An Approach to Generate Realistic HTTP Parameters for Application Layer Deception

Merve Sahin, Cédric Hébert, and Rocio Cabrera Lozoya

SAP Security Research

Abstract. Deception is a form of active defense that aims to confuse and divert attackers who try to tamper with a system. Deceptive techniques have been proposed for web application security, in particular, to enrich a given application with deceptive elements such as honey cookies, HTTP parameters or HTML comments. Previous studies describe how to automatically add and remove such elements into the application traffic, however, the elements themselves need to be decided manually, which is a tedious task (especially for large-scale applications) and makes the adoption of deception more cumbersome.

In this paper, we aim to automate the generation of deceptive HTTP parameter names for a given web application. Such parameters should seamlessly blend into application context and be indistinguishable from the rest of the parameters, in order to maximize the deception effect. To achieve this, we propose to use word embeddings trained with a domain-specific corpus obtained from existing web application source code. We evaluate our method through a survey, where we ask the participants to identify the deceptive parameters in two different web applications' APIs. Moreover, the survey is composed of two variants in order to further experiment with the impact of the quantity and enticement of deceptive parameters.

The results confirm the effectiveness of our method in generating indistinguishable honey parameter names. We also find that the participants' expectation of the ratio of honey parameters remains constant, regardless of the actual number. Thus, a higher number of honeytokens can provide a stronger defense. Moreover, making attackers aware of deception can help to obfuscate the real attack surface, e.g., by masquerading more than 10% of the real application elements to look like traps. Finally, although our work focuses on the generation of parameter names, we also discuss other related challenges in a holistic way, and provide multiple directions for future research.

Keywords: web application security, deception, active defense

1 Introduction

As part of a defense-in-depth strategy, deception works by confusing and misleading the adversary with false information, while *masking* the real nature of a system, or *repackaging* it to look like something else [15,18]. Various studies have

shown that deception can be an effective defense mechanism, not only for attack detection [43,64,19], but also for impeding the attack progress and disrupting attackers' emotional and cognitive state in various ways [35,34,32]. Moreover, deception technology market has been growing in recent years [6,7], with several commercial solutions providing data, network, application or endpoint layer deception [70,36,21,47,13].

The focus of this study is on web application layer deception. So far, the main idea has been to augment the application with deceptive elements (also called *honeytokens*, which can be in the form of HTTP parameters, cookies, HTML elements, permissions, or user accounts) in order to showcase a fake attack surface [56,43,39,37,44,59,38]. Monitoring the modifications to the values of such deceptive elements allows to detect attackers who are tampering with the application in order to find vulnerabilities. For instance, a common attack vector is called *web parameter tampering*, where the attacker manipulates the application parameters exchanged between the server and client, in an attempt to modify privileges, get access to unauthorized information, exploit business logic vulnerabilities, or disrupt the integrity of the application data [23,55]. The attacker may tamper with an object ID in the URL parameter to exploit an improper access control mechanism (known as the Insecure Direct Object Reference vulnerability [61]); or try to modify, e.g, the price of a product sent in a hidden form field, which was assumed to be immutable by the developer [68,55]. The use of deceptive elements provide a reliable source of warning in such cases, as the regular users of the application are not likely to intercept the communication and try to tamper with application data.

Most of the previous work on application layer deception focuses on how to add the deceptive elements with minimal effort. They use a reverse-proxy in front of the application that adds and removes the deceptive elements on the fly, seamlessly, so that the application itself will not require any modifications [39,37,43]. Previous work also conducts CTF based experiments to measure the effectiveness of application layer deception [43], including when the attackers are aware of the presence of deception [64].

While these studies focus on automating the injection of deceptive elements into the application, they do not really address the challenges related to the generation of such elements, leaving this as an open research problem. In fact, a survey on deception techniques in computer security [44] draws attention to the lack of proper honey-token generation strategies for web applications and cloud images. Other studies emphasize the need to create "content-oriented deceptions to deceive skilled attackers in the long term" [32] and draw attention to the difficulty of creating such context-specific elements [43]. Previous work also finds that deceptive elements should be well intertwined with the application functionality and logic, to be robust against the deception awareness of the attacker [64]. In this paper, we address this research area of automatically generating realistic deceptive elements for web applications. In particular, we focus on the automated generation of *deceptive HTTP parameters*, as they can be effec-

tively used in every API endpoint, covering a large attack surface of parameter tampering.

Deceptive HTTP parameters can be any type of HTTP parameter (such as the query, path, body or form parameters). However, coming up with context-specific deceptive parameters and embedding them into the application seamlessly accompany multiple challenges:

- How to choose realistic names for the parameters?
- How to make sure that the parameters are enticing enough?
- How to assign them plausible values?
- Where to place the parameters within an API?
- What is the optimal number of deceptive parameters?
- What should be the proper response when a certain parameter is tampered with?

Thus, *our first contribution in this paper will be to explore these six challenges, discuss the previously used strategies and other possible approaches (Section 2).*

We then focus on the first challenge, which is to automatically generate plausible deceptive parameter names that are difficult to distinguish from the real parameters. For this, *we implement a machine learning method to generate parameter names that will blend well into the context of a given application (Section 3).* In particular, we use word embeddings (a Natural Language Processing technique) trained with the source code of publicly available web applications.

Finally, *we evaluate the effectiveness of our method via a survey with 42 participants (Section 4).* With a questionnaire, we ask the participants to identify the deceptive parameters in two different web applications' APIs. *Our questionnaire also experiments with two additional challenges: the amount and enticement of deceptive parameters (Section 5).*

In addition to showing that our method successfully generates indistinguishable parameter names, we make several other observations: First, we find that the participants anticipate a certain ratio of parameters to be deceptive, regardless of the actual quantity of deceptive elements. Thus, adding a larger number of deceptive elements would mean that more of them will go undetected. Second, the addition of very obvious (conspicuous) deceptive parameters does not really help to hide the existence of realistic ones. Third, we find that the participants mislabel at least 10% of genuine parameters as deceptive, on average. This provides another evidence on the benefit of informing attackers about the use of deception.

2 Challenges

One requirement for successful deception is that it "should present plausible alternatives to the truth" [15]. This becomes even more important considering that the attacker might be aware of the presence of deception. In this section, we will discuss the challenges related to the generation of plausible and effective deceptive parameters.

2.1 Generating context-specific, realistic parameter names

A first step to generate indistinguishable deceptive parameters would be to decide on the parameter names. Ideally, at least three requirements should be met: (i) parameter name should fit in the context and purpose of the API endpoint, (ii) it should indicate a certain functionality and, (iii) it should follow the naming convention and format of the API.

Previous studies doing CTF-based experiments manually select the parameter names relevant to the content of the web application [43,64], an approach that is difficult to scale for real-world, large applications. The most relevant study on this topic by Pohl et al. [59] aims to generate honey HTML form field names for a given application. The approach of this study is to collect the form fields (names and the default values, if available) by crawling the Alexa Top 10,000 websites. Authors extract 15,255 forms and 18,210 form field names. Then, they propose an algorithm based on the Levenshtein Distance as a similarity metric, to pick the most suitable honey form field for a given set of form elements. The method is evaluated via 75 human subjects, where the participants are shown 50 HTML forms, knowing that half of them are containing an injected deceptive form field. Results show that participants' choices are significantly near random, which means they cannot identify the forms with deceptive fields. However, this approach has several limitations: First, it remains limited to the form parameters, not addressing other types of HTTP parameters. Second, as the crawler does not have the ability to authenticate, the form fields collected in the wild are only limited to the pre-authentication pages. Considering the difficulty of automating the account creation and authentication processes on arbitrary web sites, this approach is not scalable to explore the complete application context.

Overall, it remains an open problem to automatically generate different types of deceptive HTTP parameters that would cover a large part of the application [43]. In this paper our main focus is to address this challenge.

2.2 Ensuring the enticement of deceptive parameters

Generating a realistic parameter name does not necessarily mean that the parameter will be enticing for the attacker to tamper with. For instance, a parameter named "street" might be less effective in attracting (and thus detecting) attacks compared a parameter named "is_admin". A possible strategy to generate enticing parameter names can be to simulate some of the bad REST API practices. For instance:

- Parameters that seem to provide security (e.g., authentication, authorization) related configurations.
- Parameters that seem to modify data that normally should not be provided by the user (e.g., the price of an item).
- Parameters that seem to overwrite configurations or provide functionality that affects page behavior (e.g., *mode=readonly* as a query parameter).

