Automating Isolation and Least Privilege in Web Services

Aaron Blankstein and Michael J. Freedman
Department of Computer Science
Princeton University
Princeton, USA
ablankst@cs.princeton.edu and mfreed@cs.princeton.edu

***Abstract*—In many client-facing applications, a vulnerability in any part can compromise the entire application. This paper describes the design and implementation of Passe, a system that protects a data store from unintended data leaks and unauthorized writes even in the face of application compromise. Passe automatically splits (previously shared-memory-space) applications into sandboxed processes. Passe limits communication between those components and the types of accesses each component can make to shared storage, such as a backend database. In order to limit components to their least privilege, Passe uses dynamic analysis on developer-supplied end-to-end test cases to learn data and control-flow relationships between database queries and previous query results, and it then strongly enforces those relationships.**

Our prototype of Passe acts as a drop-in replacement for the Django web framework. By running eleven unmodified, off-the-shelf applications in Passe, we demonstrate its ability to provide strong security guarantees—Passe correctly enforced 96% of the applications' policies—with little additional overhead. Additionally, in the web-specific setting of the prototype, we also mitigate the cross-component effects of cross-site scripting (XSS) attacks by combining browser HTML5 sandboxing techniques with our automatic component separation.

***Keywords*-security policy inference; isolation; capabilities; principle of least privilege; web security**

I. Introduction

Network services play a central role in users' online experiences. In doing so, these services often gather significant amounts of valuable, user-specific, and sometimes privacy-sensitive data. Unfortunately, despite the importance of this data, client-facing applications are susceptible to frequent, sometimes high-profile [1, 2], break-ins that ultimately compromise user data. Many times, a security failure in a single faulty part of the application exposes data from other parts. In other cases, aberrant application behavior can be exploited to release otherwise protected data [3].

Mitigating these threats is typically not easy. Conventionally, an entire application or network service runs with one privilege, and often in one shared-memory address space. This has problematic security implications, because attacks that manage to overcome one portion of the application may affect its entirety. Even if application components can be better isolated by running with limited privileges and in separate processes, attack channels commonly exist through communication channels or shared persistent storage, such as backend databases in web applications. In this manner, attackers can target the "weakest link" of an application (which may undergo less scrutiny by developers) and then escalate their control. For example, breaking into a website's public forums can lead to access to sensitive user data and passwords, either through explicit database queries or by directly accessing values in program memory. In some cases, attackers need not even compromise application code; unexpected application behavior can lead to execution paths which ultimately leak or compromise user data [3].

To deal with these threats, our work applies three design principles. First, we split portions of application code into isolated components along natural isolation boundaries, taking advantage of the typical "switched" design of networked applications. Second, in applying the principle of least privilege [4], we minimize the amount of privilege given to each component to only that privilege which the component needs to execute at that specific time. Finally, we use dynamic analysis to infer each component's required privilege, such that the principle of least privilege can be largely automated.

While the principle of least privilege and the goal of maximizing isolation between components are old concepts, we believe that today's network-facing services provide a unique opportunity to apply these concepts. Automatically partitioning traditional, single-process applications is notoriously difficult [5, 6]: they are typically designed with large amounts of shared memory and application traces can be long, with many user interactions intertwined in the execution trace. However, today's scale-out architectures and their client/server division-of-labor offer new possibilities. They encourage developers to write server-side applications with components that offer narrowly defined interfaces and handle short requests. While this often leads developers to design their applications to support isolation, these applications usually all run in a single privilege domain and address

space. We, however, leverage these properties to automatically decompose applications into isolatable components.

This paper presents Passe, a system which realizes these design principles in a typical client-facing network application and allows for the enforcement of learned security policies in a large-scale datacenter architecture. Passe runs developer supplied applications as a set of strongly isolated OS processes, though the design also supports running each component on a separate machine. The isolated components, or what we call *views*, are restricted to what data they can access or modify. Passe protects data by limiting views to particular data queries. For example, a view which only handles displaying user messages will be restricted to only making queries which fetch user messages. These queries are restricted further by applying integrity constraints to capture and enforce the data and control-flow relationships between queries and other data sources in the system. If the argument to the above query is always derived from the "current user", then a data-flow constraint would assert that "current user" is the only valid argument to the fetch message query. If a permissions check is always executed (and returns True) before the fetch, then a control-flow constraint would assert that a permission check must be executed and must return True prior to the fetch query.

To discover the constraints for these queries, Passe monitors normal application behavior during a trusted learning phase. During this phase, our analysis engine not only learns which views make which database queries, but it also infers data-flow and control-flow relationships between database query results and later database queries. In particular, Passe captures these relationships when the dependency is one of equality or set-membership. While more limited than general control-flow or data-flow dependencies, this approach captures relationships based on object identifiers, which are how applications typically express security policies (e.g., a particular message or post is associated with a particular set of allowable user IDs). Further, by restricting the set of relationships we enforce, Passe avoids a problem where most objects in the system are control-flow or data-flow dependent, even though they may only be "weakly" dependent (i.e., due to over-tainting). Ultimately, Passe's learning phase outputs an inferred policy. These policies are capable of capturing direct data-flow and control-flow dependencies between query results and subsequent queries. For example, an application may use two queries to implement an access control: the first query checks whether the current user is in the set of authorized users, and the second query only executes if the first query returns true. Passe would enforce this by requiring that the first query always return true before the second query could ever be issued.

Our analysis phase is related to work in Intrusion Detection Systems (IDS) [7, 8], which similarly analyze the behavior of applications to infer the "normal" behavior of the application. Unlike prior work in IDS, however, Passe translates these inferred relationships into integrity constraints which the runtime will later enforce. This translation from dependency relationships to integrity constraints is exactly what enables Passe to support rich data policies in a large-scale application architecture. Our analyzer may in some cases make inferences which are *too strong*, leading to some normal application functionality being denied. In this sense, Passe is a default-deny system: if particular queries have not been witnessed by the analyzer, then those queries will not be allowed. Developers can fix overly-strict constraints by either adding test cases to correct Passe's inferences or by modifying the policy constraints manually.

While it may be a source of developer frustration, we believe such behavior has additional security benefits. The history of web application break-ins shows that applications are too often written such that, even without a remote code execution exploit, attackers can make database reads or writes that are inappropriate given what the developer actually intended [3]. Because the application's testing phase forms the basis for policy generation under Passe, it can serve as a check for such intent and helps prevent aberrant program behavior leading to data policy violations. Interestingly, *code* analysis techniques like symbolic execution, in finding the program's exact constraints, would not provide such a feature.

We built a prototype of Passe on Django, a framework for building database-backed web applications in Python. Our prototype's analysis engine runs unmodified Django applications and infers a policy configuration for the applications. This configuration specifies (i) how to split Django application into separate views, each running in their own sandboxed OS process, (ii) how to limit access for each view's database queries, according to the principle of least privilege, and (iii) what integrity constraints to place on these queries. Our design was not specific to Django, and we expect the same mechanisms could be built into other popular frameworks.

We evaluated Passe's effectiveness and ease-of-use by analyzing and running 11 off-the-shelf Django applications. We found that Passe can both restrict the set of queries each component can make, and infer richer application security policies through data-flow dependency relationships. We also evaluate the performance of our prototype on a series of application tests, measuring an increase in median request latency of 5-15 ms over normal Django, mostly due to the cost of data serialization between Passe's isolated processes. While workloads served entirely from memory suffer a 37% drop in throughput, workloads requiring database interactions, as is typical for web applications, experienced a throughput reduction of about 25%.

II. Security Goals and Assumptions

The goal of Passe's analysis is to infer the strongest possible query constraints which may be applied to isolated views. There are several potential problems with this. First, if

an application is not easily decomposed, then Passe will fail to separate it into views, or if single views are responsible for large portions of the application's behavior, the provided isolation will not be particularly useful. Second, if database queries are highly dynamic, Passe may not allow the queries at all. If queries are not protectable through simple data-flows (as one might expect in very complex applications), then Passe will not provide protections. We developed our prototype of Passe to explore how well these goals can be achieved with common web applications.

A. Threat Model

Passe assumes that application developers supply non-malicious, although possibly exploitable, application code to our framework, which runs on one or more application servers. This possibly faulty code may also include many third-party libraries. Thus, we do not trust that applications or their included libraries are secure. An attacker can exploit bugs in application views with the goal of compromising other views or shared data. Additionally, in the web setting, an attacker can return scripts to a client's browser which attempt to access or extract information from other views (this includes traditional cross-site scripting attacks).

