An Application of Jeeves for Honeypot Sanitization

Ashton Webster

Abstract

Being able to quickly create realistic honeypots is very useful for obtaining accurate information about attacker behavior. However, creating realistic honeypots requires *sanitization* of the original system from which the honeypot is derived. To achieve this the use of the Jeeves, a language based on faceted values, is extended to rapidly replace secret values with *believable* and *non-interfering* sanitized values. By making several changes to the source code of Jelf, a web server implemented in Jeeves, we are able to quickly and easily create sanitized honeypots. Our experiments show that the sanitized and unsanitized versions of Jelf only differ in response times by less than 1%.

1 Introduction

Secure Information Flow is an area of growing focus for researchers. Being able to reason about private information and the policies surrounding it is a complicated and important issue. Several programming language approaches have been taken to handle this issue, including Jif [1], Jeeves [2], and HiStar [3]. In addition to being able to secure this private data, security researchers are interested in finding ways to quickly *sanitize* this private information for use in *honeypots*. Here we define *sanitization* as the process by which private information from the execution of a process is removed or replaced so that the program state no longer discloses any private information to the user[1]. This is useful for *honeypots*, which are machines masquerading as legitimate servers for the purpose of attack detection and analysis. Specifically, sanitization provides a way of creating a safe and potentially believable environment for an attacker to be routed to in the case of an attack.

In this paper, we focus on the python implementation of the Jeeves programming language. By modifying the language, we are able to sanitize secure values in a way that is difficult to detect by the attacker. We also show that these modifications have only a very small impact on performance of the system. The rest of the paper is laid out as follows: First, summary and commentary for three papers introducing the components of the Jeeves language are provided. Next, the problem of sanitization is more formally stated and experiments are proposed. Then, the implementation details are described and the experiments are conducted. Finally, future work is proposed and final conclusions are summarized.

2 Related Work

Initially, Jeeves was introduced [2] as a language with symbolic variables that were restricted to either a "high" or "low" level based on constraints in the environment. Anecdotally, the authors presented this work at the conference immediately after a work introducing faceted values [4], and the authors from both papers introduced an extended version of Jeeves making use of faceted values. The main benefit of the faceted values over symbolic values is the ability to more quickly prune impossible values based on a limited set of possible values Finally, Yang, Hance, and Austin introduce a Database Framework extension to Jeeves called Jaqueline, requiring additional semantic rules [5]. This section gives a short overview of the contributions of each of these works, along with a final section summarizing previous work in honeypots

[1]Notice that our definition of sanitization varies from the more common *input sanitization*, which primarily seeks to prevent the use of user input as part of program execution, especially via SQL Injection or remote code injection. It also differs from sanitization in the context of anonymizing data for research purposes.

and sanitization.

2.1 Jeeves with Symbolic values

In [2], Yang proposes Jeeves. The basic structure of the paper is to first introduce the Jeeves language syntax, then introduce a "constraint functional language" λ_J. Finally, the correspondence between Jeeves and λ_J is proven, Jeeves is implemented as a Scala library, and several applications are demonstrated, along with comparisons of the the original and modified policy lines-of-code required.

The authors first explain how λ_J works. The rules are fairly similar to lambda calculus, but there is additional state maintained. There are three main "pieces" of state: G, Σ, Δ.

- G is the current path condition, which includes information about which branch we are currently on within the program
- Σ is known as the "constraints" (sometimes "hard-constraints"), which define the constraints required for the secret faceted value to be displayed for a faceted value.
- Δ is known as the "default assumptions" (sometimes "soft-constraints"), which define the secret value. Basically this simplifies the computations for underconstrained satisfaction problems by specifying which symbolic value to use in the case of ambiguity. This is essentially required for the type of SMT solver that the authors of this paper use to determine the value.