However, more research is needed to better understand what makes a parameter enticing, how to measure the quality of enticement, how does this relate to attack detection rate, and how to find the optimal level of enticement before a parameter becomes too obviously deceptive.

Although it is not the focus of our study, we partially touch on this topic by manually adding a set of conspicuous parameters (i.e., easily noticeable or obvious to the attacker, as they are either too enticing or out-of-context of the application) to one of the APIs used in our experiment. We will explain more on this in Section 4.1. On the other hand, the deceptive parameters generated with our machine learning approach can have different levels of enticement: while some of our parameters might look more enticing (e.g., "bearer"), others might look like ordinary parameters (e.g., "retry").

2.3 Assigning default values

Each deceptive parameter that is added to the application should have a plausible value, either static or changing dynamically. Depending on the parameter type, the value could be, for example, an integer, a string with a fixed set of possible values, or an array of values [9]. For instance, for a parameter named 'superuser', possible values can be 'true'/'false', 'enabled'/'disabled', or a numerical user identifier.

Note that the assigned value can also affect how enticing a parameter is: For instance, assigning a random looking value (such as a hash or UUID) might make it less enticing for the attacker, as tampering with the value would likely break the application.

Previous studies [43,64,59] do not mention a specific strategy on assigning values to deceptive elements. Thus, additional research is needed to explore this topic. One approach can be to collect API documentations in the wild to create a database of parameter names and value constraints. Research on automatically inferring single parameter and inter-parameter constraints from API documentations combined with static code analysis (such as [72,42]) might also be useful to enrich such a database. Note that, our work does not address this challenge.

2.4 Placement of deceptive parameters

As stated by Han et. al, [44], placement strategies for deceptive elements is an under-studied topic for most of the deception techniques, and for the application layer deception, a major obstacle is the difficulty to characterize the web application logic.

An approach used in previous work [43] is to refer to the OWASP pentesting guide [51] to deploy honey elements in places where the attackers look in priority to find vulnerabilities. In the context of deceptive HTTP parameters, it is likely that targeting the security-sensitive parts of the application (e.g., security-relevant state changing requests as defined in [29]) would be more effective in

attack detection. On the other hand, all the parameters are open to fuzzing and tampering, in case they would lead to e.g., injection vulnerabilities.

An important point is that, the coherence between the API endpoints must be taken into consideration while placing the deceptive parameters. Moreover, they should follow the application logic, e.g., if an endpoint creates a new resource (e.g., via POST method) with a deceptive parameter injected, the endpoint that updates the resource (e.g, via PUT or PATCH method) should also include this parameter. In future work, studies that infer producer-consumer relationships between API endpoints (such as [20]) can help to improve the placement strategy of honeytokens.

In this study, we deploy the deceptive parameters depending on where our machine learning method performs the best. As we will explain in Section 3, the parameters generated by our approach are coherent between different API endpoints, as the machine learning algorithm returns the same output values for the same set of input values.

2.5 Quantity of deceptive elements

A previous study suggests that a high quantity of deceptive elements might tip off the attacker about deception and therefore reduce its effectiveness [43]. Another study [64] indeed finds that the effectiveness of deceptive elements might reduce when the attacker is aware of deception, however, this awareness brings additional impact such as pushing attackers to deviate from their regular attack strategies.

In this study we also touch on this topic for a preliminary evaluation on the impact of the number of deceptive parameters. More details about the experiment and the results will be given in Sections 4 and 5.

2.6 Response strategy

For a robust deception effect it is important to design proper response actions against parameter tampering, which will give the feeling that the deceptive element is a real, functional part of the application [64]. Possible approaches can be to show fake error pages (e.g., faking an SQLi/LFI vulnerability [43]), fake authentication pages, or returning HTTP status codes such as service unavailable or bad request, depending on the type of the parameter. More research is needed to automatically determine a realistic response strategy for a given deceptive parameter.

3 Our Approach

As mentioned earlier, we aim to automate the generation of deceptive HTTP parameters that are in agreement with the context of the web application to be protected. For such tasks, the Natural Language Processing (NLP) domain

offers different techniques, such as specialized lists, lexical dictionaries and word embeddings.

Specialized lists have the drawback of needing to be handcrafted, a process which can be time-consuming and requires domain-specific knowledge. Lexical dictionaries have been used in traditional NLP approaches. They are networks of meaningfully and semantically related words and concepts (synsets) and provide graph representations of the relationships of a vocabulary. Nevertheless, lexical dictionaries might not be able to keep up with the quick evolution of the language, as well as with domain-specific jargon.

Finally, *embeddings* are vectorial representations of words mapped onto a reduced dimensionality space where similar words (embeddings) are close to each other. The distance between these embeddings is often measured using the cosine similarity or any other distance between vectors. Due to their data-driven nature, embeddings are able to capture the relationships between words in specific contexts. They were initially popularized in the recent years due to their applications in NLP by using rather simple neural network architectures, such as the ones proposed by word2vec [52] and GloVe [57]. More complex language models have been developed in the last years (e.g. ELMo [58], BERT [33], ALBERT [48], RoBERTa [50]), many of which are available for download and offer quick interfaces for their use [63]. Nevertheless, these models are often trained on vast English text corpus coming from natural language sources as varied as news articles, Wikipedia entries, literary [74,71] and web content [62] among others. While they are useful for generic language understanding tasks, they can struggle with applications which contain a big domain-specific vocabulary.

For this reason, some studies have concentrated their efforts in generating application-specific language models, including those for biomedical [49], clinical [17,46] or financial [69] applications. Outside the realm of natural languages, embeddings have also been used to model programming languages both in a sequence-of-tokens fashion (supported by the *naturalness hypothesis* [14]) or by embedding elements in graph representations of code (e.g., abstract syntax trees or control flow graphs) [16,28,31,24].

In this paper, we propose the use of embeddings of source code by treating it in a sequence-of-tokens fashion. The choice was made to use the smaller and simpler models like *word2vec* due to their less data-hungry nature and their ease to train them compared to more complex models. Due to the very specific nature of our application, we train our language model with a dataset specifically created for this task.

3.1 Data Collection and Training

In order to create a domain-specific dataset that will capture the terminology and technical context of web applications, we use the source code of web applications available at public GitHub repositories. We start with a list of public GitHub Java repositories with more than 5 stars (watchers), which was made available by Chen et al. [30] and includes 83,082 repository URLs collected from GHTorrent [41] database (last updated on 2019-06-01). Among these, we remove the

repositories that include `android` or `mobile` keywords in the repository name, and focus on the repositories with at least 15 stars, which reduces the list to 38,376 URLs.

As our purpose is to generate HTTP parameters for web APIs, we try to limit the training data to the repositories with web application relevant source code. We do this in a coarse-grained way, by pruning the dataset to only contain the repositories that include web related libraries: We download each repository and look for "import" statements for library names such as `org.springframework.web`, `javax.servlet`, `org.apache.http`, `httpcomponents` and `okhttpclient`. Finally, we end up with the source code from 10,324 repositories, which corresponds to 4,002,776 Java files.

We further refine the Java files according to their name: The files that are likely to not have a context related to the functionality of the application (e.g., `util, filter, exception, config, parser, test`) and the files that might have a too specific context (e.g., `coin, blockchain, droid, Activity`) are removed.

For each project, for each remaining Java file, we parse the file (using the `JavaLangParser` Python library) to extract the relevant input to train the *word2vec* model. While *word2vec* is normally trained with sentences from natural language, we construct the sentences as sequence-of-tokens collected from the source code. In particular, each of the following items forms a separate sentence by appending the relevant tokens together:

- Each method name (MethodDeclaration) and the names of method parameters
- Each class constructor (ConstructorDeclaration) and the names of constructor parameters
- The names of the class fields (FieldDeclaration)
- All the variable names in the class (VariableDeclaration)

The motivation is that each of these sentences includes tokens that are likely to belong to the same context. Note that, each token (method, parameter, variable, or constructor name) is split by underscore or camel case (if such naming convention was used), and then converted to lower case. An example of Java source code and the set of sentences extracted from it can be found in Appendix A.1.

Post-processing: In each sentence, we remove the tokens that are specific to the Java language, and tokens that do not carry any contextual meaning.[1] Finally, we train the *word2vec* model using the Python *gensim.models* library, with the default parameters.

