Sanitizing Private Data for Repair Systems

by

Katherine Jien-Yin Fang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2014

Signature redacted

Author ....
Department of Electrical Engineering and Computer Science
Feb 10, 2014

Signature redacted

Certified by.................... ......
Nikolai Zeldovich
Associate Professor
Thesis Supervisor

Signature redacted

Accepted by .. ..
Albert R. Meyer
Chairman, Master of Engineering Thesis Committee

Sanitizing Private Data for Repair Systems

by

Katherine Jien-Yin Fang

Submitted to the Department of Electrical Engineering and Computer Science
on Feb 10, 2014, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The SOLAR system helps restore interconnected system integrity after security attacks with a focus of minimizing the exposure of sensitive data in the repair logs. It builds upon Warp, a repair system for Django web applications which logs all major actions taken by the server from incoming request to outgoing response, and addresses the inherent security vulnerability of logging all data and actions. It provides application developers with a way of notating particular fields as sensitive and ensures that the exposure of these fields to the logs are minimized while maintaining and improving the reexecution of code during repair operations. A series of tests were written to show that SOLAR continues to support repair operations even with certain data removed.

Thesis Supervisor: Nikolai Zeldovich
Title: Associate Professor

Acknowledgments

I thank my advisor, Nikolai Zeldovich, for his support throughout this process from advice on the work itself through guiding the thesis organization.

I would also like to thank Ramesh Chandra for his help in understanding the Warp system. Additionally, I would like to acknowledge him and Taesoo Kim for their work on implementing the Warp recovery system for Django, allowing me to build upon it.

Thanks to my family and friends, especially the residents of Epsilon Theta, for supporting me through this process. Particular thanks goes to my habit party for keeping me on track by saving polar bears, to Adam Hesterberg for waking me up on all the important days, to Alwina Liu for showing me how to grow tomatoes, to Melissa Ko and Anders Kaseorg for motivating me through puppies and kittens, and to Sarah Gontarek for nagging me about dungeons.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Although web security is better understood than it once was, web applications still have many vulnerabilities discovered daily including cross-site scripting and SQL injections [6]. Even when the code is correct, it is possible for site administrators to configure the site improperly, potentially allowing malicious users access or privileges that should have been denied. Using those permissions, a malicious user could make unauthorized changes such as installing long-standing scripts. Discovering the extent of the harm done and reversing the effects is often difficult. In addition, more and more applications interact with one another and other web services today, allowing users to use protocols such as OpenID [2], OAuth 2.0 [1], REST or other APIs [9, 8, 11, 12] to access and manipulate information across domains. A vulnerability in one application could compromise other connected services.

Warp [4],[1] a recovery system for Django applications [7], addresses these issues. Once a request that exploits a security vulnerability has been identified, Warp can be used to rollback the effects. Warp can also help administrators recover from configuration mistakes by making it possible to retroactively cancel or alter requests to the server. Any actions that would have been prevented from proper configuration would be rolled back while all authorized requests wold remain valid. Warp also works in a distributed setting, capable of rolling back requests across servers [13, 5].

To accomplish this, Warp records the requests and responses made by users inter-

[1]Web Application RePair

acting with the application. These are stored in log files on disk. Database updates are also saved using a versioned database. Major actions such as database queries and remote requests to other servers are also stored in the log files, allowing Warp to track input and output dependencies. To perform a repair, Warp rolls back to a previous state and re-executes commands.

Although this approach allows recovery from a large number of undesirable situations, by logging the requests and responses on disk, Warp inadvertently introduces a potential security vulnerability. That is, sensitive user data might be logged in the logs themselves. For instance, many applications will send the password a user inputted to the server to be checked against the database. This data is often submitted via a form, which will be logged by Warp with the rest of the request in plaintext. Even services which use HTTPS to prevent eavesdroppers from reading packets to discover username-password combination or other sensitive information are subject to this vulnerability. This is due to the fact Warp operates alongside the application on the server, allowing it to read and record the requests as the server sees it – in plaintext.

This thesis presents SOLAR[2] as an improvement to Warp. The main issue that SOLAR addresses is removing sensitive data from the logs. The difficulty here is to remove the sensitive data while maintaining the ability to rollback and re-execute during repair. In order to do this, SOLAR relies on the application developer to identify the sensitive information that should be removed. Once identified, the system can automatically remove the information from the log while recording and replaces it with an alphanumeric token. This token can map to a single value that is stored in the database to help the program recover during repair. However, since the use cases of sanitizing sensitive data might be different, SOLAR also relies on the application developer to set the token's value as well as specify how to use that value during repair. This thesis breaks this challenge into two parts: user-inputted data from form data and HTTP basic access authentication, and server generated data such as secret keys in OAuth 2.0.

[2]Sanitation Of Logs in Asynchronous Repair

Another issue is that Warp is incapable of replaying sequences with randomized effects affecting multiple requests such as session identifiers generated on authentication. However, this type of request often is accompanied by sensitive data such as passwords. To enable repair these types of requests, SOLAR logs randomization.

Chapter 2 gives an overview of the components used in the implementation of SOLAR. Chapter 3 discusses the design of log sanitation and randomization logging, and Chapter 4 discusses the implementation of these designs. Chapter 5 presents an evaluation of SOLAR by offering an example of what must be done to alter a Warp application to be compatible with SOLAR and addresses how well the system works. Chapter 6 gives an overview of related work while Chapter 7 includes a discussion of the limitations and possible extensions of SOLAR. Finally, Chapter 8 concludes.

Chapter 2

Background

SOLAR builds upon a version of Warp that repairs Django applications. It also discusses the particular use case of cleaning authentication and authorization data from HTTP basic access authentication and OAuth.

2.1 Django

Django is a web framework written in Python [7]. Applications written in Django can be easily dropped into different Django projects, allowing them to be ported from one website to another with little effort. Django also provides several built-in features to help developers such as a model system to interface with the database, a default authentication backend, and a framework for handling forms.