Rules then take the general form

$$G \vdash \langle \Sigma, \Delta, e \rangle \to \langle \Sigma', \Delta', e' \rangle$$

where e is an expression. In general, the `mkLabel` and `mkSensitive` functions in the Jeeves Python/Scala implementation adds a default assumption to Δ that the high security/secret value will be the default (this was a design decision by the authors and does not affect the non-interference property; the public value could also be used as the default). This has certain effects on when faceted values depend on themselves as the guard, creating circular dependencies. The authors decide these will default to the high value. Here is an example in the python interface:

```
# create label
x = JeevesLib.mkLabel()
# restrict variable to require
# context == True for high value
JeevesLib.restrict(x,
    lambda x: x == True)
# Create faceted value with
# low = False and high = True,
# guarded by x
facetedValue = JeevesLib.mkSensitive(x,
    True, False)
context = facetedValue
# circular dependency: facetedValue
# depends on context
print(JeevesLib.concretize(context,
    facetedValue))
# Output: True
```

The `restrict` function adds constraints to Σ specifying what the context must be in order for the secure value to be displayed. Σ and Δ are ultimately passed to an SMT solver which either produces an error (if no satisfying conditions exist) or provides a satisfying value which guides which value (high or low) is produced.

2.2 Jeeves with Faceted values

In [6], the Jeeves language was modified to use Faceted values. In the paper, the authors explain: "The system may ... prune facets based on path assumptions: if evaluation is occurring under the assumption that guard k is true, then subsequent evaluation can assume guard k". An example which demonstrates this property is given in listing 1 in Python. The `JeevesLib.jif(cond, then, else)` function takes a condition, `then` case (in case of true), and `else` case (in case of false) to construct an if statement. `myif` represents a nested if statement. In the `mythen` case, the condition (or in this case, faceted value `fv`) is assumed to be true. Similarly, the faceted value must be false in the `myelse` case. Therefore, Jeeves is able to prune the inconsistent cases and simplify the faceted value from

Listing 1: Pruning of Facets

```
JeevesLib.init()
x = JeevesLib.mkLabel()
JeevesLib.restrict(x, lambda q: q == True)
fv = JeevesLib.mkSensitive(x, True, False)

def mythen():
    return JeevesLib.jif(fv, lambda: 1, lambda: 2)

def myelse():
    return JeevesLib.jif(fv, lambda: 3, lambda: 4)

myif = JeevesLib.jif(fv, mythen, myelse)
print(myif)
#Output: < v0 ? const:1 : const:4 >
```

ID	Jeeves ID	Jeeves Vars	User	Email
1	1	(profile==bob)	bob	[redacted]
2	1	!(profile==bob)	bob	bob@g.com

Table 1: Example of a row for a user in Jeeves database.

`<v0 ? <v0 ? const:1 : const:2> : <v0 ? const:3 : const:4>>` to `<v0 ? const:1 : const:4>` based on these assumptions. Pruning like this makes faceted values more performant than symbolic variables.

2.3 Jeeves Database

Finally, Yang et al. add database functionality to the Jeeves language. This is achieved by again extending the original λ_J core language to include union, intersection, selection, projection, and cross product operators. Additionally, a Faceted Object Relational Model (FORM) is implemented in order to populate faceted rows in the database and marshal them back to faceted values on queries. In our example project, a row with a faceted value for the email, with low value "[redacted]" and high value "bob@g.com" would be represented as in figure 1.

As part of this paper, the authors provide an open source code repository on GitHub[2]. This repository includes a python implementation of Jeeves, along with a web framework named "Jelf", which uses Jeeves for its FORM. Jelf is an extension of the python Django web framework and is used as the basis for the experiments in this paper. While the authors refer to the Jeeves database framework as "Jacqueline" in the paper

2.4 Honeypots

Creating realistic honeypots can be a powerful tool for analyzing the behavior of attackers. Hirata et al. use a method of *live-cloning* a live, legitimate server to create an identical honeypot version using file synchronization techniques [7]. However, Hirata's method does not consider "sanitization", that is, the removal sensitive files during cloning. This method is elaborated on in [8], where the authors rapidly live-clone legitimate servers to be used for honeypots, but the authors also perform basic sanitization, such as disallowing certain block reads via modification to the virtualization software and manually removing web server directories and `passwd` files. Techniques like this, which create honeypots "on the fly" are even more useful when used in conjunction with Software Defined Networking (SDN) techniques. For exam-

[2]`https://github.com/jeanqasaur/jeeves`

ple, Huang et al. uses the Snort [9] Intrusion Detection System (IDS) to automatically route traffic to dynamically created honeypots [10]. In this paper, we leave the network reconfiguration and live-cloning methods out-of-scope, but we assume it is possible to quickly redirect traffic from an attacker to a honeypot clone. Instead, we focus on the sanitization component, which we implement using Jeeves and Jelf.