Independently from this process, we also save all the variable names with built-in types (e.g., String, boolean, int, array) for each project in a seperate csv file. This corresponds to 8,844,562 variables. We later use this data to find the most suitable parameter type for the generated deceptive parameter names.

[1] These words are `has, have, init, start, stop, get, set, main, create, delete, update, read, add, remove, is, on, by, to, test, parse, write, initialize, string, int, boolean, char`.

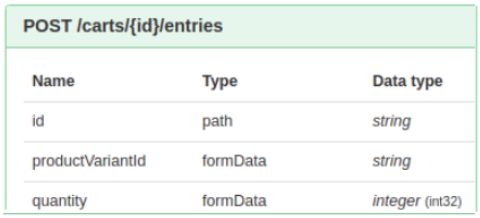
3.2 Generation of parameter names

To generate deceptive parameters for a target application, we assume to obtain the API specification of the application to start with. In particular, we assume to have an *OpenAPI specification* [54] as input. OpenAPI (formerly known as *Swagger* [67]) specification aims to standardize the descriptions of RESTful APIs. In addition, the Swagger project provides various tools for testing and development, together with a specific user interface to view and try out the API (called Swagger UI [12]). In our experiments, we use an alternative Swagger user interface called Bootprint [8], as it outputs a static HTML page with a simpler design that is more appropriate for our purpose.

Once we have the Swagger specification (which is often in json or yaml format), we flatten [66] the file and convert it to the csv format, where we have each HTTP parameter and the related information (endpoint, HTTP method, name and type) in one row. Note that, endpoints may pair with multiple different HTTP methods, and each endpoint-method pair is likely to have multiple parameters. Figure 1 shows an example API specification of an endpoint-method pair in json format (a), together with how it looks on Bootprint-Swagger UI (b) and our conversion to csv format (c).

```
"/carts/{id}/entries": {
  "post": {
    "operationId": "postCartEntry",
    "parameters": [{
        "type": "string",
        "name": "id",
        "in": "path"},
      {
        "type": "string",
        "name": "productVariantId",
        "in": "formData"},
      {
        "type": "integer",
        "name": "quantity",
        "in": "formData"
      }]}}
```

(a) Swagger json file for the endpoint-method pair



(b) Swagger UI

Endpoint	Method	Location	Name	Type
/carts/id/entries	post	path	id	string
/carts/id/entries	post	formData	productVariantId	string
/carts/id/entries	post	formData	quantity	integer

(c) API endpoint converted to csv

Fig. 1: Example Swagger input for the *POST* method of */carts/{id}/entries* endpoint.

Next, we use our *word2vec* model to generate deceptive elements for the endpoint-method pairs in the API. In particular, we use the `most_similar()` method of *word2vec* library to get the top n words that are the most similar to a list of *existing elements*. We form the *existing elements* list depending on the location of the HTTP parameter:

– **Path parameters:** Path parameters are located in the URL path of an endpoint, and often point to a specific resource [9]. We only attempt to embed a deceptive path parameter to the endpoints that already have at least one path parameter. First, we group the endpoints up to their very first path parameter. Next, within each group, we collect all the endpoint URLs, split the words by underscore or camel case if necessary, and form our *existing elements* list. If this approach does not yield any proper output (due to the thresholds that we will explain later), an alternative approach is to collect the first level URL components of all endpoints as the *existing elements* list. Note that, the generated deceptive path parameter will be added to all the endpoints in this group, to keep the consistency of parameters between endpoints.
– **Query or body parameters:** Query parameters are located at the end of the URL, after a question mark (e.g., '?name1=value1&name2=value2' format). Describing the body parameters, on the other hand, is a bit more complex: The earlier version of OpenAPI (v2) differentiates between the `formData` parameters that describe the payload of a request, and the `body` parameters that describe an object with a data structure [10]. However, the last version (OpenAPI v3) categorizes both of them under the `RequestBody` type [11].

We process the query and body parameters, for each of the endpoint-method pairs: We take the existing query or body parameter names, in addition to the tokens from the URL path of the related endpoint (again splitted by underscore or camel case, if necessary). Note that, as our *word2vec* model is deterministic once it is trained, it will generate the same output for the same set of *existing elements*. This allows us to preserve the consistency between different endpoints: For instance, two different endpoints that update an address object with the same body parameters will also be assigned the same deceptive parameter.

Finally, our algorithm aims to insert deceptive parameters only if it has a high 'confidence' that the generated element will fit in the context of the endpoint-method pair. For this, we implement the following four steps:

(i) *Making sure that the existing elements list contains sufficient input:* For query and body/form parameters, we set a threshold for the minimum number of *existing elements*: If there are fewer elements than this threshold, we choose to not generate a deceptive parameter for the given endpoint-method pair and parameter location. For path parameters, we also require a certain *number of endpoints* per group, to be able to assign a deceptive parameter to this group.
(ii) *Making sure that the input words are known to the model:* It is possible that some of the *existing elements* will not be present in the vocabulary of our *word2vec* model, as our training dataset may not contain them. Thus, our

second threshold becomes the minimum *known_words_ratio*: the ratio of *existing elements* that are present in the vocabulary. If these two thresholds are met, we take the top n most similar words as our *candidate deceptive parameters*.
(iii) *Post-processing to check if the candidate parameters have sufficient similarity to existing elements:* We compute the average similarity score of candidate parameters to the *existing elements*. (The similarity scores are returned by the `most_similar()` method.) If this value is less than our *average_similarity_score* threshold, we choose to not insert any deceptive parameter for this endpoint-method pair and parameter location.
(iv) *Post-processing to avoid repeating parameters:* Finally, we remove the candidate deceptive parameters that are morphologically too close to any of the existing elements. For example if "paymentid" is an existing element and our model generates "paymentnum", we remove "paymentnum" from the candidate list. For this, we use the `ratio()` method from the `difflib.SequenceMatcher` [3] class in Python, to compute a measure of similarity between two sequences. We set a *sequence_matching_score* threshold to decide whether a candidate should be removed. After all these steps, the final deceptive parameter becomes the first element in the *candidate deceptive elements* list, having the highest similarity score value.

Fine tuning the algorithm: We tried our algorithm on 17 real-world Swagger API documentations that we collected online (using Google dorks such as intitle:"swagger.json" site:github.com). Note that, we make sure that the collected APIs do not overlap with the GitHub repositories used in our training. By experimenting with these APIs to generate deceptive elements, we come up with a set of threshold values that provide a good starting point. Table 1 gives these threshold values, which we also use for evaluating the performance of our method in the next section.

On a final note, we also assign a type (e.g., int, boolean, string) to the generated deceptive elements using the dataset of more than 8 Million variables collected in Section 3.1. To infer a type, we first look for an exact match between the generated parameter name and the variable names dataset. If it does not exist, we again use the `SequenceMatcher` class with a similarity score threshold of 0.8. This algorithm was able to infer a type for almost all of the parameter names in our initial experiments.

In Appendix A.2, we provide examples of the deceptive parameters generated by our model for two different endpoint-method pairs.

Threshold	Value	Threshold	Value
n	5	known_words_ratio	>0.7
number of endpoints	≥2	average_similarity_score	>0.6
number of existing elements	≥2	sequence_matching_score	>0.5

Table 1: Thresholds and values that affect the generation of parameters.

4 Evaluation

In this section we aim to evaluate the performance of our method in generating indistinguishable deceptive parameters. A common evaluation method that is also used in previous work [25,60] is to ask human subjects to differentiate deceptive elements from genuine elements. However, while in the previous work the human subjects were informed upfront that 50% of elements they will evaluate are deceptive, we do not give any tips about the number of deceptive parameters. Moreover, we present the subjects with real-world APIs using the Bootprint Swagger UI, so that they can get a sense of the application and thoroughly observe all the endpoint-method pairs, parameters and their types. We use a separate questionnaire where we list all the distinct parameter names (categorized by location such as query, form, path), and ask the subjects to mark the parameters that they think are deceptive.

Although this evaluation method does not allow participants to interact with a running instance of the application, it allows them to really focus on the names of the parameters. In fact if they were able to interact with the application, they could rely on additional criteria (e.g. value of the parameter, response to tampering) to decide if a parameter is deceptive or not. Thus, presenting the participants with a static API specification better fits our purpose of evaluating the indistinguishability of parameter names.