Models allow developers to structure their data and interface with the database without directly writing queries. A model can have many fields in which data can be stored. Each model corresponds with a table in the database, and each field corresponds with a column. An application can generate new model objects based on user input as well as query these model objects and update them. Developers can also provide validation for each field entry of the model as well as across the different fields and on the overall model itself.

One often used model provided in Django is the `User` model used in the default authentication backend. It is capable of storing information such as first and last names and emails. More importantly for authentication, it stores username-password combinations where the password is stored as a string that encodes the algorithm used to hash the password, the salt, and the hash of the password. Each user is mapped to a `User` model object which is created when the user registers. Later, when a user inputs his or her information to log back into the website, the inputted password is hashed the same way and compared to the password stored on the `User` model object identified by the inputted username.

Django also provides a structured way of dealing with forms. Forms can have fields as well as validators for each field and for the form as a whole. Perhaps unsurprisingly, forms can be backed by models. In this case, the form's fields will be drawn from the fields of the model, and validation of the form calls back to the model's validation methods. Django forms also provides methods to turn the form into HTML to print out to the user as well as binding the user's input back to the form for easy handling and validation. Some forms have also been provided by default for dealing with user registration and authentication backed by the `User` model.

In addition to all the built in features, the Django community has also developed many other importable features. Two used in evaluating SOLAR are Tastypie [3] and OAuth2App [10]. Tastypie helps developers create a REST-style interface to the application's data and can accept authentication through basic access authentication. OAuth2App provides a module to help Django developers provide an OAuth 2.0 interface to their application.

2.2 Basic Access Authentication

Basic access authentication allows an HTTP user agent to provide a username password combination when making a request. This information is passed in the Authorization header and is simply Base64 encoding of the username and password separated by a colon. It provides a way to pass authentication data that does not require forms, cookies, or session identifiers, but it is the server's job to check whether or not the header data actually authenticates a user. Basic access authentication provide authentication data in both Tastypie and OAuth2App which are used to evaluate SOLAR

2.3 OAuth 2.0

OAuth 2.0 is a protocol to allow distributed authorization [1]. The purpose is to allow web applications, called clients, to access protected resources, of other web applications, called resource providers, so long as the user, or resource owner, grants permission through an access token. The protected resource might be an e-mail address or the ability to send an e-mail on your behalf.

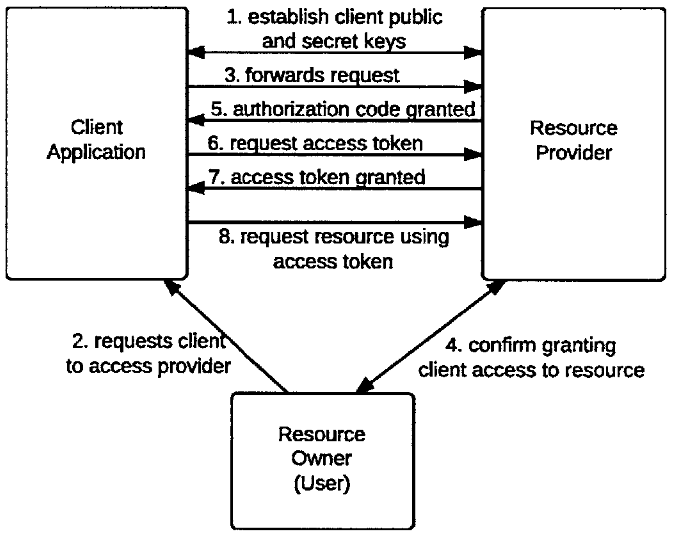
1. establish client public and secret keys
3. forwards request
5. authorization code granted
Client Application
6. request access token
Resource Provider
7. access token granted
8. request resource using access token
2. requests client to access provider
4. confirm granting client access to resource
Resource Owner (User)

Figure 2-1: Sequence of events of how the client, resource provider, and resource owner interact in OAuth.

For this to work, client applications must first get a client public and secret key from the resource provider. With this information, the resource provider will be to authenticate later requests from the client. This set of keys does not not expire, and is very long-lived. The set is unique between client and provider. That means that if multiple users are own separate resources on a given provider and want the same client to be able to access them, the client can use the same set of keys to access each user's respective resource from the provider.

This set of keys, however, is not enough. This authenticates the client with the provider, but OAuth also requires that the resource owner grant the client authorization to access the resource. To do this, the resource owner can grant a client application an authorization code with an optional scope. The authorization code does not actually grant permission itself, but is actually a short-lived, one-time code that enables the client to retrieve an access token from the provider. The scope determines what sort of access the resource owner is allowing the client application.

To use the authorization code, the client needs to send its client public and secret key to the resource provider along with the authorization code. The resource provider will then provide an access token as well as a refresh token.

Once a client has the access token, it can send the access token along with the client public and secret key to the resource provider which will then allow the client access to the protected resource. The access token also has an expiration, but the refresh token can be sent to the server to extend the expiration date of the actual access token.

In this thesis, we use OAuth2App, a Django module that provides Django applications with an OAuth 2.0 interface.

2.4 Warp

Warp is a system that allows administrators to recover from exploited security vulnerabilities and misconfigurations [4, 13, 5]. It has two modes: record and repair. During normal operation in record mode, data is stored in a versioned database, and Warp logs records for import events such as receiving a request or querying a model. In repair mode, the application receives repair commands which triggers repair. These commands can add a new request back in time, replace data that was sent in a request, or delete an old request.

If a repair command replaces data of a given request or creates a new request, the state of the server is rolled back to right before that request was sent. The request is then started with the new data, and the rest of the processing of the request is replayed. If that request happens to affect other requests, such as modify a model that is later read by a different request, then those requests are also rolled back and replayed.

When starting the server in repair mode, the server will also build the action history graph from the logs. There are Client and Django `Actors` which keep track of the state of processing a particular request. There are also `Actions`. These each correspond to one of the logged events. Each `Action` is responsible for understanding how it rolls back and also how it executes repair.

