FlowWatcher: Defending against Data Disclosure Vulnerabilities in Web Applications

Divya Muthukumaran
Imperial College London

Dan O'Keeffe
Imperial College London

Christian Priebe
Imperial College London

David Eyers
University of Otago

Brian Shand
NCRS, Public Health England

Peter Pietzuch
Imperial College London

ABSTRACT

Bugs in the authorisation logic of web applications can expose the data of one user to another. Such data disclosure vulnerabilities are common—they can be caused by a single omitted access control check in the application. We make the observation that, while the implementation of the authorisation logic is complex and therefore error-prone, most web applications only use simple access control models, in which each piece of data is accessible by a user or a group of users. This makes it possible to validate the correct operation of the authorisation logic externally, based on the observed data in HTTP traffic to and from an application.

We describe FlowWatcher, an HTTP proxy that mitigates data disclosure vulnerabilities in unmodified web applications. FlowWatcher monitors HTTP traffic and shadows part of an application's access control state based on a rule-based specification of the *user-data-access* (UDA) policy. The UDA policy states the intended data ownership and how it changes based on observed HTTP requests. FlowWatcher detects violations of the UDA policy by tracking data items that are likely to be unique across HTTP requests and responses of different users. We evaluate a prototype implementation of FlowWatcher as a plug-in for the Nginx reverse proxy and show that, with short UDA policies, it can mitigate CVE bugs in six popular web applications.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access Controls

Keywords

Web application security; Data disclosure; HTTP Proxy; Policy

1. INTRODUCTION

Web application vulnerabilities are a major source of security incidents on the Internet. Different types of vulnerabilities have different mitigation strategies: e.g. data injection and validation bugs [37] can be prevented by using templates [39] or dynamic data tracking [28].

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

ACM 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813639 .

Data disclosure vulnerabilities, which expose the data of one user to another, however, are harder to protect against because they are typically caused by semantic bugs in the authorisation logic of web applications. In the 2013 OWASP ranking of web application security risks [38], four out of the top ten risks are related to incorrectly implemented access control checks; the 2014 Website Security Statistics Report [53] states that information leakage, which includes data disclosure, is the second most prevalent vulnerability in web applications.

Web applications typically execute at a higher privilege level than that of individual users, which means that they must implement their own checks for enforcing a given access control model. Protecting against data disclosure vulnerabilities is thus difficult because a single omitted check may expose user data. While the core components of web applications receive much scrutiny by developers, missing checks often exist in third-party plug-ins and extensions. In the twelve months from August 2013, 16% of security bugs reported in CVE [9] for Drupal [16] were related to unauthorised data disclosure in plug-ins.

Existing proposals to mitigate data leakage vulnerabilities have drawbacks, and, in many cases, missing access control checks are only discovered after the vulnerability has been exploited [34]: *program analysis* techniques [50] can detect missing checks but require access to the source code, are language-specific and struggle with complex applications; techniques based on *input validation* [25] focus on injection attacks, which may cause data disclosure, but they cannot detect data disclosure due to semantic bugs in the authorisation logic; and *anomaly detection* approaches [26] can prevent unauthorised data disclosure if it constitutes a deviation from regular application behaviour, but for many applications the "normal" behaviour cannot be captured reliably.

Instead of establishing the correctness of all access control checks that are dispersed throughout an application's source code, our idea is to adopt a "defence-in-depth" approach that *validates* the correct operation of the access control policy *outside* of the application. This is enabled by the observation that many web applications, such as Drupal [16], WordPress [54] or DokuWiki [12], only implement relatively simple access control models: e.g. they distinguish between users (or groups of users) that have access rights to data objects such as web pages, posts or comments. It is thus possible to introduce an external *proxy* that observes the HTTP request and response traffic of all users and, based on a specification of the *intended* access control policy of the application, detects and prevents unauthorised data disclosure.

A proxy-based approach for mitigating unauthorised data disclosure has several benefits: it can be applied across a range of different web applications, as long as the enforced policy can be tailored.

The proxy does not need to implement the full policy of the application, but can only check a subset of it. Due to its smaller code base and single enforcement point compared to checks scattered throughout the application, the correct enforcement of the policy by the proxy is easier to guarantee. Finally, a proxy should have no performance impact and does not require modifications to the application, making it applicable to closed-source web applications and services.