3 Problem Statement

This paper seeks to answer two questions: (1) how can Jeeves/Jelf be used for honeypot sanitization? And, (2) what is the performance implication of using Jelf for honeypot sanitization relative to a plain web framework and relative to Jelf without the honeypot sanitization enabled? In order to answer the first of these questions, an implementation of honeypot sanitization built on top of the Jelf framework is provided. This implementation is described in the subsequent sections. For the second question, an experiment is conducted using the Apache Benchmark (used via its unix tool, `ab`) to evaluate the response times of the web server under a variety of conditions.

As a specific example, a password management site is implemented. This is similar to LastPass, where users can add sites with usernames and passwords, and access them with a master password. As context for the sanitization portion of this, we consider the possibility that an attacker is able to obtain a users master password. If the network administrator for this password management site is notified by the user that their password has been compromised, the administrator can route traffic using the old password to a sanitized honeypot. Another possibility is that the network administrator or an automated tool detects a horizontal brute force attack on user passwords. This type of attack is characterized by an attacker trying many user names in sequence, each with common passwords. These triggers are just examples and are left out of scope for this project. However, it is assumed that there is some ability to detect attackers masquerading as legitimate users and route them to honeypots. Research on creating these triggers and routing to honeypots is currently being conducted as part of a separate project, and is left as a "black box" here due to time constraints. Therefore, we consider an attacker model with the ability to masquerade as another legitimate user and execute requests on their behalf. We do not consider an attacker with access to the memory of the web server, and leave this problem as future work.

In addition to sanitizing the values, there are certain properties which are desirable for the sanitized values. We enumerate these here:

1. *Noninterference:* the sanitized values displayed should be completely independent of the high faceted values they hide. Generally, this is achieved via the introduction of randomness in the value selected to represent the sanitized value.

2. *Believability:* sanitized values should not alert the attacker to the fact that they are in a honeypot environment. This has several components. First, sanitized values should "make sense" in the context of the type of the field they are replacing. For example, sanitizing an email with the string `exwj2jv3p5` is not believable because this is not a valid email address. Additionally, the attacker should not observe different values by simply refreshing the same page; the values on each page should be consistent across multiple viewings. Finally, the system should only sanitize the high faceted value; the low faceted value should not be replaced as the attacker may have noticed the public value already on a legitimate personal account and realize the difference.

These desirable properties ensure that the attacker is not able to learn anything about the high security values. It also attempts to preserve the illusion that the attacker is interacting with the real system, which is important for accurately recording the behavior of the attacker. If the attacker becomes aware he or she is interacting with a honeypot, their behavior may change and negatively impact the quality of the results obtained by the honeypot. Ultimately, all of these properties are present in the implementation provided by this paper.

4 Jelf Overview

Creating faceted values using Jelf is fairly simple. Each model simply inherits from the `JeevesModel` class instead of the standard Django `Model`. The fields are specified as usual, but with two additions. First, there is a `jeeves_get_private_*` function. This function name is misleading[3] as it actually presents the "public" or low-security value for a field. For example, for the user profile, the `email` field should be private to the user, so there should be a `jeeves_get_private_email` function which returns whatever the public should see when not authorized to see another user's email (e.g. "redacted"). Secondly, there is a `jeeves_restrict_*` function. This allows the developer to specify the policy of who can see the "high" values for this model. This function accepts two parameters: the object being restricted and the *context*. One simple example of this function used for the implementation of the password manager is to simply assert `user_object == context`, which will only allow the user to view his or her own email. The underlying semantics that allow this interface to work properly along with the database are more complex. The `JeevesModel.py` file provides a subclass of the Django `Model`. Essentially, this class is responsible for marshaling faceted values to database rows and vice versa. This is where the majoriy of the modifications were performed in order to allow for data sanitization.