This action history graph tracks dependencies between models. For example, if a particular model object's data is modified in one repair request, and a later request reads the data, then the later request will also be queued for re-execution. Warp also tracks dependencies across servers by storing information about the remote requests and responses. However, Warp treats the actual HTTP requests and responses as independent. That is to say, in the actual course of an application it is common for the data in one response to become input in the next request. For example, a user will receive a session id once authenticated. This session id is passed along in the user's later requests. By not tracking this sort of dependency, if the data in the response is changed, it will not propagate the changed data to the next request. In the previous example, this means that if the user receives a new session id during repair due to randomization, the later requests will still be using the old session id. The later requests will probably be triggered for replay because the session is backed by the same model object, but the user's legitimate requests will be discarded as unauthenticated.

To allow a server to accept both normal requests and repair requests, Warp runs a repair manager. The repair manager acts as a proxy to the actual server spins up the actual server in either repair mode or record mode based on the requests it is receiving.

Chapter 3

Design

This chapter focuses on the design of Solar. It first handles the case of dealing with log sanitation from user input perhaps from user registration or authentication. Then, it details the design for the case when the server also generates sensitive data as is the cause when an OAuth 2.0 server generates keys and access tokens for clients. Finally, it deals with how to enable the repair system to successfully re-execute authentication and authorization requests.

3.1 Log Sanitation for User Input

One of the main challenge that Solar addresses is the fact that Warp saves username-password combinations in plaintext in the logs. This is a rather unfortunate breach of security. It is hard to know what fields need to be sanitized. Once the fields are sanitized, the repair system still needs to maintain enough information to re-execute the code without the original data. This problem can be broken down into three parts:

1. Knowing what to sanitize,
2. Removing the sensitive data from the logs, and
3. Ensuring repair still works without the original data.

We discuss these issues and their solutions in two different contexts. First, we discuss this in the context of receiving authentication through POST data from user-submitted forms. For this, we use Django's built-in `User` model and related Django forms. Second, we consider these issues when receiving authentication in headers because of basic access authentication. For this particular implementation, we consider basic access authentication in Tastypie.

3.1.1 Forms

While it is easy to pinpoint which field of the default authentication form is the password and therefore sensitive field, it is not generally easy to know what field(s) are should be sanitized. To support generic forms, and not just default Django forms, SOLAR relies on the developer to denote what fields are sensitive. To do this, we have augmented Django forms fields to have an optional 'sensitive' field, which is set to `False` by default. Once the POST data has been bound to a form, it is possible to identify what needs to be sanitized.

To address the issue of removing sensitive form data, we first needed to buffer the logs. Keeping the previous implementation of immediate logging would mean that the Django framework needed to know what data in the **http_body** was private prior to forwarding the data to the application. While this is possible, it destroys the notion of separating the framework from the application.

We also introduced the concept of a token. Whenever a field is denoted as sensitive, its data is replaced in the logs by a token string of the form `token$token-identifier` where `token-identifier` is a length 32 alphanumeric string. The token string maps to a developer-set value in the database which can be used to help replay the system.

A concrete example of the changes necessary to make an application compatible with SOLAR can be found in Section 5.1.

3.1.2 HTTP Basic Access Authentication

Basic access authentication is in many cases easier than dealing with forms. We know exactly what data needs to be cleaned, the data from the Authorization header, and this can be removed as soon as the server receives the request. To ensure that repair works afterwards, we use the same token methodology used with forms, relying on the developer to provide the token with which, if any, user was authenticated.

3.2 Server-Generated Confidential Data

The challenge presented in sanitizing server-generated data is that the entry point of the data is unknown. The example used in this section will be OAuth 2.0. While the actual data retrieved using OAuth 2.0 might be considered sensitive, this section focuses primarily on sanitizing data that is generated by supporting the OAuth 2.0 interface and would be present in any such application.

OAuth 2.0 has many different keys, codes, and tokens of varying degrees of sensitivity. For example, each client receives two keys – one which is public, and one which must remain secret. These keys are generally long-lived, potentially set up at the time the client web service is set up and never again touched. With knowledge of both the public and secret key of a client application, a malicious user could forge requests from the client to the resource provider. If the malicious user were to further find any access tokens, they will have essentially gained the permission from the resource owner to access the corresponding protected resource on the provider. While these access tokens can be shorter-lived, they can also be continually refreshed and kept around for a long time.

The goal is to completely eliminate these client secrets and access tokens from the logs while still maintaining the ability to perform re-execution of authorization requests.

To address this challenge, SOLAR makes extensive use of `Tokens`. Anywhere the log would write down sensitive data, we instead replace it with a token string. As before, the token has a repair value to help with re-execution using the `Token` model.

Because the data we are dealing with is largely already stored in the database, it is permissible to use the actual value presented during the actual execution as the repair value. The issue then becomes tracking down all the areas in which the sensitive data exists. We found four different areas: basic access authentication, storing and querying model objects, generating keys, and in HTTP responses.

Removing data from basic access authentication is essentially the same as in the case when the sensitive data is generated by users rather than the server. This is because the client is the one sending the Authorization headers and is essentially acting as a user of the OAuth 2.0 server. The actual data from the header is replaced with a token, which then stores the raw header data as the repair value.

To help deal with logging model object data when saving and querying model objects, SOLAR introduces the optional parameter `sensitive` for model fields. When logging a model, the data stored in `sensitive` field are replaced with a token. The actual data is stored as the token's repair data and is passed to the BufferedLog. When the BufferedLog is flushing to disk, it can search the other log entries for sensitive data and tokenize it before writing to disk. This allows SOLAR to tokenize log records corresponding to key generation as well as remove keys sent in the log record for HTTP responses.

3.3 Replaying Requests

Another minor challenge SOLAR must address is the issue that Warp cannot replay authentication sequences successfully as mentioned in Section 2.4. This occurs both when dealing with sensitive data from the user as well as from the server. This happens because randomized functions are called in record and repair mode, but due to their non-deterministic nature, a different result is returned on each call. This often causes failures to occur and possibly kicks off more requests to be repaired than necessary.

For instance, if an authentication sequence was replayed and the session id was changed, this would kick off repair for all requests touched that backing session model object. However, due to the fact response-request data dependencies are not tracked, these would now have the wrong session id, and the requests would appear unauthenticated. An example of this occurring in OAuth 2.0 is when an access token gets generated differently. Subsequent requests that would have worked using that access token would now be denied.