The approach raises two challenges, however, which we overcome in this paper: (i) how does the proxy express the access control policy of an application and keep it up-to-date as new users, groups and data objects are added? For example, when a new post is created by a user in WordPress, the proxy must associate this new data object with the users and groups that are permitted to access it. Policies should be written once by application developers who understand the intended access control model of their application; and (ii) how does the proxy efficiently track user data across HTTP requests and responses of different users in order to detect violations of the access control policy? Data tracking should be effective at identifying unauthorised data disclosure, be efficient to implement and not require changes to web applications.

We describe **FlowWatcher**, a proxy that mitigates data disclosure vulnerabilities in web applications by monitoring their HTTP traffic and prohibiting incorrect data flows.

User-data-access policy. FlowWatcher relies on a specification of the intended access control policy, which is written once by application developers in a domain-specific rule-based language as a *user-data-access (UDA)* policy. A UDA policy encodes the dynamic access control model of an application: it describes how HTTP requests and responses change the access control state. UDA policies are typically concise, making it easier to establish their correctness. Since FlowWatcher adds an additional layer of security, UDA policies do not have to be complete but can capture a subset of the application's access control model.

Dynamic policy evolution. A UDA policy contains rules that link HTTP requests and responses to (i) the definition of new users, groups or data objects and to (ii) updates of the access control policy. Each rule can match an HTTP request according to its URL, header fields or body content such as form fields. After matching, a rule can update the current access control state, e.g. give a user access to a data object, or add them to a group.

Data tracking. By intercepting the authentication method of the application, FlowWatcher associates each HTTP request and response with a user. As defined by the UDA policy, FlowWatcher selects user data objects in HTTP requests, which may be stored in form fields, and tracks ones that are likely to be *unique* due to their length or semantics, such as social security numbers or postal addresses. When a tracked object appears as part of another user's HTTP response, FlowWatcher decides if the data access is authorised based on its shadow access control state.[1]

Evaluation. We show that UDA policies for real-world web applications are *simple*—the policies for Drupal [16], WordPress [54] and DokuWiki [12] have 43, 23 and 26 lines, respectively; they are *effective*—FlowWatcher can mitigate 9 data disclosure vulnerabilities from the CVE database reported for 6 popular web applications; and FlowWatcher is *efficient*—its implementation as an Nginx plug-in [35] does not have a measurable impact on application throughput or latency.

[1]Note that FlowWatcher relates access control decisions to externally observable dataflows of unique data—its goal is not to prevent disclosure of low-entropy fields, e.g. a hometown that may be common to multiple users

Next we motivate the problem and position our approach with respect to other techniques. §3 describes our domain-specific policy language, giving examples of how it expresses the access control models of web applications. We present the design and implementation of FlowWatcher in §4 and the results of our experimental evaluation in §5. The paper finishes with a comparison of FlowWatcher to related work (§6) and conclusions (§7).

2. PREVENTING DATA DISCLOSURE

Next we discuss data disclosure vulnerabilities in web applications (§2.1), categorise existing approaches to mitigate web vulnerabilities (§2.2), and explain the idea behind our approach, emphasising the differences to previously proposed techniques (§2.3).

2.1 Data disclosure in web applications

It is difficult to implement an access control model correctly in a web application. Due to the semantic mismatch between the access control model of the underlying platform (e.g. UNIX access control lists) and that of the application, web applications must execute with a superset of all privileges of their users. This requires them to guard operations that access or update user data with access control checks to ensure their compliance with a given security policy. Access control checks are typically sprinkled throughout the source code of an application.

For example, the convention for Drupal modules [16] is to guard all database queries by tagging them with the string "node_access", which causes an access control check to occur. The following PHP fragment issues a query that returns the most recently modified pages to a user:

```
1 $nids = $query->fields('n', array('nid'))
2               ->orderBy('n.changed', 'DESC')
3               ->range(0, $number)
4               ->addTag('node_access')
5               ->execute()->fetchCol();
6 $nodes = node_load_multiple($nids);
```

The tag in line 4 filters query results based on the permissions of the user issuing the query. Omitting the tag by accident in a complex query would reveal private pages of other users in the result set.