4.1 Preparation of the API specifications

For this experiment, we choose two APIs among the set of 17 real-world APIs mentioned in the previous section, following the below criteria:

- The APIs should have more or less equal number of endpoints and parameters to achieve more reliable results in statistical tests.
- The applications' context should be easy to grasp so that the participants can make more informed decisions (i.e., reducing the randomness that might emerge from not understanding the API).
- The number of API parameters should be reasonable for manual evaluation; to not overwhelm and distract the human subjects, and to make the survey feasible to complete in a reasonable amount of time.

In particular, the two APIs we choose include (i) a cloud integration API for an e-commerce application [1], and (ii) a community based laboratory platform for various professions [4]. The first one has 63 distinct parameters and 38 endpoint-method pairs, and the second one has 74 distinct parameters and 33 endpoint-method pairs.

To prepare the APIs for the experiment, we first anonymize the specification, removing the application name, all descriptions, and fields ignored by our study such as response status[2]. Then we generate the deceptive parameters using the

[2] Full list of fields removed from Swagger: "info", "description", "host", "tags", "summary", "responses", "definitions", "enum" , "example", "security", "securityDefini-

method described in previous section. We insert the generated parameters back into the Swagger specification, so that the Bootprint Swagger UI can display them. Our algorithm generates 8 deceptive parameters for e-commerce and 9 for the laboratory platform. We will call this the *default mode*, and will denote the APIs as *E-Commerce*$_D$ and *Lab-Platform*$_D$, respectively.

In addition, our experiment also aims to measure the effect of (i) high quantity and (ii) conspicuous (i.e., easily visible, obvious, attracting attention [27]) deceptive parameters. For this, we decide to divide the participants into two groups: Each group is presented 2 applications, one of it with the default mode, and the other with one of the additional characteristics (either higher quantity of parameters, or very conspicuous parameters added). Table 2 shows various statistics on the number of distinct deceptive parameters and affected endpoint-method pairs for the API variants used in our experiment.

High quantity of deceptive parameters: We apply this variant to the e-commerce application, denoted with *E-Commerce*$_Q$. To have a significantly higher number of deceptive parameters compared to the default mode (which is *E-Commerce*$_D$), we first use the additional results generated by our model (i.e., more parameter names from the *candidate deceptive parameters* list). With this, we obtain 6 more parameters in addition to the 8 parameters generated in default mode. However, while we want this API variant to have statistically significantly higher number of deceptive elements, our model was not able to generate that many parameter names, as we apply several thresholds to choose the best candidates. Thus, we have added 15 additional, manually chosen realistic parameters.

To show that *E-Commerce*$_Q$ has significantly more deceptive parameters compared to *E-Commerce*$_D$, we employ two-proportions Z-tests: Looking at the ratio of (the number of distinct deceptive parameters) / (the total number of distinct parameters), we find a z-score of -3.0609 and p-value of .00222. Thus, the result is significant at a confidence level of 95%. Moreover, in terms of the ratio of (the number of endpoint-method pairs with deceptive elements) / (the total number of endpoint-method pairs), we also show a statistically significant difference (z-score=-1.9742, p-value=.02442, significant at $p<.05$).

Conspicuousness of deceptive parameters: We use this API variant in the laboratory platform application, denoted as *Lab-Platform*$_C$. To manually add conspicuous deceptive parameters to the API (in addition to the realistic ones), we use two different strategies:

- Parameters that look too enticing and do not follow the naming convention of the application (e.g., use of camelcase instead of underscore, uppercase letters): Examples are MakeAdmin, FullPrivileges, ADMIN_PERM, cl4ssifi3d_ID.
- Parameters that do not have any meaning or that are out-of-context of the application: Examples are yoyo, pysantx, vv, disclosed.

---

tions", "x-example", "minimum" , "maximum", "readOnly", "maxLength", "minLength" , "pattern", "required"

To make sure that the parameters are indeed conspicuous, we made an initial evaluation on 7 participants, presenting them a preliminary version of the survey and asking them to mark the parameters that they think are deceptive. All participants marked the conspicuous parameters as deceptive. Note that these participants who were involved in the initial evaluation were not invited to the real experiment.

	# distinct honeytokens → # endpoint-method pairs				# dist. parameters: honeytokens / total	# endpoint-method pairs: with honeytokens / total
	Path	Query	Form	Body		
E-Commerce$_D$ (Survey I)	$1 \rightarrow 19$	-	$5 \rightarrow 4$	$2 \rightarrow 2$	8 / 71 (11%)	22 / 38 (58%)
E-Commerce$_Q$ (Survey II)	$2 \rightarrow 19$	$3 \rightarrow 3$	$14 \rightarrow 11$	$10 \rightarrow 4$	29 / 92 (32%)	30 / 38 (79%)
Lab-Platform$_D$ (Survey II)	$2 \rightarrow 10$	$1 \rightarrow 1$	$2 \rightarrow 3$	$4 \rightarrow 8$	9 / 83 (11%)	19 / 33 (57%)
Lab-Platform$_C$ (Survey I)	$5 \rightarrow 10$	$1 \rightarrow 1$	$4 \rightarrow 3$	$7 \rightarrow 8$	17 / 91 (19%)	19 / 33 (57%)

Table 2: Breakdown of the API variants: showing the distinct number of honeytokens, affected endpoint-method pairs, and their ratios to the total number of parameters and endpoint-method pairs.

4.2 Preparation of the surveys

Our experiment consists of two survey versions. Survey I contains *E-Commerce*$_D$ and *Lab-Platform*$_C$ APIs, and Survey II contains *E-Commerce*$_Q$ and *Lab-Platform*$_D$. Note that participants were not aware that there were two different versions of the survey. We advertised the survey with a single URL that redirects to a different version of the survey each time it is requested. We changed the redirection rules from time to time, to ensure that both versions will have the same number of participants.

Both surveys start with a section that describes the purpose of the survey. In particular, it states the following:

> In this experiment, you will be presented with 2 different application APIs that include a number of honey parameters. We will ask you to identify the parameters that you think are deceptive (i.e., if you were to attack this application, you would avoid tampering those parameters to avoid being detected).

Although we anonymize the APIs beforehand, we inform the participants that they are real-world APIs, and ask them to not search for the original APIs online for the sake of the validity of the study.

The first three questions of the survey aim to learn about participants' profile (current job title) and their experience on information security and deception technology. Then we have a different section for each application, where we first give a link to the Bootprint Swagger UI of the API. We then ask participants to identify the purpose of the API, and to rate their overall understanding of the purpose of the endpoints. Finally, we list all the distinct parameter names categorized by their location (path, query, form, body) and ask the participants to mark if it is *deceptive* or *genuine*. Note that, by default all answers are set to *genuine*, to save the participants from clicking too many times. We present several screenshots from the survey (including the complete description text) in the Appendix Section A.3.

4.3 Participants

We used snowball sampling to reach security experts. We advertise the survey mainly in two communities: First, among the security researchers, experts and enthusiasts in a large software company and second, among the computer security PhD students of a graduate school. Additionally, we advertise it on social media (Twitter).

Note that the survey description warns the participants about an estimated duration of 30 minutes, which was determined during the initial evaluation phase. Participation is completely on a voluntary basis, without any compensation. Overall, our advertisement is estimated to reach at least a few hundred people and the survey received answers between April 6 and May 24, 2021.

5 Results

We received 42 responses, which correspond to 21 participants for each version of the survey (Survey I & II). This number of responses allows us to show the effectiveness of our method and to make interesting observations.

5.1 Participants' profile

Majority of participants consist of software/web developers (19%), security researchers working in industry (17%) and MSc students doing internships (17%). Moreover, some PhD students (11%), postdocs, and professors (10%) have also answered the survey. 5 participants did not answer the question about their job title. Participants rate their information security experience as 3.5 ± 1.1 on a scale from 1 to 5. Moreover, they rate their knowledge on deception technology as 2.4 ± 0.9. Overall, the participants seem to have an above average experience in information security, and an average level of familiarity with deception technology.

5.2 Participants' understanding of the APIs

In a multi-choice question, we first ask participants to identify the purpose of the API. All participants correctly identified both the e-commerce and the laboratory platform applications. Then, we ask participants to rate their understanding of the purpose of API endpoints on a scale from 1 to 5. Participants seem to have a good understanding of the e-commerce endpoints (on average 4±0.6 for *E-Commerce*$_D$ and 4±0.5 for *E-Commerce*$_Q$) and a fair to good understanding of the endpoints of the laboratory platform (on average 3.6±0.6 for *Lab-Platform*$_D$ and 3.4±0.9 for *Lab-Platform*$_C$).