In the case that the user should have been authenticated, the result of the user's later requests should also be the same. There are two possible solutions to deal with this problem. One would be to propagate the new data from response to other requests. Another way would be to log randomization to ensure that the repaired session id is the same. SOLAR follows the second approach. The advantage here is that the session id would remain the same, so the user's request would not need to be re-executed unless something other than the session id changed.

In particular, SOLAR logs the results of calls to non-deterministic functions like `random.randrandom()`. During repair, these results are then fed back to the application to simulate the environment that the original handling of the request occurred.

Chapter 4

Implementation

This chapter discusses the actual implementation of SOLAR. It begins with the implementation of `Tokens` and moves on to discuss the BufferedLog. It then details how to handle basic access authentication requests which appear both in the user- and server-generated sensitive data cases. After this, it discusses the specific modifications for the user-generated case followed by the specific modifications for the server-generated case. Then, the implementation for logging randomization is discussed. Finally, we summarize the code changes in a table.

4.1 Tokens

The goal of tokens is to replace sensitive while storing enough information to help with re-execution. Each token has associated with it three pieces of information: the token identifier, the repair value, and the raw value, and the repair value. The token identifier is a length 32 alphanumeric string and uniquely identifies the token with a `Token` model object, shown in Figure 4-1 stored in the database. The repair value is a developer-specified value that helps the system recover. It is not stored in the logs, but stored on the `Token` model object instead, and it is initialized to "unset." The raw value is never directly stored on the `Token` model object and is ultimately removed from the logs and replaced with a token string of the form `token$token-identifier`. During the original execution of the code, however, hav-

```
class Token(models.Model):
    token = models.CharField(max_length=128)
    value = models.CharField(max_length=128)
```

Figure 4-1: The model definition for Tokens.

ing the raw value is also important to perform actual computations. To retain the raw value as well as the token that will replace it, augmented token strings of the form `token$token-identifier$raw-data` are used in record mode.

Since the `Token` model is just another developer specified Django model, it is stored in the same database as the rest of the Django project. And like all models, it requires its containing application, `django.warp`, to be added to the list of `INSTALLED_APPS` in `settings.py`. However, it is undesirable to have the `Token` model be versioned. If the `Token` objects also rolled back, the `Token`'s stored value would not be available at the point of re-execution at which it was needed. To prevent this, SOLAR checks whether or not a model belongs to the application `django.warp`. If it does, it skips the steps needed to version the backing database table.

During ordinary operation while recording is enabled, the application will eventually need to access the raw input. At this point, the application can ask SOLAR to give it the `Token` as well as the raw value via the `parse_token_string` call. Given an augmented token string, the method will return a tuple of the associated `Token` model object and the raw value. An input of a regular token string will result in returning a tuple with the `Token` object and `None` for the raw value. If the input is not actually a token string, the method will simply return `None` for the `Token` object and the whole input as the raw value. This means that even when SOLAR is disabled and therefore the system isn't producing `Token`s, the regular flow of the program can still proceed.

With the token and raw value, the application can then compute on the raw value and set the `Token`'s repair value field. Presumably, the application already needs to transform the sensitive data somehow to store in the database, so storing this same value for the repair value should be fine. This does mean that it is impossible to replay the computation of that value from the raw value.

It is also assumed that the application has been modified for repair. An application can detect whether or not it is in repair because it will only have a token string and not an augmented token string to pass to `parse_token_string`. The application can then grab the `Token`'s stored repair value to help with the re-execution.

4.2 Buffered Logging

Warp logged events as soon as they happened. However, at the time the request data comes in and is logged, the Django framework often cannot be certain whether or not the data is sensitive. Instead of logging events immediately, SOLAR buffers the event records. Later, when it receives an `http_end` record, which designates the HTTP response to be returned, the BufferedLog post-processes all of the records and flushes them out to the log file. To successfully post-process, the BufferedLog stores some state based on what sensitive data there is.

4.3 Basic Access Authentication

The authentication data that comes in via HTTP basic access authentication is relatively easy to deal with. Receiving a request triggers the system to record the data in an `http_start` event. As soon as the BufferedLog receives this event, it checks for an `HTTP_AUTHORIZATION` header. If it exists, the data is replaced with an augmented token string. Because the username-password combination is Base64 encoded, it is impossible to separate out and just tokenize the password. Whenever the header is accessed, the augmented token string will be returned instead of the original raw data. When the BufferedLog writes to disk, the `http_start` record is then processed, replacing the augmented token string with a normal one.

In the regular authentication case, the repair value stored is what user, if any, was authenticated. This means that the code must first check if its in repair mode, and if it is, simply return the user identified by the repair value. If it is in record mode, it must store what user was authenticated. SOLAR cannot simply store the Base64 encoding because if a malicious user were able to break into the database and found these repair values, he or she would be able to learn more information by looking at the `Token` models than could be learned from the vanilla application. In particular, the malicious user would be able to decode the values and retrieve the valid, plaintext authentication information for users.

In OAuth 2.0, the repair value maps directly to the Base64 encoding of the client public and secret keys. This is allowable because the client secret key is already stored with the client public key on the `Client` model in the database. So, in the event of a database breach, the malicious user would not be able to learn more from the data from the `Token` models than from the existing models. Upon seeing the token while repairing, SOLAR can easily replace it with the original header data, and OAuth2App can compute the validity of the header as easily as it computed it during the original run. A better OAuth 2.0 implementation would probably store a cryptographic one-way hash of the client secret key. To deal with that, the token repair value would need to be changed to which client, if any, were authorized.

4.4 User-Generated Data

User input usually enters the server via forms. In Django web applications, this translates to user input getting bound to Django forms.