More generally, a vulnerability that leads to data disclosure may be caused by several types of bugs:

- *Missing check:* An attacker may access another user's data by exploiting the fact that no check is carried out for a particular data object. For example, Drupal suffered from a vulnerability in 2012 in which the Organic Groups add-on module was missing an access check that allowed non-members to view information about private groups (CVE-2012-2081).
- *Avoided check:* An attacker may access data without authorisation by triggering a code path that avoids an existing check, e.g. by inducing an error condition. For example, the phpBB [43] application contained a bug (CVE-2010-1627) that exposed information in private forums to non-members via RSS feeds, even if the forums were not accessible directly to the users.
- *Incorrect check:* An access control check may be implemented incorrectly—a plug-in developer may misunderstand the access control model of the application or make an incorrect assumption about an API. An example of this is an execution-after-redirect vulnerability in which a check is performed and unauthorised users are redirected to an error page, but the privileged operation is executed regardless, returning sensitive information in the HTTP response [13].

Approach	Types of bugs	Implementation	Execution time	No source code required	Unmodified runtime	No training phase	Supports policy evolution
Missing access check detection [32, 50, 51]	access control	tool	offline	✓	*n/a*	*n/a*	✓
Input validation testing [2, 3, 25, 52, 4, 49]	data sanitisation	tool	offline	✗ | ✓	*n/a*	*n/a*	✓
Execution anomaly detection [8, 21, 30]	semantic	interpreter	online	✗	✗	✗	✗
Traffic anomaly detection [24, 26, 29, 42]	semantic	proxy	online	✓	✓	✗	✗
Dynamic data tracking [10, 55]	access control	interpreter	online	✓	✗	✓	✗
FlowWatcher	**data disclosure**	**proxy**	**online**	✓	✓	✓	✓

Table 1: Existing techniques to mitigate data disclosure vulnerabilities in web applications

Protecting against the above bugs is hard because the implementation of an access control model is highly application-specific. While authentication checks are typically performed only in a single core component of a web application, according to a small number of valid strategies (i.e. using session cookies or authentication HTTP headers), authorisation logic affects *all* modules of an application, including third-party plug-ins, and can be implemented in many different ways.

2.2 Mitigating unauthorised disclosure

The research community has investigated how to protect against data disclosure vulnerabilities in web applications and proposed a range of techniques (see Table 1).

Missing access check detection. Offline techniques using static program analysis were proposed to discover application code paths with missing access control checks [32, 50, 51]. Such techniques require access to the source code, are language-specific and make assumptions about the architecture of the application, such as distinct application-specific roles usually involving different program files [50]. They also suffer from the intrinsic problems of static analysis: the analysis is conservative, potentially reporting false positives, and is unable to support arbitrarily complex applications.

Input validation testing. Many disclosure vulnerabilities, such as ones triggered by cross-site scripting (XSS) and SQL injection attacks (SQLI), are caused by the incorrect validation or sanitisation of user input data. Offline testing techniques exist that, based on generated user input, analyse or track the propagation of user data through the application in order to discover code paths without appropriate data sanitisation. Whitebox techniques [2, 4] exploit knowledge of the application source code for targeted testing; blackbox techniques [3, 49] are limited to observing the external behaviour of the application to detect bugs. Since these techniques only focus on the incorrect usage of input data, they cannot discover a more general class of bugs related to mistakes in the access control logic.

Anomaly detection. A different class of techniques treats the detection of semantic bugs as an anomaly detection problem. In a training phase, the correct behaviour of the application is observed. Execution-based approaches [8, 21, 30] record behaviour in terms of internal applications states, e.g. by instrumenting the language interpreter; network-based approaches [24, 26, 29, 42] train the model according to observed network traffic. During an attack, the anomalous deviation from the correct behaviour is reported.

The effectiveness of such techniques depends on how comprehensively the training phase captures application behaviour—new but correct runtime behaviour leads to false positives. This makes it hard to have realistic training workloads for many applications.

Dynamic data tracking. Recent proposals for shadow authentication and authorisation [10, 55] detect access control bugs by tracking data in an application. This relies on modified language interpreters that can record the flow of user data in order to establish that checks are carried out correctly. Nemesis [10] maintains authentication data externally, and carries out shadow checks before operations execute on resources such as files or database tables; Resin [55] associates data with policy objects, which then execute shadow access control checks.