5.3 Indistinguishability of deceptive parameters

To see if our method was able to generate deceptive parameters that are indistinguishable from the genuine application parameters, we analyze the results from *E-Commerce*$_D$ (Survey I) and *Lab-Platform*$_D$ (Survey II). In particular, we compute a *detection rate* for each parameter, that is the ratio of participants that marked this parameter as deceptive. Then, we compare the detection ratios of the group of deceptive parameters and the group of genuine parameters: We use Welch's t-test as the groups have unequal sample sizes and unequal variances [5].

	Detection rate statistics		Welch's t-test (p <.05)
	Deceptive	Genuine	
E-Commerce$_D$ (Survey I)	24±13	18±14	t=1, p=.34 (Not significant)
Lab-Platform$_D$ (Survey II)	26±22	12±10	t=1.8, p=.1 (Not significant)

Table 3: Detection rate statistics for deceptive and genuine parameters for the APIs in default mode.

Table 3 shows the detection rate statistics and the results of Welch's t-tests for both applications. **Although detection rates of deceptive elements are slightly higher, we do not observe a statistically significant difference in comparison to the detection rates of genuine parameters**, at a confidence level of 95%. This shows that our method is able to generate indistinguishable deceptive parameters.

Previous work suggests that a "high-quality honeytoken" cannot be distinguished from a real token, even by the experts in relevant field [25]. Thus, the fact that 71% of our participants are security professionals or web developers increases confidence in the reliability of our results.

5.4 Impact of the quantity of deceptive parameters

In this section we aim to answer the following question: Do participants mislabel (i.e., fail to detect) more deceptive parameters, when there are significantly more of them?

For each participant, we compute the ratio of deceptive parameters that they correctly labeled. We also compute the ratio of parameters that they label as deceptive, among all parameters. This second metric refers to participants' expectation (or assumption) on the quantity of deceptive elements.

	Ratio of labeled honeytokens	Ratio of labeled parameters in total
E-Commerce$_D$ (Survey I)	23±16 %	19±8 %
E-Commerce$_Q$ (Survey II)	25±19 %	17±13 %

Table 4: Average ratios of parameters that are labeled as deceptive, among (i) honeytokens and (ii) all parameters.

Table 4 shows that **regardless of the quantity of honeytokens, participants label more or less the same percentage of honeytokens as deceptive** (∼23 to 25%). Thus, in *E-Commerce*$_D$ on average 6 of the 8 deceptive parameters remain undetected by participants, while in *E-Commerce*$_Q$, 22 of 29 are undetected. Moreover, **participants' expectation of the ratio of deceptive parameters (among all parameters) remains rather constant even when there are statistically significantly more honeytokens** (∼17 to 19%). This means that, when there is a larger quantity of realistic deceptive elements, there will be more elements that will remain undetected. On the other hand, generating a very large number of realistic deceptive elements remains a challenge. As discussed in Section 4.1, we used a semi-manual approach, as our model was able to generate a limited number of high quality deceptive elements.

Finally, we also compare the detection rates of deceptive parameters in *E-Commerce*$_Q$ that were automatically generated by our model (14 parameters) and that were manually created by us (15 parameters), to see how our approach compares to manual selection. Average detection rate is found to be 21±9% for automated honeytokens, and 23±11% for manual honeytokens. Applying a Welch's t-test, we do not see a significant difference between detection rates (t=-0.44, p=.66). Thus, we can conclude that **automatically generated parameter names were as realistic as manually selected ones**.

5.5 Impact of the conspicuous deceptive parameters

In this section we aim to answer the following question: Do participants mislabel (i.e., fail to detect) more deceptive parameters, when there are some very obvious (conspicuous) honeytokens added as extra? The idea is that conspicuous honeytokens might help to hide realistic honeytokens, by attracting participants' attention.

Table 5 shows that, if we exclude the conspicuous parameters in *Lab-Platform*$_C$, participants label more or less the same percentage of honeytokens as deceptive (∼25 to 26%) in both *Lab-Platform*$_D$ and *Lab-Platform*$_C$. Moreover, participants' expectation of the ratio of deceptive parameters again remains more or

	Ratio of labeled honeytokens	Ratio of labeled parameters in total
Lab-Platform$_D$ (Survey II)	26±16 %	13±10 %
Lab-Platform$_C$ (Survey I) (excluding conspicuous)	25±14 %	12±7 %

Table 5: Average ratios of parameters that are labeled as deceptive, among (i) honeytokens and (ii) all parameters.

less constant (∼12 to 13%). Thus, **we do not observe any significant impact of adding conspicuous honeytokens on further disguising the realistic honeytokens**.

On the other hand, a significantly higher number of participants label the conspicuous honeytokens as deceptive (on average, 72±11%), in comparison to the realistic honeytokens (on average, 24±30%) in *Lab-Platform*$_C$ (Welch's t-test: p=.001). Thus, we believe that **conspicuous honeytokens can be used to tip off the attacker about the presence of deception**, in order to enable the deception awareness effect that we will discuss next.

5.6 Deception awareness effect

In this section, we look at the ratio of genuine parameters that are labeled as deceptive by the participants. On average, *E-Commerce*$_D$ and *E-Commerce*$_Q$ have 18±8% and 15±11% of the genuine parameters mislabeled, respectively. These ratios are 12±10% and 11±7% for *Lab-Platform*$_D$ and *Lab-Platform*$_C$. Thus, **we observe that at least 10% of genuine parameters were marked as deceptive across all APIs**, which means that participants would avoid tampering with those parameters in an attack scenario. We can interpret this as the effect of deception awareness. Previous studies already observe various benefits of informing attackers about the presence of deception, such as compelling them to modify their attack behavior, impeding the attack progress, and deteriorating attackers' cognitive and psychological state [64,35]. Our results demonstrate yet another benefit of deception awareness, that is, to masquerade the real application elements to look like traps.

6 Limitations and Discussion

Method: In this study we only considered the source code of Java web applications from public GitHub repositories to train the model. However, it is possible to enrich the model with other codebases and projects using different web technologies (e.g., PHP, Node.js). Note that, the number of high quality parameters that can be generated by the model depends on the richness of the training data. Another limitation of our approach is that it is not able to generate compound parameter names. This can be done as a manual post-processing step (e.g., by adding a common prefix or suffix to some parameter names), or it would require

to train a model using the compound words as single words, if a proper training set is available. In addition, although we only used *word2vec*, combining it with other NLP approaches (discussed in Section 3) is also possible.

Evaluation: The results of our evaluation survey only provide insights about whether the participants were able to distinguish between the generated parameter names and the names of genuine application parameters. *Thus, these results should not be considered as a measurement of the effectiveness of deceptive elements in attack detection.* On the other hand, it is important to note that the deceptive parameters mainly aim to detect attacks via attackers' interaction (e.g., tampering the parameter), as opposed to the traditional honeypots that aim to waste attackers' time and resources. This means that, *as soon as an attacker interacts with a deceptive parameter (e.g., with a fuzzing tool), he will be detected and the application will respond accordingly (e.g., by blocking the request or routing to a clone system).* Thus, having realistic deceptive parameters becomes a first requirement to ensure the effectiveness of deception.

As mentioned in Section 4.2, in the evaluation survey we only have *deceptive* and *genuine* options to choose between. Thus, participants are forced to make a choice even when they are not sure about the answer. In fact, we have received a few post-survey comments where the participants found some parameters to be implausible, but they were not sure if it was just due to bad API design practices, or due to deception. For instance, one participant stated that:

> Some of these APIs look off from a programming perspective. Why would you include <variable> as a query string when it might be more efficient to use it elsewhere?

Another participant said:

> I would be extra careful in a situation like this and mark things [that maybe are not deceptive] as deceptive just in case. Taking into account that programmers are not perfect, they may create parameters that are not needed. So I think this is not needed, but is it because it is deceptive or it was done like this in reality... My general approach when doing tampering is, just touch what you are sure of.

These comments imply a few points: First, it is likely that there will always be a suspicion about implausible-looking elements. Second, as also discussed in Section 2.4, it is important to keep the coherence between API endpoints and imitate realistic functionality for the generated deceptive elements. Finally, we believe that obliging participants to take a decision is a more realistic approach, as in a real attack scenario they would need to make a decision to tamper or not. In fact, previous work observes via a CTF-based experiment that, although most participants are initially very careful to not touch the suspicious-looking elements, they give up on such precautions after some time, if they cannot find an attack vector to progress [64].