We do, however, assume that attackers are unable to compromise the trusted components of Passe. Further, we trust the underlying data store, the OS running our components, and, for web browser sandboxing, we trust that browsers correctly enforce the new sandboxing features in HTML5. While these components may and do have bugs, they can be patched and updated. As a common platform shared by many websites, we believe there are greater incentives and opportunities to secure these components, as opposed to securing each application. Similar to the operating system, securing the framework only has to be done "once" to benefit all its users. Further, Passe's trusted components provide functionality which is much simpler than the actual application code.

B. Motivating Classes of Vulnerabilities

There are three classes of vulnerabilities common to networked applications—and web applications in particular—that Passe is able to mitigate. We further discuss how Passe mitigates these vulnerabilities in §VII.

1) **Poorly understood application behavior.** Even while using frameworks which prevent vulnerabilities in data operations (such as SQL queries), application developers may use library calls which have surprising behavior in unexpected settings. For example, the 2012 Github / Ruby-on-Rails vulnerability was caused by the default behavior of the Rails mass assignment operation. This operation allows web requests to set arbitrary attributes of an `UPDATE` query.
2) **Cross-Site Scripting (XSS).** A client's web browser presents a possible channel to attack Passe's isolation of views. Traditional XSS attacks may allow a vulnerability in one view to make AJAX requests to other views. For example, user input on a forum page is not properly sanitized, allowing one user to upload Javascript which, when executed by another user, has malicious effects such as changing the second user's password or accessing their sensitive data. Additionally, a compromised server-side view could use XSS as a side-channel to defeat the server-side isolations of Passe. While numerous approaches exist to filter user inputs, discover vulnerabilities, and help translate applications to use proposed W3C Content Security Policies features, these techniques either cannot find all XSS vulnerabilities, or they require programmer effort to modify Javascript code. Passe is able to mitigate many of the effects of XSS attacks using the same isolation model that it applies to application views.
3) **Arbitrary Code Execution.** Even when applications are programmed in high-level languages such as Python, there are occasional vulnerabilities allowing attackers to execute arbitrary code. While these problems may be infrequent, they are particularly damaging. For example, a vulnerability in the Python YAML library enabled attackers to gain complete control of an attacked Django runtime [9].

C. Security Properties of Passe

In the event that an attack against a particular view succeeds, Passe continues to provide the following security properties:

P1: Isolation of Execution. An attacker is unable to inspect or alter the memory of other application views. This provides security for the execution of other views. In the context of web applications, this applies to cross-site AJAX requests: only application views which normally communicate using AJAX are allowed to communicate during the secure execution mode.

P2: Isolation of Data. An attacker is unable to read or modify portions of the durable data store that are not accessed by the compromised view during its normal (i.e., correct) execution. For example, if the application is an online store with an attached discussion forum, and an attacker compromises only the forum view, he would still be unable to read or modify data associated only with the store's functionality.

P3: Enforcement of Data Policy. An attacker is unable to violate high-level application data policies, even when the data concerned is normally accessible by the compromised view. For example, a correctly-behaving view may only fetch messages for end users that are "logged in" to the service, but because different users are logged in at different times, the view has high-level access to the entire set of messages. Even so, Passe ensures the finer-grain application security policy: even once comprising the view, an attacker cannot read

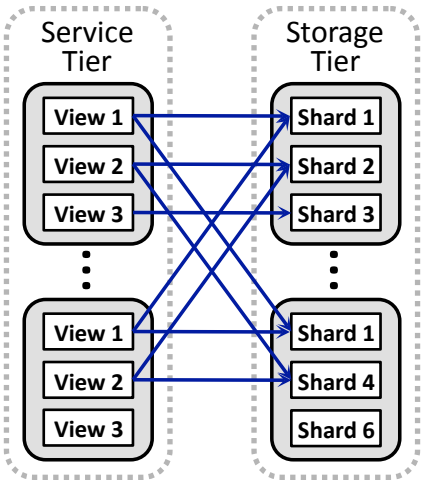

Figure 1: A typical tiered service architecture.

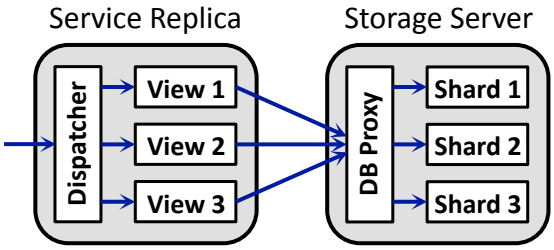

Figure 2: High-level overview of the Passe runtime.

messages of users that are not logged in. More specifically, Passe preserves data-flow dependencies on database query arguments and control-flow dependencies between queries. If, during a normal execution, a particular database query argument is always the result of a prior database query or the user's authenticated credentials, then even in the case of a compromised view, that argument must still come from that source.

III. Passe Design

The design of Passe's runtime accommodates the typical tiered, scale-out architecture of most client-facing datacenter services, illustrated in Figure 1. In this architecture, a request is forwarded to an appropriate machine in the service tier. The service tier (also called the application tier) is comprised of multiple machines, possibly running the same application code, which access shared storage through the storage tier. The storage tier is comprised of possibly many machines, handling separate, potentially replicated partitions (or "shards") of the shared storage.

In Passe's runtime (Fig. 2), applications are decomposed into isolated views, running in separate sandboxed environments. This can be achieved through OS-level mechanisms, such as AppArmor [10], or by running views on entirely separate machines. Each of these views is responsible for handling specific requests which a dispatcher will forward. Passe introduces a stateless proxy between the service and storage tiers which interposes on data queries. This trusted proxy approves or denies data queries based on a set of constraints. These constraints are applied to the supplied queries and a supplied token, ensuring that application data policies remain in effect even during a compromise of the application.

Passe provides an analysis system which monitors the "normal" execution of applications, and during this, learns the data-flow and control-flow dependencies of the application. This learning process occurs during an explicit testing or closed deployment phase, during which we assume components are not compromised.

A. Interacting with a Shared Data Store

Passe provides data isolation between application views. If two views never share data through the shared data store, then the compromise of one view should not affect the data of the other. While secure, strict isolation greatly limits the type of applications one can build—applications frequently need to share data to provide basic functionality. For example, sensitive information in a database's user table may be shared between nearly all of an application's components.

In Passe, we allow application views to interact with a shared data store through a query interface. Conceptually, an *unbound query* (as we will see later, in SQL these are query strings) has a set of arguments. Normally, when an application issues a query, it supplies an unbound query and a tuple of argument values. For example:

```
result = fetchUserMessage(uname = "Bob")
```

In order to enforce data policy, Passe must constrain the arguments to queries. However, these arguments are not necessarily hard-coded constants and may instead derive from prior database results. For example, a view displaying all of a user's friends' updates might issue two queries:

```
friends = fetchFriendsOf(uname = "Bob")
updates = fetchUpdates(author in friends)
```

Here, data from the first query is used as an argument value in the second query. Passe will attempt to enforce this relationship: that the second query should *only* contain arguments from the first query. In fact, this example demonstrates a data-flow dependency. In a data-flow dependency, the argument value of one query is equal to the result of a previous query. Another type of dependency is the control-flow dependency. In this case, the result of one query affects whether or not a second query would even be issued. Passe captures data-flow and control-flow dependencies which can be expressed with equality relationships. Figure 3 shows example application code demonstrating dependencies.

B. Protecting the Shared Data Store

Passe employs two mechanisms to enforce dependencies: a **database proxy** and **cryptographic tokens**. Every response from the proxy includes a token. This token is a set of

```
# T0 is an initial token
{R1, T1} = getUID({"Alice"},T0)
# Data-flow Dependency
{R2, T2} = isAuthed({R1}, T1)
# Control-flow Dependency:
if R2 = TRUE:
  {R3, T3} = getData0({}, T2)

{R4, T4} = getACL({}, T3)
# Control-flow Dependency:
if R1 in R4:
  {R5, T5} = getData1({}, T4)
```

Query	Constraints
getUID	Unconstrained
isAuthed	Data-Flow: (Argument == R1)
getData0	Control-Flow: (R2 = TRUE)
getACL	Unconstrained
getData1	Control-Flow: (R1 in R4)

Figure 3: Example queries demonstrating the types of data-flow and control-flow dependencies. Queries take an argument set and a token as input.

key-value pairs which encode results from previous queries. Every request to the proxy must include a token, which the proxy will use to check that particular data dependency relationships are met (and that application code is not trying to issue unauthorized queries). This token allows the database proxy to track what information has been returned to the view while remaining stateless itself (particularly important if the system employs multiple such proxies). In order to prevent compromised code from altering this token, it is cryptographically MAC'ed by the proxy. The key used for this operation is shared by Passe's dispatcher and proxy. To prevent replay attacks with this token, the dispatcher and proxy include nonces in each token and track the nonces which have already been used.