4.4.1 Forms

Forms fields have been modified to include an optional 'sensitive' field. Upon binding data to a form, it becomes possible to determine whether or not that data is denoted as sensitive. At this point, the form takes the names of the fields that need to be tokenized and passes them to the BufferedLog via the `tokenize_fields` method. This method only checks for whether or not the field exists in the `http_body` and not in other events. If the particular field name is found in the `http_body`, a token

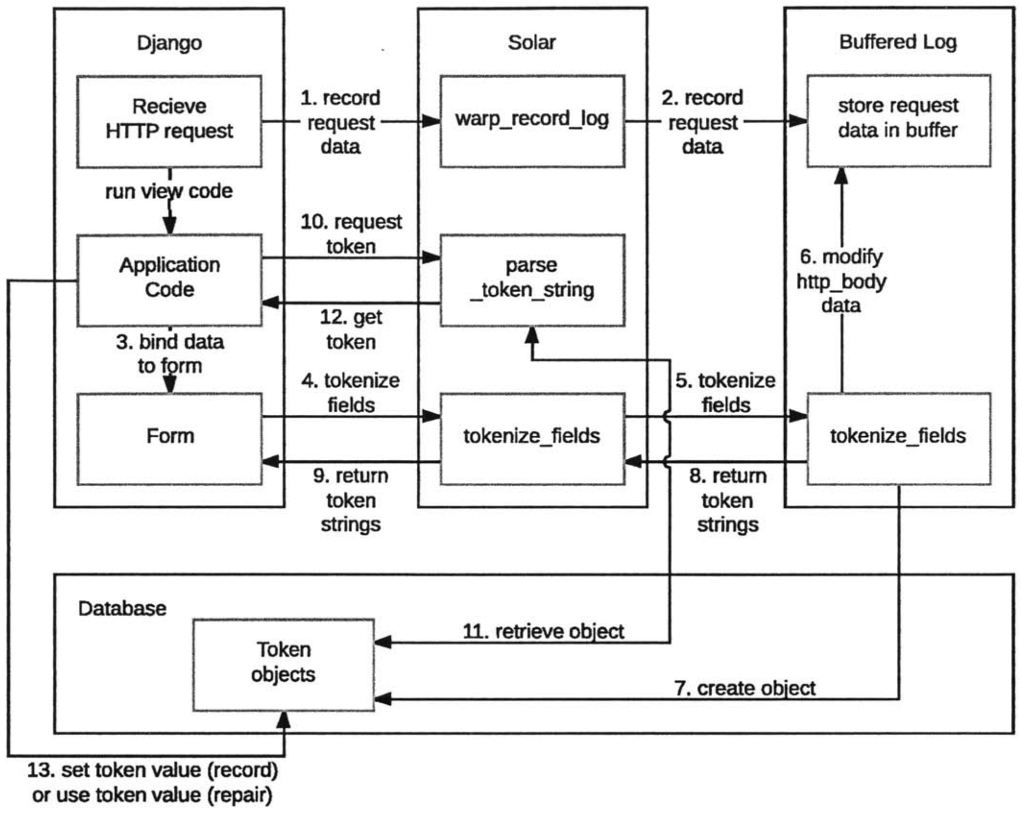

Figure 4-2: Diagram of how tokens interact with application code and Warp.

is generated, and the field's corresponding data in the `http_body` record is replaced with the new token's augmented token string. The `tokenize_fields` method finally returns with a dictionary mapping the field names to the augmented token string that replaced the field data. The rest of the application can now deal with the newly tokenized data. Figure 4-2 depicts the flow of data regarding tokenizing.

When it is time for the BufferedLog to flush to disk, SOLAR goes through and post-processes the records. To do this, the BufferedLog remembers all of the tokenized fields and will strip the raw value from the augmented token strings in the `http_body`, leaving just the normal token string.

4.5 Server-Generated Data

This implementation closely follows what modifications are necessary to be able to repair the OAuth2App module. Specifically, it considers how to remove the client secret key. Removing it from the Authorization header on basic access authentication requests has already been discussed in Section 4.3. The other three areas the secret key appears is dealing with model operations, in key generation, and in logging HTTP responses.

4.5.1 Models

It is also important to remove sensitive data when logging a record which stores sensitive data in its fields. This happens when the `Client` model for OAuth2App is saved. In particular, it holds the client secret key. Similar to how SOLAR deals with forms, it is the developer's job to mark which fields on which models are sensitive. To do this, the base `Field` class has been modified with a new optional named argument, `sensitive`, which defaults to `False`. A developer can simply mark a field, such as the `secret` field on the `Client` model, as sensitive by passing in `sensitive=True`. This is shown in Figure 4-3.

With fields tagged, SOLAR can now remove the sensitive data on model operations. This includes saving the model object in the first place as well as dealing with later queries. These correspond with the `model_save` record and the `queryset:iter` record respectively. Luckily, these two record types call `model_value` in `django.warp.model` to marshal the data values and ignore Warp-added columns such as `_end_time`.

In SOLAR, this method is modified to check for whether or not a field is sensitive. If it is, then the value is replaced with a token string, and the raw value is stored as part of the mapping on the corresponding `Token` model object. Once again, since the sensitive data is already stored in the database, it is not unreasonable to store it on the `Token` model object so long as it does not get written to the logs.

Finally, upon discovering a secret field, the method also informs the `BufferedLog` of the sensitive data. This allows us to deal with scrubbing the data in other log

```
  class Client(models.Model):
      name = models.CharField(max_length=256)
      user = models.ForeignKey(User)
      description = models.TextField(null=True, blank=True)
      key = models.CharField(
          unique=True,
          max_length=CLIENT_KEY_LENGTH,
          default=KeyGenerator(CLIENT_KEY_LENGTH),
          db_index=True)
      secret = models.CharField(
          unique=True,
          max_length=CLIENT_SECRET_LENGTH,
          default=KeyGenerator(CLIENT_SECRET_LENGTH),
+         sensitive=True)
      redirect_uri = models.URLField(null=True)
```

Figure 4-3: The `Client` model of `oauth2app` modified to mark the `secret` field sensitive.

records.