7 Related Work

While there are many studies that aim to generate various deceptive content or honey elements, we focus on the ones that relate to web application security.

HoneyGen [25] aims to create relational database with fake entries, based on the rules extracted from a real database. For evaluation, the method is applied on a database from a real-world dating website, to create fake profiles with different personal information attributes. The experiment involved 30 pairs of profiles, each pair having one real and one fake persona. The 109 participants who joined the experiment were unable to distinguish the fake profiles that have high similarity to the real profiles. B.Hive [59], as discussed in Section 2.1, aims to generate honey form field names using a dataset of form fields collected from top websites. While this approach is limited to form parameters from pre-authentication pages, our approach targets all types of HTTP parameters and a wide range of application contexts. A more recent study [22] proposes to allow the user to enrich the UI of a web application with custom honey HTML elements (e.g., link, button, icon), via a browser extension. The idea is that the genuine users would be aware of these 'tripwires' (and not interact with them), but an attacker could easily click on them once he gains access to the account. While the names of the honey HTML elements are ideally chosen by the user, authors also implement a suggestion tool based on a Markov model of URLs gathered from the Common Crawl [2] dataset. However, the paper does not provide an evaluation on the quality of the suggested names.

BogusBiter [73] proposes to generate honey credentials that will be fed into phishing pages to conceal the real credentials of the user. The idea is to start with an initial set of credentials, and generate additional credentials by substituting certain characters of the username and password with different characters, each time. Other relevant studies propose different approaches for password guessing, based on a combination of specialized lists, lexical dictionaries and word embeddings [53] or deep learning techniques [45].

Finally, several studies aim to detect parameter tampering by looking at input validation failures or inconsistencies between client and server state [26,65,40]. Our work is complementary to these approaches.

8 Conclusion

This work automates the generation of realistic deceptive parameter names for different types of HTTP parameters. We demonstrate the effectiveness of our method via a survey based experiment, and find that the participants anticipate a certain ratio of elements to be deceptive, regardless of the actual quantity or enticement level of the honeytokens. Additionally, we observe that at least 10% of genuine API parameters were marked as deceptive by the participants, which demonstrates the potential benefit of informing the attackers about the presence of honeytokens. Finally, we provide various directions for future work by looking into the challenges that needs to be addressed for a complete automation of API layer deception.

References

1. Adobe CIF REST API and data model. https://github.com/adobe/commerce-cif-api/tree/deprecated
2. Common Crawl. https://commoncrawl.org/the-data/get-started/
3. difflib Helpers for computing deltas. https://docs.python.org/3/library/difflib.html
4. Fablabs.io Developer Guide. https://docs.fablabs.io/swagger/index.html
5. Scipy stats ttest reference. https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ttest_ind.html#scipy.stats.ttest_ind
6. Deception Technology Market. https://www.marketsandmarkets.com/Market-Reports/deception-technology-market-129235449.html (Feb 2017)
7. Global Deception Technology Market: Growth, Trends and Forecast to 2025 - ResearchAndMarkets.com. https://www.businesswire.com (April 2020)
8. bootprint-swagger. https://github.com/karlvr/bootprint-swagger (2021)
9. Describing Parameters. https://swagger.io/docs/specification/describing-parameters/ (2021)
10. Describing Request Body. https://swagger.io/docs/specification/2-0/describing-request-body/ (2021)
11. Describing Request Body. https://swagger.io/docs/specification/describing-request-body/ (2021)
12. Swagger UI. https://swagger.io/tools/swagger-ui/ (2021)
13. Acalvio: Shadowplex autonomous deception. https://www.acalvio.com/why-acalvio/ (2019)
14. Allamanis, M., Barr, E.T., Devanbu, P., Sutton, C.: A survey of machine learning for big code and naturalness. ACM Computing Surveys (CSUR) **51**(4), 1–37 (2018)
15. Almeshekah, M., Spafford, E.: Planning and integrating deception into computer security defenses. In: Proceedings of the New Security Paradigms Workshop. NSPW'14 (2014)
16. Alon, U., Zilberstein, M., Levy, O., Yahav, E.: code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages **3**(POPL), 1–29 (2019)
17. Alsentzer, E., Murphy, J.R., Boag, W., Weng, W.H., Jin, D., Naumann, T., McDermott, M.: Publicly available clinical bert embeddings. arXiv preprint arXiv:1904.03323 (2019)
18. Anderson, P.: Deception: A healthy part of any defense in-depth strategy. sans.org/reading-room/whitepapers/policyissues/deception-healthy-defense-in-depth-strategy-506 (March 2002)
19. Araujo, F., Hamlen, K.W., Biedermann, S., Katzenbeisser, S.: From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 942–953. CCS'14, ACM, New York, USA (2014)
20. Atlidakis, V., Godefroid, P., Polishchuk, M.: Restler: Stateful rest api fuzzing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 748–758 (2019). https://doi.org/10.1109/ICSE.2019.00083
21. Attivo Networks: Threat detection. https://attivonetworks.com/solutions/threat-detection/ (2019)
22. Barron, T., So, J., Nikiforakis, N.: Click this, not that: Extending web authentication with deception. In: Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security. p. 462474. ASIA CCS '21, Association for Computing Machinery, New York, NY, USA (2021)

23. Beagle Security: Parameter Tampering. `https://beaglesecurity.com/blog/vulnerability/parameter-tampering.html` (2021)
24. Ben-Nun, T., Jakobovits, A.S., Hoefler, T.: Neural code comprehension: A learnable representation of code semantics. arXiv preprint arXiv:1806.07336 (2018)
25. Bercovitch, M., Renford, M., Hasson, L., Shabtai, A., Rokach, L., Elovici, Y.: Honeygen: An automated honeytokens generator. In: Proceedings of 2011 IEEE International Conference on Intelligence and Security Informatics. pp. 131–136 (July 2011). https://doi.org/10.1109/ISI.2011.5984063
26. Bisht, P., Hinrichs, T., Skrupsky, N., Bobrowicz, R., Venkatakrishnan, V.N.: Notamper: Automatic blackbox detection of parameter tampering opportunities in web applications. In: Proceedings of the 17th ACM Conference on Computer and Communications Security. p. 607618. CCS '10, Association for Computing Machinery, New York, NY, USA (2010)
27. Bowen, B.M., Hershkop, S., Keromytis, A.D., Stolfo, S.J.: Baiting inside attackers using decoy documents. In: Chen, Y., Dimitriou, T.D., Zhou, J. (eds.) Security and Privacy in Communication Networks. pp. 51–70. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
28. Cabrera Lozoya, R., Baumann, A., Sabetta, A., Bezzi, M.: Commit2vec: Learning distributed representations of code changes. arXiv preprint arXiv:1911.07605 (2019)
29. Calzavara, S., Conti, M., Focardi, R., Rabitti, A., Tolomei, G.: Mitch: A machine learning approach to the black-box detection of csrf vulnerabilities. In: 2019 IEEE European Symposium on Security and Privacy (EuroSP) (2019)
30. Chen, B., Jiang, Z.M.: Studying the use of java logging utilities in the wild. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE). pp. 397–408 (2020). https://doi.org/10.1145/3377811.3380408
31. Chen, Z., Monperrus, M.: A literature study of embeddings on source code. arXiv preprint arXiv:1904.03061 (2019)
32. Cohen, F., Marin, I., Sappington, J., Stewart, C., Thomas, E.: Red teaming experiments with deception technologies. `http://all.net/journal/deception/RedTeamingExperiments.pdf` (2001)
33. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
34. Ferguson-Walter, K., Shade, T., Rogers, A., Niedbala, E., Trumbo, M., Nauer, K., Divis, K., Jones, A.P., Combs, A., Abbott, R.G.: The tularosa study: An experimental design and implementation to quantify the effectiveness of cyber deception. In: HICSS (2019)
35. Ferguson-Walter, K.J., Major, M.M., Johnson, C.K., Muhleman, D.H.: Examining the efficacy of decoy-based and psychological cyber deception. In: 30th USENIX Security Symposium (USENIX Security 21). USENIX Association (Aug 2021)
36. Fidelis Cybersecurity: Fidelis deception. `https://www.fidelissecurity.com/wp-content/uploads/2019/04/Fidelis-Deception-1905.pdf` (2019)
37. Fraunholz, D., Schotten, H.D.: Defending web servers with feints, distraction and obfuscation. In: 2018 International Conference on Computing, Networking and Communications (ICNC). pp. 21–25 (March 2018)
38. Fraunholz, D., Antón, S.D., Lipps, C., Reti, D., Krohmer, D., Pohl, F., Tammen, M., Schotten, H.D.: Demystifying deception technology: A survey. CoRR **abs/1804.06196** (2018)