While such approaches can accurately identify data disclosure, dynamic data tracking requires non-standard language interpreters and runtime systems and has a performance overhead, which is challenging in production environments [48]. In addition, these approaches do not support the specification of dynamic access control policies in a high-level language, precluding policy evolution based on user actions.

2.3 Proxy-based disclosure detection

As shown in Table 1, the goal of FlowWatcher is to provide a practical approach for the mitigation of data disclosure vulnerabilities in today's web applications. Compared to previous approaches, we explore a different point in the design space: instead of modifying the source code of applications or the runtime system executing them, we want to provide a solution that facilitates adoption because it can be applied to any black-box application. In addition, we want to introduce a negligible performance overhead, not rely on a brittle training phase, and support the change of access control policy over time.

We make the observation that, for many web applications, the underlying access control model is relatively simple, and that the majority of usage involves no more than this simple model. Web applications such as WordPress [54], Drupal [16], Evernote [20], Dokuwiki [12], phpMyAdmin [44] and phpBB [43] all model access control decisions based on access control lists: they associate principals, such as users or groups of users, with access permissions to data objects, such as articles, posts and comments. This makes it possible to *validate* the correctness of access control checks externally, i.e. outside of the web application, as long as the current state of the access control model is known.

Our hypothesis is that we can provide a *web proxy* to detect and prevent unauthorised data disclosure. The proxy interprets observed network traffic into and out of an application based on an understanding of the application's intended access control policy. The benefit of a proxy-based approach is that it can be applied transparently to existing deployments without changes to applications or their language interpreters, and can support encrypted traffic by terminating the encrypted connection. In addition, an adequately-provisioned proxy does not impact performance.

However, using an external web proxy to detect data disclosure introduces two challenges: (i) the *dynamic access control policy* of the application must be expressed in a way that permits the proxy to maintain the current access control state for the application (§3); and (ii) the proxy must *track* the propagation of data from one user to another in order to detect that data was disclosed (§4).

2.4 Threat model

Our approach covers threats from both authenticated and unauthenticated users of an application who want to read data belonging to other users by exploiting data disclosure vulnerabilities, as described in §2.1.

We assume that the backend data store used by the application is secure and not directly accessible to attackers. Our approach also does not cover SQL injection attacks, in which the attacker can obfuscate the data leaked—as discussed above, specialised techniques for mitigation already exist.

In addition, we only focus on threats to data confidentiality and not data integrity. For example, we cannot prevent one user from taking advantage of an application vulnerability in order to modify the data belonging to another user, because it is possible for the application to make updates in the backend data store without the proxy being aware of this. The proxy can only mediate when data is returned to the user.

3. USER-DATA-ACCESS POLICY

To address the challenge of maintaining a dynamic access control policy, we propose a new domain-specific language, the *user-data-access (UDA) policy language*. It allows application developers to specify the intended access control model for their applications by relating *users* to the *data* that they are permitted to *access*.

There are two interesting requirements for the language. First, as the web proxy can only observe the client-server HTTP communication, the language must express the access control policy using only the information contained in HTTP requests and responses. The entities in the access control model, i.e. the data objects to be protected and the identities of users and groups, must be represented in terms of HTTP request URLs, request and response headers and form field data in requests. In a UDA policy, this is done using *definition rules*.

Second, the language must support dynamic evolution of access control policy by reacting to *changes* in the policy, such as updates to access control lists or group memberships. The policy language must therefore relate HTTP requests that administer the policy to policy changes. In a UDA policy, this is done using *update rules*.

We first describe the *entities* that are manipulated by definition and update rules. We use a UDA policy for the Drupal [16] content management system (version 6), which is representative for that of other applications, as a running example.

3.1 Entities

A UDA policy stipulates that certain data objects that belong to a given user should only be visible to a subset of other users. There are two entities in UDA policies: (a) *data objects* that contain data items, which represent the user-generated data that should be protected; and (b) *users* and *groups* of users that possess access rights for data objects.