4.5.2 Key Generation

Since the client secret key and access token are both randomized strings generated on the fly, it is important to log their generation for the purpose of re-execution. However, this means that this is yet another location to sanitize the sensitive data from. These keys are generated when the corresponding model object is created. This means that the generation is followed by a `model_save` operation while processing the same request. Although it is difficult to know upon generation whether or not the randomized string will be a public or a secret key, we will know which is which once the `model_save` record occurs. Now that the `model_save` informs the BufferedLog of the sensitive data, when flushing the log, SOLAR can check for that particular string of data in any of the `wrapped_function` log records. Upon finding any, SOLAR rips out the sensitive data, and replaces it with a token string for the log. And similar to `model_save`, it stores a `Token` model object to map the token identifier to the raw data.

4.5.3 HTTP Response

The last area the client secret key appears is in HTTP responses. The processing for this happens as the log is written to disk. Essentially, SOLAR performs a search for any sensitive data, replaces it with a token string, and maps the corresponding Token to the raw data.

4.6 Logging Randomization

Logging randomization became necessary to replay authentication sequences. To do so in the user generated data case, SOLAR wraps random.randrange() as seen in Figure 4-4, which is base call for most of the calls for randomization in Django, using a decorator. The decorator allows us to reuse this method of wrapping for other sources of randomness.

```
from warp import log as warp_log

class Random(_random.Random):

    ... other method definitions ...

    @warp_log
    def randrange(self, start, stop=None, step=1, int=int,
                  default=None, maxwidth=1L<<BPF):

        ... method definition ...
```

Figure 4-4: Code demonstrating how to wrap the random.randrange() function.

When in record mode, the decorator notes that it has begun logging a function, then calls the wrapped function. At the end, it toggles off the logging indicator and has SOLAR store the result along with the name of the wrapped function. If at any point the execution wants to log another function and the indicator is still on, meaning that this call is a subcall to an already wrapped function, SOLAR will simply perform the execution without storing the result.

A newly implemented action, the `WrappedFunction`, allows these log entries to be added into the action history graph. They are stored as a dictionary mapping the function name to an ordered list of `WrappedFunctions`, each representing a value returned during the original execution. These are then connected to the Django Actor that handles the request. So when a wrapped function now calls through the decorator code during repair, SOLAR returns the next value rather than making a call to the actual function.

The decision to only store the result of the outermost wrapped function is due to the fact on repair, only the outermost wrapped function will be returned. This is because none of the wrapped function's code will actually be executed, so if the decorator also logged the results to sub-calls, those results would not be popped off their respective lists. This causes future calls to wrapped methods to return the wrong value.

There are a few fine points to bad addressed about this system. While adding the decorator makes it possible to replay sequences that could not previously been replayed, adding this decorator to functions requires much care. In particular, the wrapped code should be as small as possible because the code inside is not actually replayed on repair. This means that if the wrapped function's return value were to change on replay, this would not be caught. Furthermore, any side-effects such as manipulating models would be lost.

To enable the re-execution of authentication sequences, this decorator was added to `random.randrange()`. Unfortunately, this was not enough as the Django's session module also calls `os.getpid()` as a source of randomization. Luckily, the decorator makes it easy to wrap any function. Since it was difficult to add the decorator to the `os` module, we instead added it to the function which called it, `_get_new_session_key`.

Improvements to this wrapped logging would include making the interior code replayable as well as being able to log the `os.getpid()` directly rather than the surrounding function. It would probably also be better to log the arguments that were passed to the wrapped function and return logged results based on this in addition to the function's name.

4.6.1 Server Initiated Randomness

Because nearly every model in OAuth2App creates a randomized key, implementation of the re-execution of OAuth 2.0 also relies on randomization logging. Unfortunately, the logging on `random.randrange()` does not catch this randomization. Instead, OAuth2App implements a KeyGenerator which creates keys by taking the sha512 hash of a universally unique identifier (UUID). When creating a length n randomized key, the KeyGenerator returns the first n characters of the hash. To record this, we applied the same decorator from Section 4.6 to OAuth2App's KeyGenerator.

While it may seem preferable to apply the decorator to the UUID generator, `uuid.uuid4()`, because it is a Python module and not in a specific Django module, applying it to the KeyGenerator makes it easier to later remove sensitive data from other logs. Attempting to track down the UUID creation call that generated secret data from the resulting data stored on models is trickier than simply matching the key generated. Leaving the UUIDs logged is not an option because if discovered, it is easy to compute the sha512 hash of UUIDs found and generate valid keys.

4.7 Code Change Summary

The code change for SOLAR is summarized in Table 4.1.

Component	Description	Lines of Code
Solar	Solar code implemented on top of Warp	768
Django	Modifications made to Django framework	119
Python	Changes to Python modules	30
Documents	Example Django application	26
OAuth2App	Example Django OAuth 2.0 application	47

Table 4.1: Table summarizing code changes.

Chapter 5

Evaluation

This section provides answers to the following questions:

- What needs to be changed in an application in order to work with Solar?
- Can SOLAR correctly replay requests containing sensitive data without leaking information to the logs?
- What fields can be successfully sanitized?

5.1 Example Application

There are several steps that a developer needs to take to prepare an application for SOLAR. The following details the changes made to Django's built-in user authentication system as an example of the necessary modifications to make an application compatible with SOLAR. In particular, Django provides a `User` model along with several default forms found in `django.contrib.auth.forms` such as `UserCreationForm`, a user registration form, and `AuthenticationForm`, a log in form. The developer does need to expose the forms to users with views as well as bind input to the forms once the data is posted, but since SOLAR does not require modifications at the views level, the specific views will be omitted. Furthermore, since the modifications needed for user creation and login are similar, this section focuses only on authentication.

```
    password = forms.CharField(label=_("Password"),
                               widget=forms.PasswordInput,
                               sensitive=True)
```

Figure 5-1: Code demonstrating making a form field sensitive.

The first major change is marking the form's `password` field as sensitive. As seen in Figure 5-1, this is as easy as setting the optional `sensitive` field of a form's field to `True`.

The second necessary step is setting the value on the token during record and modifying control flow for repair. For authentication, this starts in the `check_password` method which can be found on the `User` model. This method takes in the password as entered into the form ultimately returns whether or not it matched the password stored on the `User`. However, to do so, it first performs a backwards compatibility check and updates the password on the actual `User` model to support the newer standard of storing passwords if it passes the old password check. Otherwise, the data is passed to a second method, `check_password2`, which performs the actual check.