In order to approve a query, the database proxy consults an access table and applies a two-stage verification process. In the first stage, the proxy checks whether the requested unbound query is whitelisted for that particular view. If not, then the request is denied (in this sense, the proxy is *fail-safe*). In the second stage, the proxy checks if the set of constraints associated with the unbound query is satisfied by the supplied argument values and token. Figure 3 displays a set of constraints for the associated unbound query.

C. Learning Constraints

Passe infers access-table entries during a learning phase. In this phase, Passe uses dynamic taint tracking to learn the data and control-flow dependencies experienced during the normal execution of application code. The developer can either supply test cases or run an "internal beta". Once this phase has completed, Passe translates dependencies into the token-based constraints which will form the access table entries. These inferences would allow any of the witnessed traces to run, but then errs on the side of strictness. If the analyzer provides too strict of a configuration, the developer can either increase the number of test cases or alter the configuration manually.

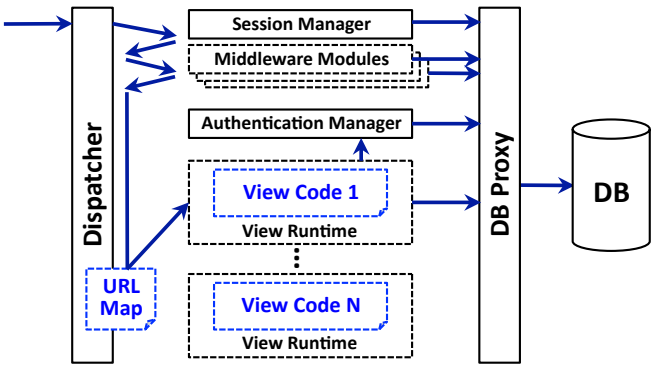

Figure 4: Each arrow represents a communication channel in Passe's runtime. Solid lines are trusted; dashed are untrusted. Blue lines correspond to developer-supplied code, which is embedded into Passe's runtime components. When a request arrives at the dispatcher, it first gets processed by any number of middleware modules (which includes the session manager), before getting matched against the URL Map for dispatching to the appropriate view.

IV. Passe Runtime in Web Setting

We implemented Passe as a drop-in replacement for Django, a popular Python-based web framework which relies on the "model-view-controller" design pattern. This pattern defines a logical separation between various computation and data components. In this setting, the Passe runtime (shown in Figure 4) involves the following components:

- The **Dispatcher** uses the URL of a request to decide which view will handle a particular request.
- The **Session Manager** handles mapping user cookies to stored sessions.
- The **Authentication Manager** checks users credentials (username and password) and associates the current session with that user.
- The **Database Proxy** mediates access to a SQL database.
- The **View Server** provides a wrapper around the view function for handling inter-process communication and automatically binding tokens to requests in and out of the view. The view function itself is the unmodified developer code.

A. Isolating Views

In automatically decomposing applications into isolated views, we must solve four problems, related to (i) determining view boundaries, (ii) translating function calls into inter-process communication, (iii) dealing with global variables, and (iv) sandboxing these processes.

Passe determines application boundaries by leveraging the design of Django. In Django, application developers specify a mapping of requests to functions which handle the logic and HTML rendering of those particular requests. In Passe,

we treat each of these functions as a separate view, such that each view is responsible for handling a complete request.

Passe must translate what were previously simple function calls into inter-process communication. Passe wraps application code with a view server, which handles marshalling function calls into this inter-process communication. This uses the Pyro library for Python remote objects, which automatically serializes the arguments of remote procedure calls using Python's `pickle` module. The deserialization process is unsafe: if any object can be deserialized, then arbitrary code may be executed. This is dealt with by modifying the deserialization code to only instantiate objects of a white-listed set of types.

Because application code now runs in separate processes, previously shared global variables are no longer shared. However, in order to support a scalable application tier, developers are encouraged to share global variables through a narrow interface by modifying values in a single request state object. In Passe, changes to this object are propagated back to the dispatcher by our view server. In order to minimize this overhead, Passe computes and sends a change-set for this object. The dispatcher checks that the change-set is valid (e.g., a view is not attempting to change the current user) and applies it to a global request object.

Passe sandboxes these views by engaging Linux's AppArmor and creating specific communication channels for the processes. Each of the views communicates over specific Unix domain sockets with the dispatcher and the database proxy. As each view server starts, an AppArmor policy (which Passe defines) is engaged, and the view server becomes sandboxed. This prevents views from making system calls, communicating over the network, or reading from the filesystem. Views may only read from a limited set of files required for their execution. This set of files includes the Python libraries and the application source code, allowing the view to read and execute those files. When executed, these files run within the view's sandbox. Network access is limited to the Unix sockets used to communicate with Passe's components.

B. Constraining SQL Queries

Applying Passe constraints to SQL queries requires two mechanisms. First, we need to specify how a SQL query maps to our notion of an unbound query. Second, we need to specify how SQL query results are stored and referred to in the token.

In Django, applications issue queries as a query string and an ordered list of arguments. For example, a view might supply a query string

```
SELECT (text,fromUser) FROM msgs WHERE toUser = ?
```

and an argument "Bob". For Passe, we treat the query string itself as the unbound query. In the access table, we store the strings with a cryptographic hash of the query string (to reduce the required storage).

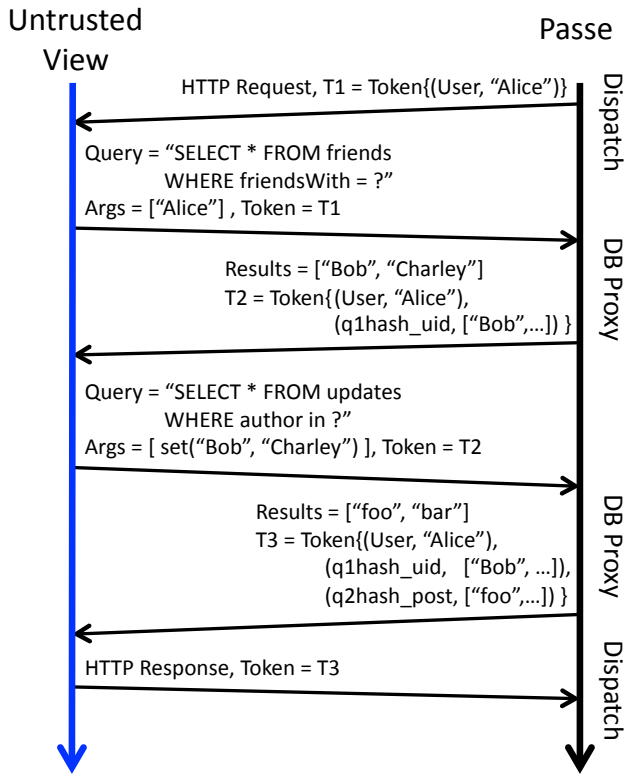

Figure 5: At the start of a request, Passe provides a view with an initial token containing the current user. As the view makes database queries, it supplies the current token which Passe's proxy uses to check the integrity constraints on the arguments supplied to the query. The database proxy replies with updated versions of the token based on the query results.

In order to store query results in the token, we again use a hash of the query string to refer to the "source" of the data. In addition, we separate the results by *column* so that Passe can place constraints at the column granularity, rather than the entire result set.

C. Handling a Web Request

When a view receives a request, the dispatcher gives it an initial token containing the current user (or "anonymous" if appropriate) and any HTTP request variables (query strings and header fields).

The contents of Passe's tokens during the execution of a view are shown in Figure 5. As the view makes requests into trusted Passe components, those components respond with updated token values, which may in turn be used in future requests. Whenever updating a token, the trusted component generates a new message authentication code (MAC) covering the token's latest contents.

This example demonstrates how Passe's constraints operate in practice. When operating correctly, this view displays all posts from the current user's friends, in this case Alice's friends. From a security perspective, the view should only be able to enumerate the user's friends and only read those updates from her friends (confidentiality properties). In more detail, the view initially receives the HTTP request object

and a token containing the current user. The view makes two database queries, each with an argument protected by a data-flow constraint. The first query derives its argument from the "User" key. The second query's argument, however, is derived from the results of the first (and matches a key that names the first query and the "uid" column). These constraints ultimately enforce the application's desired policies, even if the view were compromised: the view can only see updates for users contained in the result set of the first query and that query can only return the current user's friends.

D. User Authentication and Session Management

Passe's constraints specify a relationship between a query and the trusted sources that supply the values to that query (or affect the control-flow leading to that query). Certain token keys—such as the current user and HTTP request parameters—do not originate from prior queries, however, but rather serve as a foundation for the constraints of a view's queries. It is vital that the mechanisms used to generate these tokens are sound: If an adversary can forge the current user, confidentiality is largely lost.