Data objects. A *data object* o_i is created by the application in response to user input. When an application creates a new data object, we assume that it is assigned a unique application-specific identifier ID_i. For example, an article in Drupal is a data object, and it is assigned a unique URI.

A data object o_i contains a set of *data items*, $D(o_i) = \{d_1, d_2, \ldots\}$, for which access control must be enforced. Data items are entered by users in form fields of HTTP requests and should be returned in an HTTP response to a user only if allowed by the access control policy. For example, when creating an article in Drupal, a user enters values for the `title` and `body` form fields, which are the data items for the `article` data object.

We impose the constraint on each data object o_i that it must have a high likelihood of referring to a *unique* set of data items $D(o_i)$. As we explain in §4.3, this permits the web proxy to *track* the data items of data objects across HTTP requests and responses, thus observing the flow of data between users.

For example, a data object $D(o_1) = \{ssn\}$ may contain a unique form field that stores a social security number. Alternatively, the combination of multiple data items may be unique: a data object may contain two data items that store a postal address, $D(o_2) = \{street\_address, postcode\}$. While each data item on its own is not unique, their combination has a high probability of being unique.

Users and groups. A *user* u_i is a principal that can make authenticated requests for data objects, i.e. they have access to a set of data objects, $\{o_1, o_2, \ldots\}$. Typically a web application establishes a *session* into which an authenticated user's requests are collected.

Users may also be organised into *groups*, which simplifies the assignment of permissions. Each group g_i has a unique identifier. For example, the Organic Groups module [18] in Drupal allows users to create and manage groups, each identified through a URI. Group members can create objects of any content type that Drupal supports, such as articles or pages, and share them with other members of the group.

3.2 Rules

To link to HTTP requests and responses, UDA policies are *rule-based*, similar to firewall rules. Rules are triggered based on request URLs and constraints on values. They define identifiers for groups, users, and data objects, thus allowing the proxy to mirror the access control lists maintained by the web application.

Rules in UDA policies can be of one of two types:

1. *Definition/removal rules* (`+` or `-`) intercept the creation or deletion of entities in the policy, i.e. data objects, users or groups. They extract required information about new entities to add them to the access control policy, such as identifiers. The proxy can then maintain a mapping between entities in the UDA policy and references to these entities in HTTP requests or responses.
2. *Update rules* (`*` or `->` or `-/>`) are related to changes of the access control policy, such as updating an existing data object, adding a group to the access control list of a data object, or removing a user from a group. They refer to previously-defined entities.

Syntactically all rules have a *rule preamble* and a *rule body*:

```
<type> <URL spec> [if <constraint>] /* preamble */
  { /* rule body */ }
```

The rule preamble includes a rule type, a URL specification and an optional constraint: the *type* specifies whether it is a definition/removal or update rule; the *URL specification* describes the URLs for which the rule is triggered; the *constraint* indicates additional conditions that must hold for the rule to trigger, such as predicates on the values of form or header fields; finally, the *rule body* contains a set of assignments or mapping statements that update the access control policy maintained by the proxy. As described below, these statements refer to fields associated with the entities from the rule preamble.

For a received HTTP request, the proxy matches the URL specification and other request parameters against each rule preamble and, if satisfied, executes the statements in the rule body.

Listing 1 shows part of the rules of the UDA policy for the access control model used by Drupal.[2] Line 9 is a a rule preamble:

Listing 1: Excerpt from the UDA policy for Drupal

```
/* Definition rules */
1  user+ "/*" if (res_hdr "Set-Cookie" re"SESS.*")
2  {  id := formfield "name", res_hdr "Location"
       ↪ re"/?q=user/([0-9]+)";
3     token := res_hdr "Set-Cookie" re"SESS.*"; }

4  group+ "/?q=node/add/group"
5  {  id := res_hdr "Location" re"/?q=node/([0-9]+)"; }

6  data+ Article re"/?q=node/add/article"
7  {  id := res_hdr "Location" re"/?q=node/([0-9]+)";
8     item := formfield "title", formfield "body"; }

9  data+ PrivateGroupName "/?q=node/add/group" if
       ↪ (formfield "og_private"="1")
10 {  id := res_hdr "Location" re"/?q=node/([0-9]+)";
11    item := formfield "title"; }

   /* Update rules */
12 data* Article re"/?q=node/[0-9]+/edit"
13 {  id = url re"/([0-9]+)/edit";
14    item[0] = formfield "title";
15    item[1] = formfield "body"; }

16 user -> group re"/?q=og/users/[0-9]+/add_user"
17 {  user.id = formfield "og_names";
18    group.id = url re"/([0-9]+)/add_user"; }

19 user -> data re"/?q=node/add/.*" if (formfield
       ↪ "status"="0")
20 {  user.id = authenticated_user;
21    data.id = res_hdr "Location"
          ↪ re"/?q=node/([0-9]+)"; }

22 group -> data re"/?q=node/add/*" if (formfield
       ↪ "status"="1")
23 {  group.id = formfield "og_groups";
24    data.id = res_hdr "Location"
          ↪ re"/?q=node/([0-9]+)"; }

25 group -> PrivateGroupName "/?q=node/add/group" if
       ↪ (formfield "og_private"="1")
26 {  group.id = res_hdr "Location"
       ↪ re"/?q=node/([0-9]+)";
27    PrivateGroupName.id = res_hdr "Location"
          ↪ re"/?q=node/([0-9]+)"; }
```

```
data+ PrivateGroupName "/?q=node/add/group" if
    ↪ (formfield "og_private"="1")
```

where `data+ PrivateGroupName` is the rule type (explained below), followed by a URL specification. The URL specification is expressed as a substring that is matched against the HTTP request URL. If the string is prefixed by `re`, it is executed as a regular expression. The constraint states that the request must have a form field "`og_private`" with the value "`1`".

In general, constraints (and assignments in the rule body) can use the keyword `formfield` to refer to form fields in the HTTP request body, `url` to refer to the request URL, and `req_hdr` and `res_hdr` to refer to header fields in the HTTP request or response, respectively. In each case, the data can be filtered with the help of regular expressions. For example, the following constraint checks if an HTTP response has a header field "`Set-Cookie`" that contains a session cookie:

```
if (res_hdr "Set-Cookie" re"SESS.*")
```

[2]The policy assumes that the Organic Groups module [18] is enabled. It omits the removal and additional object definition rules—the complete policy is in Appendix A.

3.3 Definition and removal rules

Definition and removal rules intercept the introduction of new users, groups and objects in the application, and their removal. The preamble of a definition rule starts with the name of the created entity followed by a "+" character (e.g. `group+`); analogously, removal rules include a "-" character. The rule body has a set of assignment statements that record more information about the entity.

User definition. For each HTTP request and response, the proxy must know the identity of the associated user. Therefore a user definition rule describes the authentication process of the application: it specifies how to intercept an authentication request and obtain a token for the user after the authentication has succeeded.

Listing 1 (line 1) shows the user definition rule for Drupal, with the following rule preamble:

```
user+ "/*" if (res_hdr "Set-Cookie" re"SESS.*")
```

The rule is prefixed with type `user+` to indicate that a new user is added. Since a login can be attempted from any Drupal page, the URL specification contains a wildcard. The constraint stipulates that, for successful authentication, the "`Set-Cookie`" header in the HTTP response must contain an entry for a session cookie beginning with "`SESS.*`".

Its rule body then contains two assignment statements:

```
{ id := formfield "name", res_hdr "Location"
    ↪ re"/?q=user/([0-9]+)";
  token := res_hdr "Set-Cookie" re"SESS.*"; }
```

The first assignment associates the authenticated user with two unique application-specific identifiers: the user name, which is extracted from the request form field "`name`"; and an internal identifier that Drupal assigns to the user, obtained from the redirect location in the response header.[3] The second assignment collects the authentication token (in this case a session cookie), which is associated with the user for a given session. The proxy thus maintains a mapping between the `user.id` and `user.token` variables, linking subsequent HTTP requests to that user session.

Group definition. A group definition rule links the creation of a new access control group to the policy maintained by the proxy.