As seen in Figure 5-2, only five lines were changed in `check_password`. At the beginning of the method, the token string was parsed into an actual `Token` object and the raw value using the utility method `parse_token_string`. If the raw value was actually returned from the utility method rather than a `None` value, then we know that the server is currently operating in record mode rather than repair mode. The operations that would have been performed without SOLAR are performed. The final computation, the password in its new standardized form, is stored in the `Token` using the utility function `set_token`.

During repair, this backward compatibility step is skipped, but the `Token`, whose value is in the new standard, is forwarded on to `check_password2`, which assumes that the password is in the new standard.

The objective of `check_password2` is to actually compute whether or not the inputted password matches the `User`'s actual password. The actual modifications

```
  def check_password(self, raw_password):
+     token, raw_password = parse_token_string(raw_password)

      # Backwards-compatibility check. Older passwords won't include
      # the algorithm or salt.
-     if '$' not in self.password:
+     if raw_password and '$' not in self.password:
          is_correct = (self.password ==
                        get_hexdigest('md5', '', raw_password))
          if is_correct:
              # Convert the password to the new, more secure format.
              self.set_password(raw_password)
              self.save()
+             warp.util.set_token(self.password)
          return is_correct

-     return check_password2(raw_password, self.password)
+     return check_password2(raw_password, self.password, token)
```

Figure 5-2: Code modifications to make check_password compatible with the SOLAR system.

```
- def check_password2(raw_password, enc_password):
+ def check_password2(raw_password, enc_password, token=None):
      algo, salt, hsh = enc_password.split('$')
-     return constant_time_compare(hsh,
-                 get_hexdigest(algo, salt, raw_password))
+     # if repairing, grab the hashed password from the token
+     if raw_password:
+         hashed_password = get_hexdigest(algo, salt, raw_password)
+         warp.util.set_token(hashed_password)
+     elif token:
+         hashed_password = token.value
+     return constant_time_compare(hsh, hashed_password)
```

Figure 5-3: Code modifications to make check_password2 compatible with the SOLAR system.

can be found in Figure 5-3. First, the password is parsed from its stored form to read out the algorithm, salt, and actual hash. This is left unmodified. The final line in the original code is to hash the inputted password and use a constant time compare to check the hashes. In the modified code, this has been split into two separate parts: finding the hash and comparing the hashes.

In record mode, finding the hash follows the same methodology as the unmodified code. This can be done because the `raw_password` parameter is the raw value that the user submitted in the authentication form. The method can simply use `get_hexdigest` using the algorithm, salt, and raw input to find the hash. Since `raw_password` will not exist while re-executing this code in repair, the `Token` needs to be used. While in record mode, the hash digest is stored as the `Token`'s value, which can then be retrieved during repair.

Finally, the method uses the same method, `constant_time_compare` to determine whether or not the hashes are the same. This allows replay on authentication unless modifications have been made to how to hash a password given the algorithm and salt.

5.2 Correct Re-execution

To demonstrate that SOLAR is capable of correctly replaying requests without leaking sensitive information to the logs, several tests were implemented.

5.2.1 Documents Application

For the single sever, we implemented a simple documents application that allowed users to log in, create documents, and edit them. The user authentication system used was primarily Django's default authentication system with a slim view around it to interact with the forms. Users could log in during normal execution and create and edit documents. Warp could support replaying or cancelling document creation and editing, but could not deal with the re-execution of the actual authentication. To do this, Warp also needed to record the plaintext user-inputted password in the log.

SOLAR successfully replaces the password from the `http_body` record with a token string. Furthermore, it is possible to either cancel the request or re-execute it such that the user has the same session identifier. Any other requests from that user with the same session identifier can then be safely skipped.

We were also able to implement a Tastypie REST-ful API on top of the documents application. This would allow users to pass their username-password combination through the Authorization header to edit documents that they owned. Warp could handle this test case without SOLAR, but would log the base 64 encoded username-password combination to disk. SOLAR is capable of tokenizing the header and re-executing the authentication sequence successfully.

5.2.2 OAuth 2.0

For OAuth 2.0, we looked at a module called OAuth2App. Warp was capable of recording authentications using OAuth2App as well as cancelling invalid OAuth 2.0 operations, but it could not handle re-execution of authorization requests.

To evaluate whether or not OAuth 2.0 would work with SOLAR, we created the following workload:

1. Create account with OAuth2App, the resource provider.
2. Create client (generates public and secret key) for a client application.
3. Create authorization code for accessing user's email for the client.
4. Redeem the authorization code for an access token.
5. Use the access token to get the email address registered with the account.

To determine whether or not re-execution worked, the repaired database was checked against the results produced during regular execution. That is to say, we ensured that the same client keys, access token, etc. were produced from the repaired system. Performing re-execution beginning at any of these step was done successfully.

We also ensured that the client's secret key was removed entirely from the logs. The only points at which the client's secret key are exposed are the same ones they would be exposed at without using the repair system. More specifically, the client key can only be found in the database and in the actual returned HTTP response.

5.3 Sanitizable Fields

It is probably possible to remove all application data from the logs by replacing them with tokens that map to the actual data using `Token` model objects. This would effectively remove all data from the logs. However, for each model field sanitized, there can be potentially many `Tokens` associated with it which would bloat the database unnecessarily.

The most care was needed when sanitizing the password field in the single server implementation. This is primarily due to the fact the raw value, the actual password, is never actually stored in the database, so it would be inadvisable to store the raw password as the token's repair value. The hardest place to deal with this was with basic access authentication. The entire Authorization header data needed to be replaced with a token, but this also meant that the username was also tokenized. Rather than being able to store the hash of the password as was done with form-inputted password values, it was necessary to store which user, if any user, was authenticated. In addition, because the original code flow had many ways in which to exit the authentication method, the additional code required to handle tokens is interspersed everywhere.

