security & Privacy **12**(5), 42–51 (2014)
8. Kaufmann, E., Bernstein, A., Zumstein, R.: Querix: A natural language interface to query ontologies based on clarification dialogs. In: ISWC. pp. 980–981 (2006)
9. Li, F., Jagadish, H.V.: Nalir: an interactive natural language interface for querying relational databases. In: ACM SIGMOD. pp. 709–712. ACM (2014)
10. Lopez, V., Fernández, M., Motta, E., Stieler, N.: Poweraqua: Supporting users in querying and exploring the semantic web. Semantic Web **3**(3), 249–265 (2012)
11. Manning, C., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S., McClosky, D.: The stanford corenlp natural language processing toolkit. In: ACL. pp. 55–60 (2014)
12. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. In: Advances in neural information processing systems. pp. 3111–3119 (2013)
13. Popescu, A.M., Etzioni, O., Kautz, H.: Towards a theory of natural language interfaces to databases. In: IUI '03. pp. 149–157. ACM (2003)
14. Pruski, P., Lohar, S., Goss, W., Rasin, A., Cleland-Huang, J.: Tiqi: answering unstructured natural language trace queries. Req. Eng. **20**(3), 215–232 (2015)
15. Schütze, H., Manning, C.D., Raghavan, P.: Introduction to information retrieval, vol. 39. Cambridge University Press (2008)
16. Smith, J., Johnson, B., Murphy-Hill, E., Chu, B., Lipford, H.R.: Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In: ESEC/SIGSOFT FSE 2015. pp. 248–259. ACM (2015)
17. Speer, R., Havasi, C.: Representing general relational knowledge in conceptnet 5. In: LREC. pp. 3679–3686 (2012)
18. Tablan, V., Damljanovic, D., Bontcheva, K.: A natural language query interface to structured information. In: ESWC. pp. 361–375. Springer (2008)
19. Zhao, M., Grossklags, J., Liu, P.: An empirical study of web vulnerability discovery ecosystems. In: ACM SIGSAC. pp. 1105–1117. ACM (2015)