Traditionally, Django has two mechanisms for associating a request with a user. Either a view can explicitly call into the authentication library which returns the associated user, or the request is part of a session already associated with a logged-in user. In the latter case, before a view handles a request, the dispatcher calls into the session manager, which reads the request's session cookie and checks whether it is already associated with a user.

In Passe, we modified these two mechanisms so that both session and authentication manager will securely embed the current user's ID in a token, rather than simply returning the user to the dispatcher or login view, respectively. This change also entails that these managers know the shared symmetric key used to MAC tokens.[1] To prevent a compromised view from stealing session cookies, the Passe dispatcher elides session information from the token before forwarding the request to the view.

E. Isolating Views at a Client's Browser

An end user's browser presents a significant attack channel for an attacker with control of a view. The attacker can return an HTML page with malicious code used to circumvent least-privilege restrictions and thus access other portions of the web application. For example, if an attacker compromises a view A which cannot access restricted portions of the database, the attacker can return Javascript which loads and scripts control over another view B. View A can then use the results of view B to gain access to otherwise inaccessible portions

[1]In fact, our implementation could have left the session manager unmodified, as it only communicates with the dispatcher, which could have embedded the user ID on its behalf. Because the authentication manager is accessed by an untrusted login view, however, it must implement this secure token embedding itself.

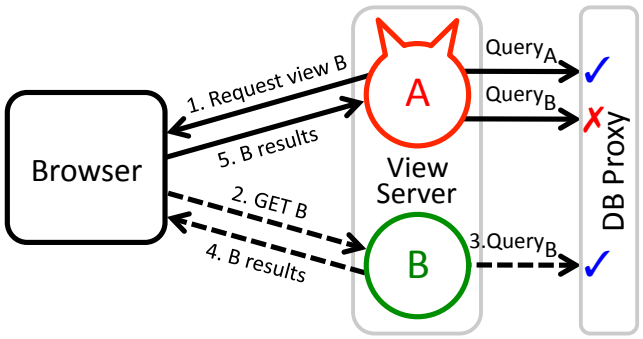

Figure 6: An attacker who has compromised view A is unable to directly query the database for view B's data. However, by returning a script to the user's browser, the attacker can exfiltrate B's data by having the browser make seemingly normal requests to view B.

of the database, as shown in Figure 6. This attack is similar to XSS in that the same-origin policy fails to protect other portions of the web application from the malicious script. Typically, applications prevent XSS attacks by filtering user inputs. However, an attacker with control of a view can circumvent these protections by inserting Javascript directly into a response. Even when an application uses a feature such as Content Security Policies (CSP), entire domains are typically trusted to supply scripts [11].

To mitigate this cross-view attack channel, Passe supports execution with isolation *even at the client browser*. In particular, to preserve isolation between views, Passe's dispatcher interposes on AJAX requests between views. The dispatcher keeps a mapping, learned during the Passe training phase, of which views are allowed to originate scripted requests to other views. Based on this mapping, the dispatcher approves or rejects requests. This requires the dispatcher to know which view originated a particular request and whether that request is an AJAX request. The dispatcher derives this information from two HTTP headers: `Referer` and `X-Requested-With`.

To prevent adversaries from circumventing these checks, Passe must ensure that an attacker cannot remove or modify these headers. Towards this end, Passe sandboxes a view's HTTP responses using the new HTML5 `sandbox` attribute with `iframes`. Every view's response is wrapped in this sandboxed environment by the dispatcher. We implemented a trusted shim layer which ensures that the headers are correctly added to each outgoing AJAX request. Our approach is similar to the shim layer used to enforce privilege separation in HTML5 applications as introduced by Akhawe et al. [12].

F. Applicability to Other Frameworks

While much of Passe's prototype implementation is concerned with Django-specific modifications, Passe's architecture is directly applicable to other web frameworks as well. For example, in Ruby on Rails, the dispatcher would make routing decisions based on Rails' *routes* and views would be separated along Rails' *ActionControllers* and *Views*.

However, because some frameworks do not include standard authentication libraries, Passe would need to provide a new third party authentication library, or support some of the most common ones.

V. The Passe Analysis Phase

During the analysis phase, Passe monitors application execution with the following goals:

- **Enumerate Views.** Passe's runtime only executes a fixed set of views. The analysis phase is responsible for enumerating these views and assigning each of them a unique identifier.
- **Enumerate Queries.** Passe associates each view with the SQL queries witnessed during analysis.
- **Infer Dependency Relationships between Queries.** The analysis phase is responsible for determining data and control-flow relationships between queries, prior query results, and other data sources.
- **Translate Dependencies into Enforceable Constraints.** Dependencies witnessed during the learning phase must be translated into constraints which the Passe runtime is capable of enforcing.

Passe's analysis phase achieves these goals using a combination of taint tracking and tracing. As queries execute, Passe constructs an event log and adds taint values to the database results, and once execution has completed, Passe processes this log and outputs the allowed queries for each view, and the associated constraints on those queries. The analysis phase runs on a PyPy Python interpreter which we modified to support dynamic taint tracking.

A. Dynamic Taint Tracking in Python

In order to support dynamic taint tracking for Passe, we developed a modified version of the PyPy Python interpreter (our modifications are similar to approaches found in [13, 14, 15]). Our modifications allow for fine-grained taint tracking through data passing operations and some control-flow tracking. The interpreter exposes a library which application-level code can use to add taint to a particular object, check an object's current taint, and check any tainting of the current control-flow.

Each interpreter-level object is extended with a set of integer taints. As the interpreter processes Python bytecode instructions, any instruction which returns an object propagates taint from the arguments to that object. Additionally, because many functions in PyPy are implemented at the interpreter level (and therefore are not evaluated by the bytecode interpreter), these function definitions also need to be modified to propagate taint. For our prototype implementation, only functions for strings, integers, booleans, unicode strings, lists, dictionaries, and tuple types were modified.

In order to track control-flow tainting, the interpreter checks the taint of any boolean used to evaluate a conditional jump. If the boolean contains a taint, this taint is added to the current execution frame. Taints are removed when their originating function returns. In our prototype, the current control-flow taint does not propagate during data operations—if a control-flow taint is active while a data-flow operation occurs, the result is not tainted with the control-flow taint. While this causes the analysis to miss some control-flow dependencies, this is purely a limitation of the prototype, and the applications we tested were not affected. While including this feature will increase the possibility of over-tainting, Passe's constraints only capture equality and set-membership relationships which mitigates many of the effects of over-tainting.

B. Tainting Data Objects and Logging Query Events

During the analysis phase, Passe creates a log of query events by tracing the normal Django execution of the application. In order to track the data-flow and control-flow dependencies in the application, these events contain taint tags for each logged data object.

In addition to capturing query calls, Passe must properly add taints to data objects as they flow through the application. As HTTP requests enter the application, Passe taints the initial data objects. Later, as each database query is made, Passe also adds taints to each result column.

When Passe captures a query call, it logs the event with the following information:

1) The view responsible for the query.
2) The query string.
3) An ordered list of the query's argument values and the current set of taints for each of those values.
4) Any previous database results or initial data objects, and these objects' associated taints.
5) The control-flow taint set for the current execution context. In addition to a set of taint tags, for each permissions library call which affects the control-flow, the name of the checked permission is included. The permissions library is special-cased because of the "root" permission set. (Existence of a particular permission may be checked, or if the user is root, then the check is always approved.)

This information will allow the analyzer to translate witnessed dependency relationships between queries and objects into the integrity constraints used by the Passe runtime. Dependency relationships here are captured by the included taint sets.

C. Inferring Constraints from Dependency Relationships

Knowing that specific database query calls depend on previous query results is not sufficient for the Passe runtime to enforce constraints. Rather, Passe collects the logged query events and uses these to infer enforceable constraints. To do this, Passe collects all of the events for a particular *(query string, view)* pair and merges the dependency relationships.

The analyzer constructs constraints which are the union of all witnessed data-flow relationships: if, across all witnessed query events, multiple data-flow relationships exist for a particular query argument, then *any* of those sources will be allowed to satisfy that argument's constraint. On the other hand, if in some query events, no data-flow relationships exist at all, then the argument will be left unconstrained. To capture data-flow relationships, the analyzer only checks for *equality* and *set membership* relationships. These two relationships capture relationships based on object identifiers, which are typically how applications express security policies. (For example, a particular message or post is associated with a particular set of allowable user IDs.) As we will see in our example applications (§VII), no policies were missed because of this limitation. By requiring equality, Passe mitigates many of the problems normally associated with over-tainting. Normally, if "too much" taint propagates in an application, constraints based solely on taints will be too strong. In Passe, however, both the taints and the values are required to be equal, which reduces the chance of over constraining a particular query. In cases where this does happen, then the developer can include test cases where equality does not hold which will prevent that constraint from being inferred.