39. Fraunholz, D., Reti, D., Duque Anton, S., Schotten, H.D.: Cloxy: A context-aware deception-as-a-service reverse proxy for web services. In: Proceedings of the 5th ACM Workshop on Moving Target Defense. MTD '18, ACM, New York, NY, USA (2018)
40. Fung, A.P., Wang, T., Cheung, K.W., Wong, T.Y.: Scanning of real-world web applications for parameter tampering vulnerabilities. In: Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security. p. 341352. ASIA CCS '14, Association for Computing Machinery, New York, NY, USA (2014), https://doi.org/10.1145/2590296.2590324
41. Gousios, G.: The GHTorrent dataset and tool suite. In: Proceedings of the 10th Working Conference on Mining Software Repositories. pp. 233–236. MSR '13 (May 2013), /pub/ghtorrent-dataset-toolsuite.pdf, best data showcase paper award
42. Grent, H., Akimov, A., Aniche, M.F.: Automatically identifying parameter constraints in complex web apis: A case study at adyen. CoRR **abs/2102.00871** (2021), https://arxiv.org/abs/2102.00871
43. Han, X., Kheir, N., Balzarotti, D.: Evaluation of deception-based web attacks detection. In: Proceedings of the 2017 Workshop on Moving Target Defense. MTD '17, ACM, New York, NY, USA (2017)
44. Han, X., Kheir, N., Balzarotti, D.: Deception techniques in computer security: A research perspective. ACM Comput. Surv. **51**(4), 80:1–80:36 (Jul 2018)
45. Hitaj, B., Gasti, P., Ateniese, G., Perez-Cruz, F.: Passgan: A deep learning approach for password guessing. In: Deng, R.H., Gauthier-Umaña, V., Ochoa, M., Yung, M. (eds.) Applied Cryptography and Network Security. pp. 217–237. Springer International Publishing, Cham (2019)
46. Huang, K., Altosaar, J., Ranganath, R.: Clinicalbert: Modeling clinical notes and predicting hospital readmission. arXiv preprint arXiv:1904.05342 (2019)
47. Illusive Networks: Attack detection system. https://www.illusivenetworks.com/technology/platform/attack-detection-system (2019)
48. Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., Soricut, R.: Albert: A lite bert for self-supervised learning of language representations. arXiv preprint arXiv:1909.11942 (2019)
49. Lee, J., Yoon, W., Kim, S., Kim, D., Kim, S., So, C.H., Kang, J.: Biobert: a pre-trained biomedical language representation model for biomedical text mining. Bioinformatics **36**(4), 1234–1240 (2020)
50. Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., Stoyanov, V.: Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692 (2019)
51. Matteo Meucci and Andrew Muller : Testing guide 4.0. https://owasp.org/www-project-web-security-testing-guide/assets/archive/OWASP_Testing_Guide_v4.pdf (2021)
52. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781 (2013)
53. Ocanto-Dávila, C., Cabrera-Lozoya, R., Trabelsi, S.: Sociocultural influences for password definition: An ai-based study. In: Proceedings of the 7th International Conference on Information Systems Security and Privacy (ICISSP). pp. 542–549 (2021)
54. OpenAPI Initiative: OpenAPI Specification v3.1.0. https://spec.openapis.org/oas/v3.1.0.html (2021)
55. OWASP: Web Parameter Tampering. https://owasp.org/www-community/attacks/Web_Parameter_Tampering (2021)

56. OWASP Foundation: Appsensor detection points. https://www.owasp.org (2015)
57. Pennington, J., Socher, R., Manning, C.D.: Glove: Global vectors for word representation. In: Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). pp. 1532–1543 (2014)
58. Peters, M.E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., Zettlemoyer, L.: Deep contextualized word representations. arXiv preprint arXiv:1802.05365 (2018)
59. Pohl, C., Zugenmaier, A., Meier, M., Hof, H.J.: B.hive: A zero configuration forms honeypot for productive web applications. In: 30th IFIP International Information Security Conference (SEC). pp. 267–280 (May 2015)
60. Pohl, C., Zugenmaier, A., Meier, M., Hof, H.J.: B.Hive: A Zero Configuration Forms Honeypot for Productive Web Applications. In: Federrath, H., Gollmann, D. (eds.) 30th IFIP International Information Security Conference (SEC). ICT Systems Security and Privacy Protection, vol. AICT-455, pp. 267–280. Hamburg, Germany (May 2015), part 4: Network Security
61. PortSwigger: Insecure direct object references (IDOR). https://portswigger.net/web-security/access-control/idor (2021)
62. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I.: Language models are unsupervised multitask learners. OpenAI blog **1**(8), 9 (2019)
63. Řehůřek, R., Sojka, P.: Software Framework for Topic Modelling with Large Corpora. In: Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks. pp. 45–50. ELRA, Valletta, Malta (May 2010), http://is.muni.cz/publication/884893/en
64. Sahin, M., Hebert, C., Oliveira, A.: Lessons learned from sundew: A self defense environment for web applications. In: Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) in NDSS Symposium 2020 (01 2020). https://doi.org/10.14722/madweb.2020.23005
65. Skrupsky, N., Bisht, P., Hinrichs, T., Venkatakrishnan, V.N., Zuck, L.: Tamperproof: A server-agnostic defense for parameter tampering attacks on web applications. In: Proceedings of the Third ACM Conference on Data and Application Security and Privacy. p. 129140. CODASPY '13, Association for Computing Machinery, New York, NY, USA (2013)
66. Swagger: Flatten a swagger spec. https://goswagger.io/usage/flatten.html
67. Swagger: OpenAPI Specification Version 2.0. https://swagger.io/specification/v2/
68. The MITRE Corporation: CWE-471: Modification of Assumed-Immutable Data (MAID). https://cwe.mitre.org/data/definitions/471.html (March 2021)
69. Theil, C.K., Štajner, S., Stuckenschmidt, H.: Explaining financial uncertainty through specialized word embeddings. ACM Transactions on Data Science **1**(1), 1–19 (2020)
70. ThinkstCanary: Canarytokens. https://canarytokens.org (2019)
71. Trinh, T.H., Le, Q.V.: A simple method for commonsense reasoning. arXiv preprint arXiv:1806.02847 (2018)
72. Wu, Q., Wu, L., Liang, G., Wang, Q., Xie, T., Mei, H.: Inferring dependency constraints on parameters for web services. In: Proceedings of the 22nd International Conference on World Wide Web. p. 14211432. WWW '13, Association for Computing Machinery, New York, NY, USA (2013)
73. Yue, C., Wang, H.: Bogusbiter: A transparent protection against phishing attacks. ACM Trans. Internet Technol. **10**(2) (Jun 2010)

74. Zhu, Y., Kiros, R., Zemel, R., Salakhutdinov, R., Urtasun, R., Torralba, A., Fidler, S.: Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In: Proceedings of the IEEE international conference on computer vision. pp. 19–27 (2015)

A Appendix

A.1 Example construction of sentences from Java source code

Example Java sourcecode

```
public class LiveProfile {

        private final String id;
        private final String name;
        private final String firstName;
        private final String lastName;
        private final String gender;
        private final String email;
        private String link;
        private String locale;
        private String updatedTime;

        public LiveProfile(String id, String name, String
            firstName, String lastName, String gender,
           String email, String locale) {
                this.id = id;
                this.name = name;
                this.firstName = firstName;
                this.lastName = lastName;
                this.gender = gender;
                this.email = email;
        }
}
```

Extracted sentences to train the model

```
live profile id name first name last name gender email locale
id name first name last name gender email link locale updated time
```

A.2 Example output of the model

– Endpoint-Method pair: /products/search - GET
Existing query parameters:

–Name–	–Type–	–Location–
text	string	query
filter	array	query
selectedFacets	array	query
queryFacets	array	query
sort	array	query
offset	integer	query
limit	integer	query