In the OAuth 2.0 implementation, since nearly all of the secret keys were already stored in the database, the SOLAR-introduced tokens often simply stored the original value that the token replaced. Any further secret keys simply needed one line of code on the model to be sanitized. Since the `BufferedLog` technically only stores the actual data that needs to be replaced, it is possible that it unnecessarily replaces data. That is to say, a sensitive model field might store the data "abcdef" which would cause the `BufferedLog` to believe all instances of "abcdef" were secret whereas in reality, this might not be true. However, because all of the keys, codes, and tokens used in OAuth 2.0 are randomized, this is extremely unlikely. It is also unlikely that any of the different tokens interfere with one another since each one is at least 10 characters long. In the extremely rare case that this did happen, it probably still would not matter as both would get tokenized and the value would still be stored in the database for later retrieval. This would be a larger problem if we were not able to store the original raw value in the database.

Chapter 6

Related Works

There exist several repair systems that log randomized data or discuss privacy concerns.

Warp, as implemented for PHP, addressed the issue of randomize data. [4]. However, the focus of the Django implementation was on being able to recover from distributed workloads across multiple servers and ignored the issue of randomization [13, 5].

Sprenkle et al. built a system that is capable of recording user sessions as well as replaying the recorded sessions as test cases that mimic a real-world environment, but they do not address the issue of sensitive data.

Unlike SOLAR which captures all data into and out of the Django application, SCARPE, built by Orso and Kennedy, can selectively capture data of an application's subsystem in a log and replay the events [14]. This is similar to how SOLAR deals with logging randomization. They address the privacy issue by discussing the possible removing of any subsystems with confidential data to prevent the logging of such data, but acknowledge that this is not always possible. Another suggestion they make is the possibility of replaying the events on the users' machine and then to collect the resulting, sanitized analysis at the end. This would be an interesting direction to explore. To do this, users would probably have to record data in addition to the server's logs.

Chapter 7

Discussion and Future Work

SOLAR builds upon Warp. Warp is capable of handling most re-execution situations including inter-server interactions. SOLAR presents two major improvements. First, it is capable of re-executing randomized sequences, enabling it to repair authentication and authorization requests, which Warp could not handle. SOLAR also sanitizes the repair logs of sensitive information such as passwords and provides a framework to allow future developers to integrate SOLAR into their applications, giving them the benefits of a repair system without added breach of security from storing sensitive data in the logs.

However, while SOLAR can help re-execution and also sanitizes the log, it is still possible that the sensitive data leaks out through other means perhaps due to administrative configuration error. As of right now, SOLAR can rollback offending requests, but it cannot undo damage done from unauthorized reads. SOLAR could be extended to help identify leaked data, and with tagged fields on forms and models, this extension could also rate how devastating the situation is in terms of whether or not the data leaked was sensitive. Another potential extension would be to audit how sensitive data is being used in an application and where it is being exposed to users and other applications.

The current implementation of SOLAR generates a new `Token` model object every time a model save or query happens on a model with a sensitive field whether in record mode or repair mode. Warp's logs and versioned database already grow in size over time and eventually need garbage collection. The rate of `Token` generation only adds to this problem. While garbage collection is probably necessarily no matter what optimizations are made, there are a couple of optimizations that can be made regarding decreasing the number of `Tokens` generate. For example, it should be possible to limit SOLAR to generating only one `Token` per sensitive field per request. It should also be possible to reuse the same `Token` for the corresponding repair entry to a record entry if the data in fact stays the same.

There can also be further improvements on token strings. One particular improvement would be to make it possible to compare two augmented token strings for equality. Rather than testing the entire token string, only the raw value should be compared. This type of operation occurs when validating user registration forms that have a password and a password confirmation fields.

Chapter 8

Conclusion

Many web applications store private information about its users. While many tools have been built to help maintain security of web applications, there are few tools that help recover from such vulnerabilities. Warp is a system that records enough data as the web application runs so that it can recover from an exploited vulnerability. However, it introduces new security weaknesses by logging sensitive data that should be kept secret. This thesis presented SOLAR, an improvement to Warp that addresses the challenge of maintaining a working repair system while also sanitizing the logs.

SOLAR accomplishes this by providing a logging mechanism for randomness, a tagging mechanism for sensitive data on forms and fields, and a tokenizing mechanism for those fields. A large focus of this thesis has been to demonstrate that it is possible to re-execute authentication and authorization code without logging sensitive details.

This thesis also included a discussion of SOLAR's limitations, improvements that could be made upon it, as well as other, related directions this could be taken such as auditing how private data is used in an application.

Overall, it is important to keep minimum exposure of sensitive data to the outside world, and SOLAR works to accomplish that goal in repair systems.

Bibliography

[1] OAuth community site. URL `http://oauth.net/`.

[2] OpenID foundation website. URL `http://openid.net/`.

[3] Welcome to tastypie! URL `http://django-tastypie.readthedocs.org/`.

[4] Ramesh Chandra, Taesoo Kim, Meelap Shah, Neha Narula, and Nickolai Zeldovich. Intrusion recovery for database-backed web applications. *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 101–114, October 2011.

[5] Ramesh Chandra, Taesoo Kim, and Nickolai Zeldovich. Asynchronous intrusion recovery for interconnected web services. *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, November 2013.

[6] National Vulnerability Database. CVE statistics, February 2014. URL `http://web.nvd.nist.gov/view/vuln/statistics`.

[7] Django Software Foundation. The web framework for perfectionists with deadlines. URL `https://www.djangoproject.com/`.

[8] ifttt, Inc. Put the internet to work for you. URL `https://ifttt.com/`.

[9] Google, Inc. Google apps script. URL `http://www.google.com/script/`.

[10] HiiDef, Inc. Django oauth 2.0 server app. URL `https://github.com/hiidef/oauth2app`.

[11] Yahoo, Inc. Pipes: Rewire the web. URL `http://pipes.yahoo.com/pipes/`.

[12] Zapier, Inc. Automate the web. URL `https://zapier.com/`.

[13] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich. Recovering from intrusions in distributed systems with dare. *Proceedings of the 3rd Asia-Pacific Workshop on Systems (APSYS)*, July 2012.

[14] Alessandro Orso and Bryan Kennedy. Selective capture and replay of program executions. *Workshop on Dynamic Analysis (WODA)*, May 2005.