Passe captures control-flow relationships similarly. For each query event, Passe determines which control-flow relationships affected that event. Passe then creates a set of invariants for the query based on these relationships. Here, unlike in data-flow constraints, there is a higher chance of over-fitting the invariant. For example, the `getUID` query in Fig. 3 affects the control-flow of `getData1`. Passe could infer an invariant containing an `OR`ed set of user IDs. This invariant, however, is too strong in practice: it fits the invariants to precisely those witnessed during the testing phase. Thus, rather than unioning these sets of possible invariants, Passe takes the intersection of these sets to construct the final invariant. For this example, the invariant `R1 in R4` would be witnessed for all `getData1` events, while invariants such as `R1 = Alice's UID` would only be witnessed for a few events.

When this translation phase ends, each view will be associated with a set of allowable queries. Each of these queries will have an associated set of control-flow and data-flow constraints, which are exactly what the Passe runtime uses in its access table to enforce query integrity.

D. Monotonicity of Analysis

By design, the Passe analysis phase does not witness all possible code paths and, therefore, Passe's inferences may prevent certain code paths from executing correctly. However, developers can increase the number of test cases witnessed by the Passe analyzer and increase the allowable code paths. In respect to this, Passe guarantees monotonicity: additional tests cannot *reduce* the set of allowable queries. To see why this is the case, imagine that Passe witnesses an additional query event for some *(query, view)* pair. If this event creates any new data-flow constraints, they only increase the allowable data-flows. If, previously, that particular query argument was unconstrained, then it will remain unconstrained (again, because data-flow constraints only add new paths). The same is true for any new control-flow constraints, because a control-flow constraint will only be added if it holds for all the witnessed events of a particular *(query, view)* pair.

E. Impacts of False Positives and Negatives

Passe's analysis phase is capable of both false positives and false negatives when detecting dependencies. False positives occur when the application developer wishes to allow data-flows which Passe's analysis phase does not witness. This results in a policy which disallows those data-flows. The developer can resolve such false positives by including new test cases which cover those particular data-flows. Passe is also capable of false negatives when detecting dependencies. In these scenarios, Passe will generate a policy which is too permissive, such that, in the event of an application compromise, an attacker would be able to execute data-flows which should be constrained. This can only occur if a witnessed dependency is not captured by our taint tracker. As discussed in §V-A, our prototype can fail to detect certain kinds of control-flow dependencies. (Building a more full-featured PyPy taint-tracker is part of ongoing work.) A developer can remedy such missed dependencies by manually inserting those dependencies into the outputted policy. While this is an unsatisfying method, we did not encounter any such cases of false negatives in our tested applications. This is also a current limitation of our prototype, rather than Passe's underlying method; a more complete implementation of taint-tracking in the Python interpreter would not encounter false negatives while detecting dependencies.

VI. Implementation

We implemented the Passe runtime and analysis engine as modified versions of Django 1.3. For the analysis engine, we modified certain Django libraries to make analysis easier—in particular, the authentication and database libraries—by adding annotating calls for the tracer. Further, we use our modified version of the PyPy 1.9 interpreter to perform our dynamic taint tracking.

For the runtime, we modified the Django dispatcher to support interprocess calls to views, and the database, authentication, and session libraries were modified to make proxied requests. A Gunicorn HTTP server just acts as the network front-end, accepting HTTP requests before passing them to Django (and its dispatcher). Our database proxy provides support for two different database backends, PostgreSQL and SQLite.

In total, Passe required 2100 new lines of code (LOC) for Passe-specific functionality, as well as changing 2500 LOC in the Django library and 1000 LOC in PyPy. Our HTML5

	views	queries	constrained queries	constraints	policies	app actions	admin actions	loc	code coverage
social-news	47	411	68%	653	17	155	46	4375	90%
swingtime calendar	19	95	52%	379	9	17	101	1187	97%
simplewiki	19	132	71%	370	13	46	69	1057	95%
django-articles	22	219	75%	455	15	12	139	992	100%
django-forum	18	165	81%	436	15	34	111	510	100%
whoami blog	15	47	49%	129	9	6	74	487	95%
wakawaka wiki	20	114	71%	323	7	39	92	471	99%
django-profiles	9	23	47%	69	5	18	37	230	100%
portfolio	10	24	61%	103	6	8	19	118	96%
django-polls-tutorial	5	29	62%	87	5	15	25	77	99%
django-notes	5	10	37%	16	4	16	0	65	100%

Figure 7: For each application, we measure the number of browser actions performed in our test suites for Passe's analysis phase and the number of discovered views, queries, constraints, and higher-level policies. Code coverage numbers reflect what percentage of the application code was covered by our tests (both in runtime and analysis phases). The lines of code (loc) not covered were either completely unreachable or unreachable through web browsing alone.

sandbox requires 320 lines of Javascript code, which are inserted into responses automatically by our dispatcher.

A. *Unsupported Django Behavior.*

While we attempt to provide an interface identical to Django, our modifications do require some changes to this interface: views are forced to authenticate users through the default authentication library, which we modified, applications cannot use arbitrary global variables and the URL Map may only contain views found during analysis.

Developers may attempt to authenticate users directly, circumventing the authentication library. In our system, this will fail, as only the authentication server is able to create a new token for the application. This is problematic for applications that use external sources for authentication (e.g., OAuth). Our prototype could be extended to support different authentication libraries, or to provide a generic API which would allow Passe to be integrated with custom authentication method. This, however, would still require modifying some applications to use our API.

Because views in Passe run in separate processes, global variables cannot be used to pass information between views. However, passing information through global variables is discouraged by the Django framework. Using global variables in this way can lead to problems even in normal Django deployments where multiple worker processes are responsible for processing concurrent requests in parallel. Because these workers do not share an address space, propagating information through global variables could lead to inconsistencies between requests. As such, none of the applications we tested used global variables in this way.

Django allows developers to modify the URL Map to add new views dynamically. While Passe could possibly be extended to support such behavior by giving the new view the same permissions as the parent view, this was not implemented. Instead, if a view attempts to modify the URL Map, it fails, as it has no access to the dispatcher's URL Map object. We did not encounter this problem in any of the applications we tested.

VII. Protections for Real Applications

To understand how Passe executes real applications, we ran and tested ten open-source Django applications, manually inspecting both the application source code and Passe's inferred policies. We assessed the source code for instances of application security policies and, in particular, those impacting data integrity or confidentiality. We then checked whether Passe correctly enforces those policies. Across all applications, we found four instances of clearly intended policies missed by Passe, exposing the application to confidentiality violations. In this section, we evaluate the following questions:

§VII-A. How does Passe mitigate our three classes of vulnerabilities?

§VII-B. How difficult is it to construct end-to-end test cases for Passe to analyze applications?

§VII-C. What coarse-grained isolation properties can Passe provide to applications?

§VII-D. Are there security policies in these applications which Passe's dependency constraints cannot enforce?

§VII-E. Examining three case studies in more depth, how does Passe capture fine-grained security policies?

§VII-F. How do common web application vulnerabilities apply to Passe?

A. *Passe in the Presence of Vulnerabilities*

1) Unexpected Behavior: When applications exhibit unexpected behavior, Passe is able to prevent the attacker from compromising the database in many cases. For example, in the 2012 Github / Ruby-on-Rails attack, a mass assignment

vulnerability allowed users to set the UID for the public key being uploaded. This allows users to upload a key for *any* user. In Passe, the normal path of the code would create a constraint for the `UPDATE` statements asserting that the UID must be equal to the current account.

2) XSS Attacks: Passe can mitigate many of the effects of XSS vulnerabilities. Passe restricts the content returned by views to only making AJAX requests to a whitelisted set of other views. For example, if a view containing user-generated content (such as a forum page) does not normally call to other views, then no XSS vulnerabilities in the view will be able to call other views. While this does not prevent scripting attacks which do not make cross-site requests, it does prevent views from using scripts to open attack channels against unrelated views. This allows Passe to preserve isolation between views, even at the client's browser.

3) Arbitrary Code Execution: Some web applications contain bugs which allow arbitrary code execution. For example, a commonly used YAML library in Django allowed arbitrary object deserialization, ultimately allowing remote code exploits [9]. YAML parsing libraries exposed node.js and Rails applications to a similar attack [16, 17]. Passe mitigates the threat of this by restricting developer-supplied code to specific database actions.

Unfortunately, an attacker who has complete control over a view can launch a phishing attack, displaying a convincing login screen to users. This is more damaging than normal phishing attacks as this page will be served by the correct web host. Therefore, it is still important to recover from an attack quickly, even though Passe continues to provide data security properties during an attack. Other similar systems are also susceptible to this attack, including those incorporating more formal mechanisms such as DIFC [18].