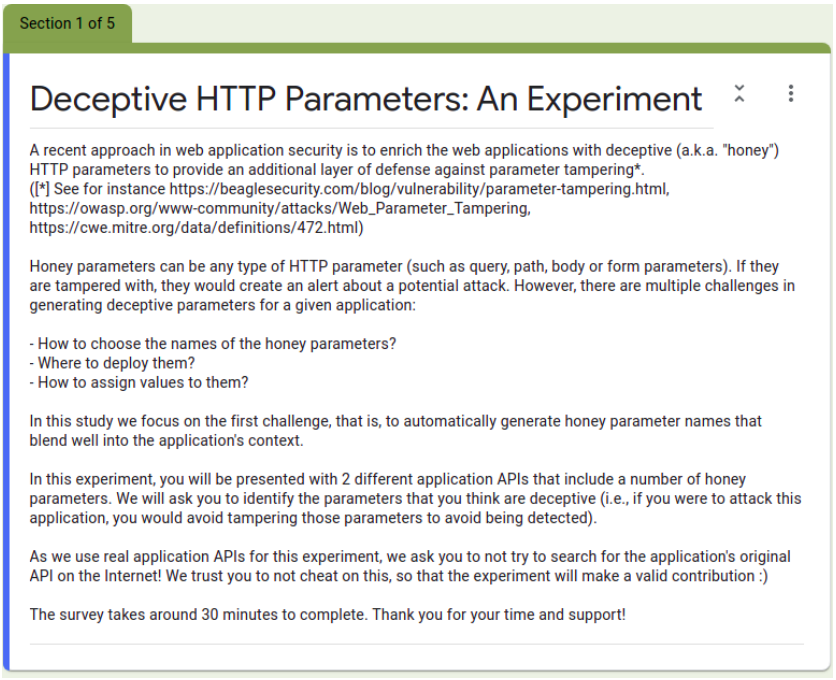
Candidate deceptive parameters (n=5): `sorts, filters, filtered, criteria, facet`
Remaining parameters after the post-processing steps: `criteria`
Recommended type for the selected honeytoken '`criteria`': **string**

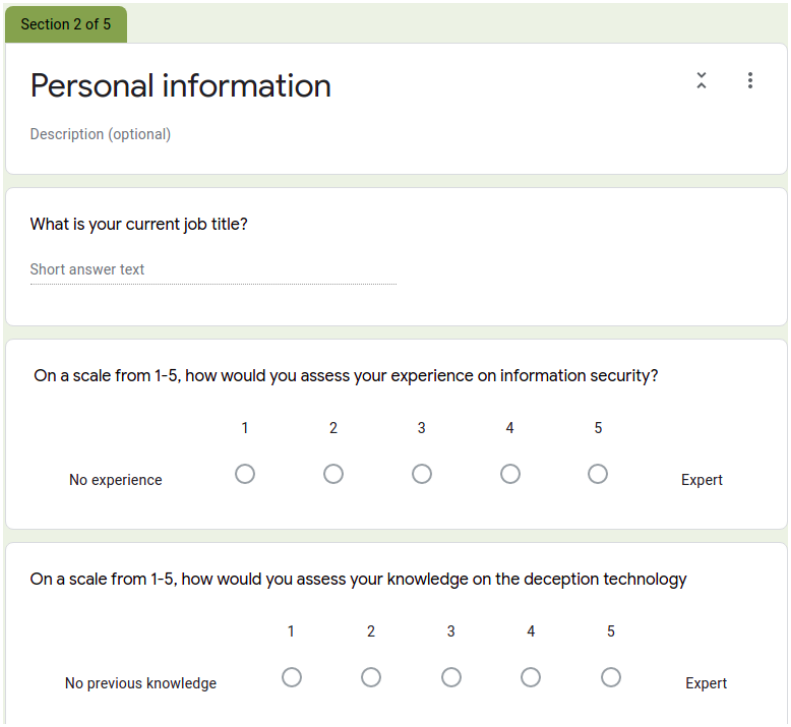
– Endpoint-Method pair: /carts - POST
Existing form parameters:

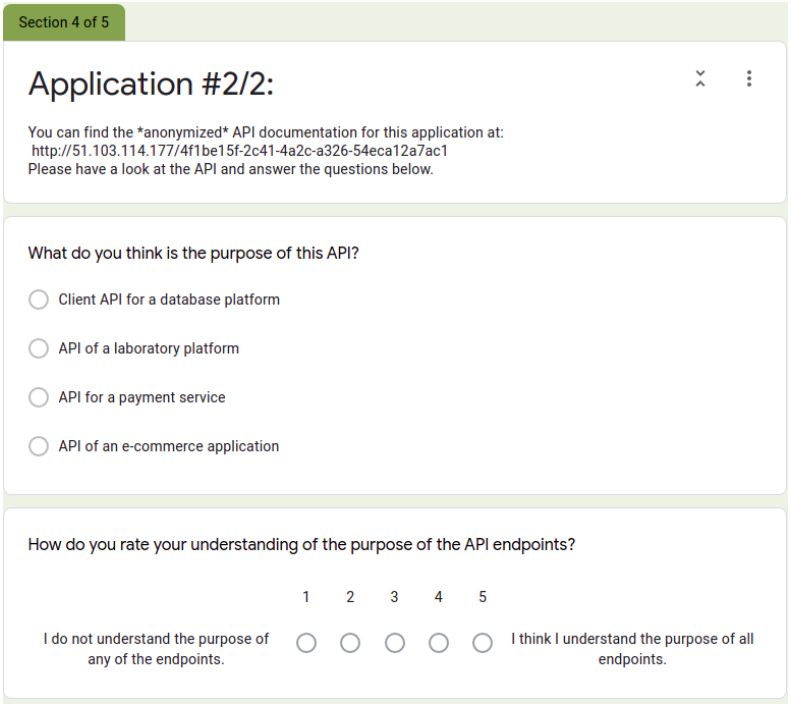
–Name–	–Type–	–Location–
currency	string	formData
productVariantId	string	formData
quantity	integer	formData

Candidate deceptive parameters (n=5): `price, sku, uom, retail, taxed`
Remaining parameters after the post-processing steps: `price`, `sku`, `uom`, `retail`
Recommended type for the selected honeytoken '`price`': **int**

A.3 Survey - Introduction message, personal data collection and example form parameters from *Lab-Platform*$_D$ application




Section 2 of 5
Personal information
Description (optional)
What is your current job title?
Short answer text
On a scale from 1-5, how would you assess your experience on information security?
1 2 3 4 5
No experience
Expert
On a scale from 1-5, how would you assess your knowledge on the deception technology
1 2 3 4 5
No previous knowledge
Expert


Section 4 of 5
Application #2/2:
You can find the *anonymized* API documentation for this application at:
http://51.103.114.177/4f1be15f-2c41-4a2c-a326-54eca12a7ac1
Please have a look at the API and answer the questions below.
What do you think is the purpose of this API?
Client API for a database platform
API of a laboratory platform
API for a payment service
API of an e-commerce application
How do you rate your understanding of the purpose of the API endpoints?
1 2 3 4 5
I do not understand the purpose of any of the endpoints.
I think I understand the purpose of all endpoints.

Now, please try to identify if the parameters listed in the questions are honeytokens (e.g., deceptive elements that you think that we have added), or if they are genuine parameters. Note that all parameters are set to 'Genuine' by default.

Form Parameters

	Deceptive	Genuine
client_id	○	◉
client_secret	○	◉
redirect_uri	○	◉
bearer	○	◉
access_token	○	◉
username	○	◉
code	○	◉
uaa	○	◉
grant_type	○	◉

A.4 Partial Swagger UI for endpoints related to the form parameters above

POST /oauth/token

DESCRIPTION

REQUEST BODY

application/x-www-form-urlencoded

REQUEST PARAMETERS

Name	Description	Type	Data type
client_id		formData	*string*
client_secret		formData	*string*
redirect_uri		formData	*string*
code		formData	*string*
uaa		formData	*string*
grant_type		formData	*string*

DELETE /oauth/authorize

DESCRIPTION

REQUEST BODY

application/x-www-form-urlencoded

REQUEST PARAMETERS

Name	Description	Type	Data type
client_id		formData	*string*
client_secret		formData	*string*
redirect_uri		formData	*string*
bearer		formData	*string*
access_token		formData	*string*

POST /oauth/authorize

DESCRIPTION

REQUEST BODY

application/x-www-form-urlencoded

REQUEST PARAMETERS

Name	Description	Type	Data type
client_id		formData	*string*
client_secret		formData	*string*
redirect_uri		formData	*string*
bearer		formData	*string*
username		formData	*string*