5 Method

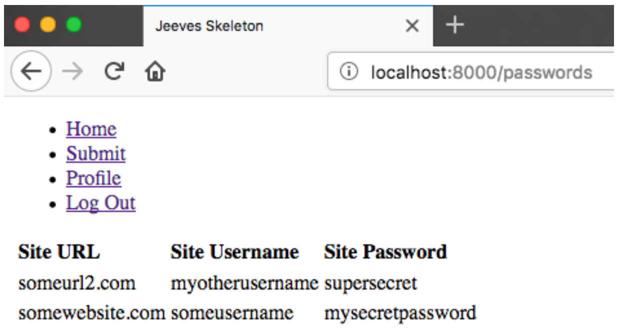
In order to add sanitization to the Jelf framework, the first step was adding state to the Jeeves library to indicate whether the honeypot sanitization mode was enabled or disabled. This was straightforward; a field `_is_honeypot` was added to the `JeevesState` class, which is global to the Jeeves library. For simplicity, a URL route was registered which simply toggled the honeypot sanitization (`/honeypot`). It is also possible to set this during startup of the web server.

[3]In fact, there is a github issue to fix its naming: `https://github.com/jeanqasaur/jeeves/blob/master/jeevesdb/DOC.md`

Finding the best spot in the Jeeves library to implement sanitization was a challenge. As a strawman approach, consider adding sanitization at the last moment, when the faceted value is being concretized. It is difficult to achieve all of the desirable properties at this stage. Specifically, picking a value which is both believable and non-interfering is challenging. An early attempt with this method involved randomly picking from a set for each type to sanitize. However, this is not believable, as refreshing the page results in different results. Hashing the high value to obtain a pseudo-random, consistent value as an index into an array of possible sanitization values violates the non-interference policy, because the attacker will be able to discern when two secret values are the same because the hashes will be the same.

A useful observation is that the "Jeeves ID" field, stored with each value in the database, is generated uniformly at random and is in no way related to the high level faceted value. This is helpful for achieving believability because this value does not change on the page refresh. For our second attempt, we considered sanitization at the `get()` and `all()` functions in the `JeevesModel.py` file, which get a single record or get all records after filtering, respectively. A hash of the Jeeves ID field was used as an index into an array of possible sanitized values. This satisfied all of the desirable properties of sanitized values and appeared to work well. Unfortunately, the solution implemented in the `all()` function did not scale well with larger sets of results, and even sets of size 8 or higher essentially froze the server. This is likely due to the way that the faceted values were organized for lists of faceted values. Basically, the organization of this structure was such that each value was a binary, nested "tree" of faceted values with N levels for N rows, where each level was a branch on the condition for the given faceted value. This leads to exponential run time in order to traverse the faceted value trees, and was too slow.

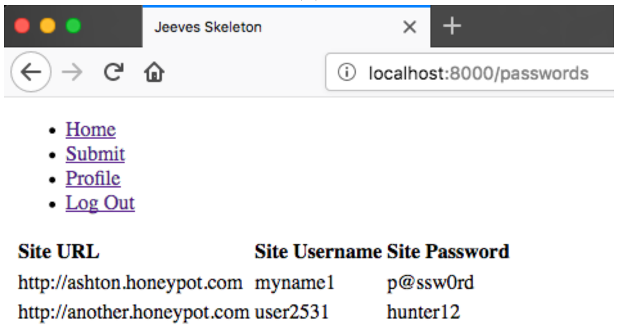
Our third attempt was finally successful. The sanitization of values for this attempt happened in the `jiter()` function, which directly queried the underlying database for values. The source code added is provided in listing 2. Basically, this `if` statement sanitizes a faceted value corresponding to a



- Home
- Submit
- Profile
- Log Out

Site URL	Site Username	Site Password
someurl2.com	myotherusername	supersecret
somewebsite.com	someusername	mysecretpassword

(a)