B. Building End-to-End Tests

To perform end-to-end tests, we wrote a test suite for each application using Selenium, a library for automating web-browser actions. Our suites tested applications entirely through their web interfaces by performing normal application actions. After running an application through Passe's testing phase, we ran the application in the secure runtime of Passe, and when the inferred constraints were too strong, we added more tests to the test suite. Each of these test suites took less than an hour to construct and was comprised of about 200 lines of Python.

To understand how much testing was required for each application, we measured the number of browser actions in each of the test suites we developed. The table in Figure 7 displays these measurements. An important note is that while each application required a large number of browser actions to test the application, many of these actions were performed on the `django-admin` interface. Because this is a standard interface, a more advanced testing framework could automatically generate tests for this interface, a possible direction for future work.

In order to run with Javascript sandboxing on the browser, Passe requires that a mapping from AJAX requesting views to the responding views be constructed. To see how much additional burden was required to generate this mapping, we tested the most AJAX-intensive application in our test cases (`social-news`) in the Javascript sandboxing mode. We modified our end-to-end test scripts so that elements would be selected in the sandboxed frame rather than in the parent window. Other than these changes, the original end-to-end tests we developed to capture query constraints were sufficient to capture all of the allowable AJAX requests as well.

C. Isolation Properties

To understand how much isolation Passe provides by restricting each view to the set of queries it needs, we measured the proportion of views that can access each table of an application. Half of our applications' tables are readable by at most 7 views. Still, some tables can be accessed by nearly all of an application's views. For example, in the blogging applications, the user table holds an author's username. Because most of the views display authored posts, the user table can be read by most views. When we look at views with write access, however, the separation is much stronger. Fully half of the tables for all applications are writable by only one or two views. Of course, these results do not speak to the guarantees provided by the inferred constraints, which further strengthen the application's security properties.

We measured the number of constraints Passe inferred for each application (Fig. 7). Additionally, we characterized higher-level policies by inspecting constraints by hand, discovering that Passe successfully finds 96% of the 105 possible application policies. Because we characterized these policies by hand, we cannot eliminate the possibility that we incorrectly characterized or left out policies.

D. Fetch-before-Check Problems

It is important to understand the scenarios in which Passe can miss an application policy. In all of the applications tested, Passe missed four application policies (out of 105 total policies): one in each of the `simplewiki`, `django-articles`, `django-profiles`, and the `django-forum` applications. In all four cases, the application code fetched a data object *before* evaluating a permission check. The `simplewiki` application, for example, performed read permission checks only *after* fetching wiki pages from the database. While this behavior poses no confidentiality problem if the application is never compromised, it is clearly not enforceable by Passe. This breaks Passe's assumption that the normal behavior of the system does not leak information. (Still, such an implementation can be dangerous from a security perspective:

Even when not compromised, the application fetches data even when permission is denied, which may expose the database to a denial-of-service attack if the fetched content may be sizable.)

This behavior can be quickly remedied, however. We fixed Passe's missed inferences by moving the permission checks to precede the data queries. In each of these applications, these changes required modifying fewer than 20 lines of code.

E. Case Studies

Social News Platform: `social-news` is a Django application which provides the functionality of a social news service like Hacker News or Reddit. Users can submit stories to one of many user-created topics and each of these stories supports a comment section. Users vote an item up or down, and the final count of votes determines an item's score. A recommendation system then uses these votes to suggest other stories which a user might like.

The `social-news` application contains three queries which need to be modified for the application to run with Passe. The application constructs these queries using string replacement to insert arguments rather than using SQL argument passing. This is known to be a less safe programming practice, as it can expose applications to SQL injection attacks. However, in Passe, such bugs cause an application to fail *safely*, as Passe will simply deny the constructed queries which it does not recognize. In order for the application to run correctly, 5 lines of code were modified so that query arguments were passed correctly.

With these changes, Passe correctly finds and enforces 17 policies for the application. Most of these are data integrity policies. For example, only the author of a post is authorized to change the content of that post. However, the post's score is an aggregation of other users' votes and each user is allowed to contribute only one vote to this score. Passe captures this by constraining the queries which log the votes, and the queries which tally those votes. Constraints are applied on the `up-vote` and `down-vote` views. These views issue an insertion query which puts a new row into a link vote table, or an update query which changes the direction of the vote. Passe ensures that these queries are associated with the current user, and a database uniqueness constraint ensures that only one row exists per user. The application then updates the `score` field of the news item by issuing a `COUNT` query and a subsequent `UPDATE` query. Passe ensures that the updated score value is exactly the result of the associated count query.

CMS Platform: `django-articles`, one of the applications we tested on Passe, provides a simple CMS platform that allows users to create and manage a blog. New articles can be added with a web interface, and it supports multiple authors, article groups, and static pages.

This CMS application, like many Django applications, includes and relies on the `django-admin` module. This module provides a full-featured interface for modifying, adding, and deleting models in any application. To support any significant set of Django applications, Passe must support the admin module, and it does. Passe is able to infer strong constraints for this module. The admin module makes calls to Django's permissions library, and Passe infers control-flow constraints based on those calls. In the case of the CMS platform, the Passe proxy requires that a view present a token possessing the "Article.add" permission to add a new article to the database.

Passe additionally enforces integrity constraints on queries creating and modifying blog posts. In particular, Passe requires that a new post's author matches the current user and that the content of that article matches the user's request variables. Passe ensures that a user is allowed to post articles by checking that user's coarse permission set.

Web Forum: We also tested a simple forum library `django-forum` under Passe. This application allows developers to run an online forum, which supports user accounts, user groups, and multiple forums with display permissions. To support creating new groups and forums, the application uses the default `django-admin` interface.

`django-forum` supports linking particular forums to user groups, such that a given forum is hidden to and cannot be modified by users outside of that group. In the application, this access control is implemented by (i) retrieving the set of groups for the current user, (ii) issuing queries both to fetch those forums with a matching group and those with a public (null) group, and (iii) fetching all threads to those forums. Note that the application never explicitly declares the policy that users should only be able to see threads in their groups; it is only implicit in the application's execution. Passe makes this policy explicit, and it is enforced by the database proxy as a data-flow dependency (Figure 8).

The `django-forum` application also provides an example of a control-flow constraint. Before adding discussion threads to a forum, a view checks whether the current user has access to the forum. This check involves a SQL query which fetches the set of allowable users, and then, in Python, the view checks whether the current user is in that set. If the user has access, the view fetches the appropriate forum's ID, and uses this ID as an argument for creating a new thread.

In this example, the first query is a control-flow dependency. Later queries do not have arguments reflecting its return values, and thus these three queries do not form a data-flow dependency. However, the Passe analyzer correctly determines that the first query, along with the current user, has tainted the control-flow, and infers a constraint that the current user must be contained within the set of allowable users for the second and third queries to be executed.

```
SQL0:
 SELECT forum ... WHERE group in user.groups
   and user.id == y
 DATA-FLOW CONSTRAINT: y = (current_user)
SQL1
 SELECT forum ... WHERE group == NULL
SQL2:
 SELECT thread ... WHERE id == x
 DATA-FLOW CONSTRAINT: x = (SQL0.ID OR SQL1.ID)
```

Figure 8: `django-forum` executes three SQL statements to retrieve the accessible threads for the current user. SQL2 is restricted to only return threads from forums discovered in SQL1 or SQL0. These assertions chain these queries together, enforcing their data-flow relationship.

F. Common Web Vulnerabilities and Their Effects on Passe

Though Passe protects against more general exploits, it is important to understand how various common web vulnerabilities are relevant to Passe and its protections.

SQL Injection. For the purposes of Passe, SQL Injection attacks are mitigated by the use of Django, which strongly encourages developers to issue queries using separate query strings and query arguments. For applications which do not use this argument passing method, Passe will prevent these from causing SQL injection vulnerabilities. This is because Passe's database proxy expects that the only parts of a query allowed to change are the arguments to the SQL query. If the query changes from string manipulation rather than SQL argument passing, then the query will no longer be recognized as the "same" query witnessed during training, and Passe's proxy will reject the query. This requires that the developer change their application and adopt the preferred approach.

Cross-Site Request Forgery. Django mitigates CSRF attacks by requiring forms to carry CSRF tokens, which are used to check that requests came from a legitimate web request. If an attacker compromises a view, they can always forgo this protection for that particular view. Worse, however, this attacker may be able to steal CSRF tokens and use them for other views. To mitigate this attack, we can associate CSRF tokens with particular views, and thus prevent a view compromise from affecting other views.

Click Jacking. An attacker may attempt to load web pages secured by Passe in a HTML frame and maliciously access the page using Javascript. In order to prevent this attack both from external sites, and from an internal view which an attacker has compromised, Passe adds the `X-Frame-Options` header to all outgoing responses. This prevents the web page from being displayed inside of a frame.