- Home
- Submit
- Profile
- Log Out

Site URL	Site Username	Site Password
http://ashton.honeypot.com	myname1	p@ssw0rd
http://another.honeypot.com	user2531	hunter12

(b)

Figure 1: Comparison of (a) unsanitized and (b) sanitized web pages.

model that is loaded from the database if (a) all the conditions are met (indicating the high value would be displayed) and (b) the web server is in the `_is_honeypot` state. Function names of the form `jeeves_get_honeypot_`{`fieldname`} are added to the model, where {`field_name`} is replaced by a field name of the model. This function accepts a parameter for the Jeeves ID, which is then hashed and used as an index into an array of possible values for each field. An example of a model with one of these functions is provided in listing 3. This method does not have the issue with slow run time because it sanitizes values before they are collected into the previously mentioned tree structure.

The result of these modifications are shown in figures 1a and 1b. A set of potential sanitized values for each field can be specified and selected from. Even the same values secret values may receive different sanitized values in this manner.

6 Experiments

In order to evaluate the performance implications of this modification, we compare our modified Jelf implementation with the unmodified Jelf. Furthermore, a vanilla Django version of the password manager was implemented for comparison. We also compare the response times with 1, 10, 100, and 1000 rows of password data to consider the impact of number of number of rows on performance. The results are shown in table 2. All experiments were conducted on a 2011 MacBook Pro with 6GB RAM and a 2.8 GHz Intel Core i7 processor using the `ab` utility.

The sanitized and unsanitized Jelf implementations have response times that are within 1% of one another. This suggests the modifications do not have a large impact on performance relative to the original Jelf implementation. However, both the sanitized and unsanitized Jelf implementations suffer from very slow response times at the 100 and 1000 row level, with times as slow as 7 seconds for 1000 rows. This slow performance is consistent with the results obtained in [5] during the tests of the Jeeves database performance as measured on an example conference management example implementation, but is likely too slow for practical use at this time. Furthermore, both of the Jelf implementations are orders of magnitude slower than the vanilla Django implementation.

7 Limitations and Future Work

Unfortunately, there are several limitations with the proposed method of sanitization that need to be addressed for practical use. First, if the attacker attempts to create a new saved password within the sanitized version, the newly created value will also be sanitized. Obviously this will alert the attacker that something strange is going on and may signal to them that the environment has been converted to a honeypot. To combat this issue, an additional flag

Listing 2: Code snippet from `JeevesModel.py`

```
# If all policy conditions are true, then it will present the high value
# so we need to replace it with the sanitized value
if (all(v == True for v in value_list) and JeevesLib.jeevesState._is_honeypot):
    # for each field for the model
    for field in self.model._meta.fields:
        # if atttribute has honeypot function
        if hasattr(self.model, "jeeves_get_honeypot_" + field.name):
            # call the honeypot function
            honeypot_function = \
                getattr(self.model, "jeeves_get_honeypot_" + field.name)
            jeeves_id = var_name.split('__')[2]
            honeypot_value = honeypot_function(jeeves_id)
            # assign the sanitized value to the faceted value high object
            setattr(obj, field.name, honeypot_value)
```

Listing 3: Code snippet from `JeevesModel.py`

```
def pick_among(jeeves_id, replacement_set):
    return replacement_set[hash(jeeves_id) % len(replacement_set)]

class Password(Model):
#...
    @staticmethod
    def jeeves_get_honeypot_site_password(jeeves_id):
        return pick_among(jeeves_id, ["p@ssw0rd", "hunter12", "iloveyou"])
#...
```

# of Records Displayed	Jelf (Sanitized)	Jelf (Not Sanitized)	Plain Django
1	109 (4)	110 (6)	7 (1)
10	155 (12)	152 (5)	8 (2)
100	668 (23)	634 (34)	10 (2)
1000	7368 (264)	7140 (186)	30 (3)

Table 2: Results for web frameworks with different number of passwords saved. Values are the mean of 10 runs, and parenthetical values are the standard deviation of the times of these runs. All values shown in milliseconds.

may need to be added to signify that certain values were created by the attacker and should not be sanitized. Second, it is possible that despite the small difference between the Jelf Sanitized and Jelf Unsanitized values, the attacker could still discern between the request response times. For example, consider an attacker who creates a (legitimate) account with 1000 passwords and times requests to access these passwords on the unsanitized mode of Jelf. Then, the attacker executes a brute force attack as described previously, attempting to log in as other users using common passwords in a horizontal fashion. Suddenly, they are able to log in to an account with 1000 passwords. To determine whether the successfully guessed a password or were simply redirected via the previously described trigger, the attacker could time a series of requests. As seen in table 2, they can expect times of more than 200ms slower than previously observed if on the sanitized mode, or approximately the same as before for the unsanitized mode, and thus discern their environment. The obvious solution for this issue is to make the access of the sanitized and unsanitized values virtually identical. This would probably require sanitizing the database (a sanitization-time operation) instead of the values during access (a request-time operation). The trade-off is that there may be more delay, especially for large databases, sanitizing potentially millions of records during the conversion to the honeypot versus sanitizing however many records are on the page during the request. Another idea is to inject random delays regardless of length to thwart this side channel attack, but this could have a negative impact on performance.

8 Conclusions

Sanitization is an interesting, yet often overlooked component of dynamic honeypot creation. In this work, we consider an extension of the Jeeves language and the Jelf web framework which allows for sanitization via faceted values. We implement a simple password management service in Jelf and demonstrate the efficacy of our sanitization method on this example project. By attempting several different implementation points for sanitization, it is determined that sanitization immediately after extraction from the database during faceted value creation is the most feasible option. Furthermore, we evaluate the performance of this request-time sanitization and find that Jelf with sanitization serves requests only slightly slower than Jelf without sanitization. Future work should consider the trade-off of request- and sanitization-time sanitization, and also look for improvements to the overall performance of the Jeeves language and Jelf framework.

References

[1] A. C. Myers, "JFlow: Practical Mostly-Static Information Flow Control," *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '99*, pp. 228–241, 1999. [Online]. Available: http://portal.acm.org/citation.cfm?doid=292540.292561

[2] J. Yang, K. Yessenov, and A. Solar-Lezama, "A language for automatically enforcing privacy policies," *ACM SIGPLAN Notices*, vol. 47, no. 1, p. 85, 2012.

[3] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making Information Flow Explicit in HiStar," *Osdi'06*, vol. 54, no. 11, p. 93, 2006.

[4] T. H. Austin and C. Flanagan, “Multiple facets for dynamic information flow,” *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL ’12*, vol. 47, no. 1, p. 165, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2103656.2103677

[5] J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong, “Precise, dynamic information flow for database-backed applications,” *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2016*, pp. 631–647, 2016. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2908080.2908098

[6] T. H. Austin, J. Yang, C. Flanagan, and A. Solar-Lezama, “Faceted execution of policy-agnostic programs,” *Proceedings of the Eighth ACM SIGPLAN workshop on Programming languages and analysis for security - PLAS ’13*, p. 15, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2465106.2465121

[7] A. Hirata, D. Miyamoto, M. Nakayama, and H. Esaki, “INTERCEPT+: SDN support for live migration-based honeypots,” *Proceedings - 2015 4th International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security, BADGERS 2015*, pp. 16–24, 2017.

[8] S. Biedermann, M. Mink, and S. Katzenbeisser, “Fast dynamic extracted honeypots in cloud computing,” *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop - CCSW ’12*, p. 13, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2381913.2381916

[9] M. Roesch, “Snort: Lightweight Intrusion Detection for Networks.” *LISA ’99: 13th Systems Administration Conference*, pp. 229–238, 1999. [Online]. Available: http://static.usenix.org/publications/library/proceedings/lisa99/full{_}papers/roesch/roesch.pdf

[10] N. F. Huang, C. Wang, I. J. Liao, C. W. Lin, and C. N. Kao, “An OpenFlow-based collaborative intrusion prevention system for cloud networking,” *Proceedings of 2015 IEEE International Conference on Communication Software and Networks, ICCSN 2015*, pp. 85–92, 2015.