VIII. Performance Evaluations

To evaluate the performance of Passe, we ran three applications in Django and Passe, measuring the time to fetch each application's home page. Our testing system used Gunicorn 0.15 to dispatch web requests through WSGI, PostgreSQL 8.4 as the backing database, Python 2.7, and Ubuntu 11.04. Gunicorn is the default Python HTTP server on popular hosting platforms such as Heroku. Because Django and Passe do not require many of the features of Apache, lighter-weight servers such as Gunicorn may be used. Our test server had 2 Intel Xeon E5620 CPUs, with 4 cores each clocked at 2.40GHz. Because Passe's database proxy pools database connections, in order to fairly compare the throughput with plain Django, the plain Django version used pgpool for connection pooling. (Without pgpool, Passe outperformed vanilla Django in throughput measurements.)

In order to understand Passe's performance on real applications, we examine performance on the case study applications we detailed earlier. Further, to explore the base overhead Passe imposes on simple applications, we developed a benchmark application that immediately renders a response from memory without using the database.

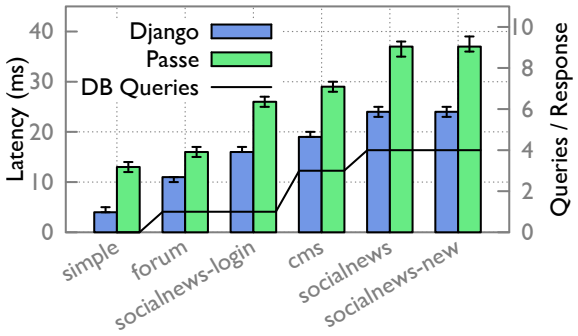

Figure 9: Request latency to access home page of applications. Error bars indicate the 90th and 10th percentiles. The black line represents the number of queries each view issues per request.

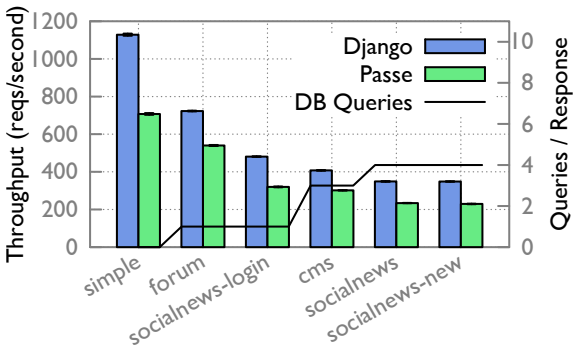

Figure 10: Mean throughput of applications running in Passe and Django over five trials (error bars show stddev). Each trial involves 8K requests. The black line represents the number of queries each view issues per request.

A. Latency and Throughput

We measured latency of requests by repeatedly fetching application pages with a single user on the JMeter benchmarking platform. Figure 9 plots the latencies of 1000 requests. While Passe's latency overhead of 5-13 ms is not insignificant, applications and service providers often target much larger end-to-end latencies, e.g., Amazon cites 300 ms as their desired 99.9% latency for requests [19]. In comparison, Passe's overhead is not an excessive burden.

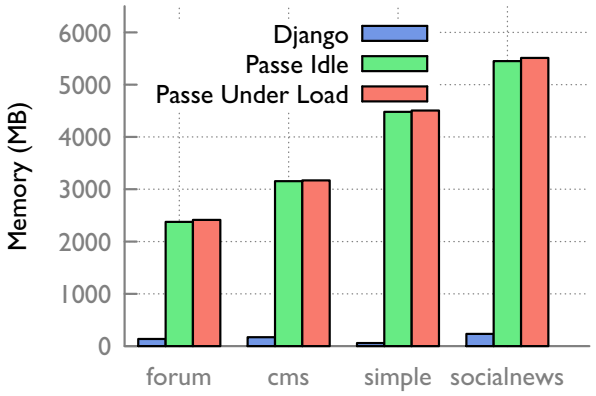

Figure 11: Memory usage of Passe in comparison to normal Django.

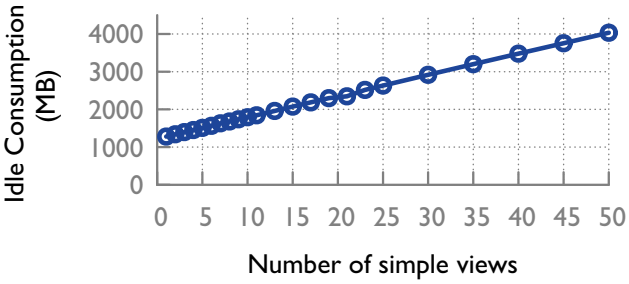

Figure 12: Idle memory consumption of Passe while varying the number of views spawned by the `simple` application benchmark.

To characterize the throughput overhead of Passe, we used JMeter configured with 40 simultaneous users. We ran Gunicorn with 8 worker processes (one worker per CPU core). When running Django, each of these workers ran a separate Python interpreter with a separate copy of the Django application. When running Passe, each worker ran a separate Python interpreter connected to separate instances of Passe.

Throughput results (Figure 10) show that the cost of Passe may vary greatly based on the application. For the simple benchmark, which requires little I/O and no database operations, Passe reduces throughput by 37%. However, for actual applications that often require more I/O and query the database, such as the forum or CMS applications, we find Passe reduces throughput by about 25%.

B. Memory Overhead

In addition to latency overhead, Passe adds memory overhead, as each view requires a separate OS process. To characterize this overhead, we measured the total memory consumption of Passe while running 8 workers for our benchmark applications (Figure 11). While this memory overhead is large, it does not increase significantly under load. Rather, the memory overhead corresponds to the number of views in the application, each of which causes 8 separate processes to be spawned. In order to understand the behavior of this relationship, we varied the number of views in our `simple` application. Figure 12 shows the linear correspondence between the number of views and the memory consumption. Based on these measurements, modern servers with a fairly modest 16 GB of RAM would be able to run applications with hundreds of views. In comparison, the most complicated application we tested had only 47 views. While on dedicated servers, memory consumption may not be prohibitive, in memory-constrained environments (e.g., hosted virtual machines), this cost may be excessive. In these cases, a more aggressively optimized version of Passe could make better use of copy-on-write memory and other memory saving techniques to reduce this overhead. Such performance improvements remain as future work.

IX. Related Work

Intrusion Detection Systems: Our approach is most related to previous work in intrusion detection. Like intrusion detection systems, Passe uses dynamic analysis of execution to "learn" the normal or intended behavior of the application. Some work, such as DFAD [7] or SwitchBlade [20], has similarly used taint tracking to check whether a detected deviation in program behavior was the result of a code injection attack. This does not address attacks where code injection was not the root cause. Other intrusion detection work, such as Swaddler [21], SENTINEL [22] and DoubleGuard [8], has analyzed internal program state to infer potential invariants using Extended Finite State Machine modeling.

Passe differs from this work in two major ways. First, Passe actively splits applications into sandboxed components, allowing Passe to more easily infer constraints and to support more restrictive constraints than could otherwise be achieved. Second, Passe's enforcement mechanisms operate without instrumenting application code or requiring a stateful database proxy. This prevents arbitrary code exploits from defeating the system and allows the proxy to be implemented in a scale-out architecture.

AutoISES [23] attempts to infer relationships between security checks and data accesses. In general, Passe cannot know which queries are "permission checks," and so must make inferences about the relationships between queries.

Automated Vulnerability Detection: Some work in vulnerability detection has used a similar inference model to find potential errors in application code [24]. Several systems for detecting web vulnerabilities use program analysis to find bugs which can be readily identified once they are found (e.g., application crashes, malformed HTML, or forced browsing) [25, 26, 27]. For finding general data-flow violations which are harder to characterize, Passe cannot use the same kind of analysis.

Other work attempting to detect data-flow vulnerabilities has used a similar approach to Passe. For example, in [28], "normal" usage of the application is analyzed dynamically. In [29], taint tracking is used to identify cases in which user input has not been properly sanitized. Such work is focused on finding bugs rather than developing policies for

a secure runtime, as in Passe. Thus, many of these projects' mechanisms cannot be applied to Passe's setting.

Decentralized Information Flow Control: Passe differs significantly from traditional DIFC systems [30, 31, 32, 33, 34, 35], as Passe learns application policy during an analysis phase, while DIFC systems require developers or users to explicitly annotate data or code. Because DIFC systems require reasoning about information labels, application code may still be vulnerable to aberrant behavior. This is true even for automatic instrumentation systems such as SWIM [36], which still requires developer-supplied policies. Hails [18] applies the DIFC model to web applications while using a shared data store. Hails requires applications to be written in a safe subset of Haskell. Hails' policies provide stronger guarantees than Passe, but require explicit policy specification.

XSS Protection: There has been significant work in preventing XSS attacks. Much of this work has focused on filtering and sanitizing user inputs. Passe addresses a stronger class of threats, in which the attack has compromised part of the application code. Other work allows applications to specify exactly which scripts should be executed and in what form [37, 38], or focuses on using the proposed CSP standard [11] to separate trusted script sources from the data of the HTTP response [39]. Other client side solutions use taint tracking or fine-grained policies to limit the threat of XSS attacks [40, 41] XSS-Guard learns the set of scripts a web application typically constructs, and blocks unrecognized scripts [42]. While these approaches may work in Passe's setting, the approach we chose reuses the view-level isolation model from the rest of our system's protections. This allows us to unify isolation at the server with isolation at the client.

Other Approaches to Web Security: Resin [43], a system which uses explicit policies to specify allowable data flows, can provide many of the same properties as Passe. However, because Resin relies on data-flow tracking in the application during runtime, it is susceptible to remote code exploits.

Systems such as Diesel [44] and OKWS [45] provide web frameworks strongly rooted in the principle of least privilege. Passe provides much richer constraints and does not require explicit separation from the developer. SELinks [46] supports enforcing explicit security policies near the database. Unlike Passe, policies are compiled into user-defined functions at the database.

X. Conclusion

This paper introduces Passe, a system that provides security guarantees for applications using a shared data store. Passe decomposes applications into isolated views which execute in sandboxed environments. Passe enforces the integrity of data queries by using cryptographic tokens to preserve learned data and control-flow dependencies. In doing so, Passe infers and enforces security policies without requiring developers to specify them explicitly (and sometimes erroneously).

Our Passe prototype is capable of executing unmodified Django applications. We test Passe on eleven off-the-shelf applications, detail some of its inferred constraints, demonstrate several examples of security vulnerabilities it prevents, and show that it adds little performance overhead.

Acknowledgments

We thank the anonymous PC and our shepherd Andrei Sabelfed for the constructive feedback. We are grateful to Nickolai Zeldovich for insightful discussion about our project. We are also thankful to Edward Felten, Arvind Narayanan, Ariel Rabkin, David Shue, Wyatt Lloyd, Matvey Arye, Ariel Feldman, Marcela Melara, Xiaozhou Li, and Amy Tai, for all of their helpful comments during the course of this project. This research is supported by NSF Award CSR-0953197 (CAREER) and a Sloan Research Fellowship.

References

[1] M. Schwartz, "Hackers hit Symantec, ImageShack, but not PayPal," http://www.informationweek.com/security/attacks/hackers-hit-symantec-imageshack-but-not/240049863, 2012.

[2] T. Smith, "Hacker swipes 3.6m Social Security numbers, other data," http://usat.ly/TkBM0H, 2012.

[3] T. Preston-Warner, "Public key security vulnerability and mitigation," https://github.com/blog/1068-public-key-security-vulnerability-and-mitigation, 2012.

[4] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proc. IEEE*, vol. 63, no. 9, 1975.

[5] D. Brumley and D. Song, "Privtrans: Automatically partitioning programs for privilege separation," in *Proc. USENIX Security*, 2004.

[6] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng, "Secure web applications via automatic partitioning," in *Proc. SOSP*, 2007.

[7] S. Bhatkar, A. Chaturvedi, and R. Sekar, "Dataflow anomaly detection," in *Proc. IEEE S & P*, 2006.

[8] M. Le, A. Stavrou, and B. Kang, "Doubleguard: Detecting intrusions in multitier web applications," *IEEE TDSC*, vol. 9, no. 4, pp. 512–525, 2012.

[9] https://www.djangoproject.com/weblog/2011/nov/, 2011.

[10] http://wiki.apparmor.net/, 2012.

[11] "Content Security Policy 1.1," www.w3.org/TR/CSP11/, Jun 2013.

[12] D. Akhawe, P. Saxena, and D. Song, "Privilege separation in HTML5 applications," in *Proc. USENIX Security*, 2012.

[13] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proc. IEEE S & P*, 2010.

[14] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proc. ISSTA*, 2007.

[15] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for java," in *Proc. ACSAC*, 2005.

[16] N. Poole, "Code execution via YAML in JS-YAML Node.js module," https://nealpoole.com/blog/2013/06/code-execution-via-yaml-in-js-yaml-nodejs-module/, Jun 2013.

[17] A. Patterson, "Serialized attributes YAML vulnerability with Rails 2.3 and 3.0 [cve-2013-0277]," https://groups.google.com/d/msg/rubyonrails-security/KtmwSbEpzrU/NzjxkM7HLjAJ, Feb 2013.

[18] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo, "Hails: Protecting data privacy in untrusted web applications," in *Proc. OSDI*, 2012.

[19] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proc. SOSP*, 2007.

[20] C. Fetzer and M. Süßkraut, "Switchblade: enforcing dynamic personalized system call models," in *Proc. EuroSys*, 2008.

[21] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna, "Swaddler: An approach for the anomaly-based detection of state violations in web applications," in *RAID*, 2007, pp. 63–86.

[22] X. Li, W. Yan, and Y. Xue, "Sentinel: securing database from logic flaws in web applications," in *Proc. ACM CODASPY*, 2012.

[23] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou, "AutoISES: Automatically inferring security specifications and detecting violations," in *Proc. USENIX Security*, 2008.

[24] D. Engler, D. Y. Chen, S. Halem, A. Chou, and B. Chelf, "Bugs as deviant behavior: a general approach to inferring errors in systems code," in *Proc. SOSP*, 2001.

[25] P. Bisht, T. Hinrichs, N. Skrupsky, and V. Venkatakrishnan, "Waptec: whitebox analysis of web applications for parameter tampering exploit construction," in *Proc. CCS*, 2011.

[26] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding bugs in dynamic web applications," in *Proc. ISSTA*, 2008.

[27] F. Sun, L. Xu, and Z. Su, "Static detection of access control vulnerabilities in web applications," in *Proc. USENIX Security*, 2011.

[28] V. V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, "Toward automated detection of logic vulnerabilities in web applications," in *Proc. USENIX Security*, 2010.

[29] A. Nguyen-tuong, S. Guarnieri, D. Greene, and D. Evans, "Automatically hardening web applications using precise tainting," in *Proc. 20th IFIP International Information Security Conference*, 2005.

[30] A. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *TOSEM*, vol. 9, no. 4, pp. 410–442, 2000.

[31] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris, "Labels and event processes in the Asbestos Operating System," in *Proc. SOSP*, 2005.

[32] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in HiStar," in *Proc. OSDI*, 2006.

[33] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard OS abstractions," in *Proc. SOSP*, 2007.

[34] W. Cheng, D. R. K. Ports, D. Schultz, V. Popic, A. Blankstein, J. Cowling, D. Curtis, L. Shrira, and B. Liskov, "Abstractions for usable information flow control in Aeolus," in *Proc. USENIX ATC*, 2012.

[35] Y. Mundada, A. Ramachandran, and N. Feamster, "Silverline: Data and network isolation for cloud services," in *Proc. HotCloud*, 2011.

[36] W. R. Harris, S. Jha, and T. Reps, "DIFC programs by automatic instrumentation," in *Proc. CCS*, 2010.

[37] M. Ter Louw and V. Venkatakrishnan, "Blueprint: Robust prevention of cross-site scripting attacks for existing browsers," in *Proc. IEEE S & P*, 2009.

[38] Y. Nadji, P. Saxena, and D. Song, "Document Structure Integrity: A robust basis for cross-site scripting defense." in *Proc. NDSS*, 2009.

[39] A. Doupé, W. Cui, M. H. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna, "deDacota: Toward preventing server-side XSS via automatic code and data separation," in *Proc. CCS*, 2013.

[40] L. A. Meyerovich and B. Livshits, "Conscript: Specifying and enforcing fine-grained security policies for Javascript in the browser," in *Proc. IEEE S & P*, 2010.

[41] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis." in *Proc. NDSS*, 2007.

[42] P. Bisht and V. Venkatakrishnan, *XSS-GUARD: precise dynamic prevention of cross-site scripting attacks*. Springer, 2008, pp. 23–43.

[43] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Improving application security with data flow assertions," in *Proc. SOSP*, 2009.

[44] A. P. Felt, M. Finifter, J. Weinberger, and D. Wagner, "Diesel: Applying privilege separation to database access," in *Proc. ASIACCS*, 2011.

[45] M. Krohn, "Building secure high-performance web services with OKWS," in *Proc. USENIX ATC*, 2004.

[46] N. Swamy, B. J. Corcoran, and M. Hicks, "Fable: A language for enforcing user-defined security policies," in *Proc. IEEE S & P*, 2008.