As shown in Listing 1 (line 4), it starts with `group+`. Its rule body records the identifier of the new group. In this example, it is specified in the response header "`Location`" as a numeric value at the end of the URI, captured by a regular expression:

```
{ id := res_hdr "Location"
    ↪ re"/?q=node/([0-9]+)"; }
```

Data object definition. Object definition rules begin with `data+` and, when triggered, add a new data object to the access control policy. Each data object has a name for referral in update rules. For example, the UDA policy for Drupal has an object definition rule for articles called "`Article`" (Listing 1, line 6):

```
data+ Article re"/?q=node/add/article"
```

In its rule body, an object creation rule must specify the identity assigned to that data object; and the data items of that object that need to be tracked by the proxy.

[3]We configure Drupal to redirect users upon login to the user account page. Other applications may require a similar approach to allow FlowWatcher to capture user-specific identifiers.

For Drupal, the identity is assigned in the same way as for group creation discussed earlier; the list of data items tracked by the proxy are defined as the values of the"`title`" and "`body`" fields in the request form:

```
{  id := res_hdr "Location"
     ↪ re"/?q=node/([0-9]+)";
   item := formfield "title", formfield "body"; }
```

This specification of data items assumes that the combination of the "`title`" and "`body`" form fields has a high likelihood of containing unique data (when above a minimum length, see §4.3). If the proxy observes the same data in another HTTP response, it can thus assume that the user-generated data came from this original request.

3.4 Update rules

Update rules describe updates to (i) the data items tracked for a data object; (ii) the membership of groups by adding or removing users; and (iii) the access control lists of data objects by adding or removing access permissions of users or groups. Updates to data objects use the unary operator "`*`"; updates to group membership and access control lists use the binary operators "`->`" (for addition) or "`-/>`" (for removal). For such rules, the rule body links the entities in the access control policy to values derived from HTTP requests and responses.

Data object updates. Updates to the data items of a data object are tracked by a rule with type `data*`, as shown in Listing 1, line 12. The rule body identifies the data object using an identifier. The list of stored data items is updated with the values specified in the named form fields. For example, the following replaces the first element of the data item list "`item[0]`" with the data in the form field "`title`":

```
{  id = url re"/([0-9]+)/edit";
   item[0] = formfield "title";
   item[1] = formfield "body"; }
```

Group membership updates. Group memberships in the policy are maintained dynamically using a rule that begins with `user -> group`, as shown in Listing 1, line 16. Its rule body identifies the user and group in question:

```
{  user.id = formfield "og_names";
   group.id = url re"/([0-9]+)/add_user"; }
```

The user is identified by the form field "`og_names`", and the group is specified by a numeric identifier from the request URL.

Access control updates. To limit the set of users that can access a data object, each object maintains an access control list. Access control list updates are prefixed with

```
{user | group} -> object_name
```

where *object_name* refers to a previously-defined data object, specified in one of the object definition rules. If the *object_name* is `data`, the rule applies to all data objects without more specific rules.

For example, the following rule preamble (Listing 1, line 22) is triggered to add a group to the access control lists of all published data objects. In Drupal, the form field "`status`" is set to "`1`" to denote published content:

```
group -> data re"/?q=node/add/*" if (formfield
    ↪ "status"="1")
```

Drupal's policy states that published content should be visible to the groups specified in the value of the form field "`og_groups`", and a numeric identifier in the response header field "`Location`" determines the article identifier:

```
{  group.id = formfield "og_groups";
   data.id = res_hdr "Location"
       ↪ re"/?q=node/([0-9]+)"; }
```

Unpublished content, with the form field "`status`" set to "`0`", should not be visible to other users. In Listing 1, line 19, the UDA policy therefore only gives access to the authenticated user, as provided by the built-in variable `authenticated_user`:

```
{  user.id = authenticated_user;
   data.id = res_hdr "Location"
       ↪ re"/?q=node/([0-9]+)"; }
```

Since these two rules refer to the data object as `data`, they apply to any Drupal content type, which has the published/unpublished status set during creation (i.e. article, basic page, book page, etc.). This reduces the number of required update rules, making a separate rule for each content type unnecessary.

If there are exceptions to default rules, a UDA policy can include more specific rules. For example, for groups in Drupal, we only want the subset of private groups to be subject to access control enforcement. The policy thus has a specific rule `group -> PrivateGroupName` in line 25, which overrides the generic rule in line 22.

3.5 Discussion

The UDA policy language is designed to make it easy for application developers to express the essence of their access control model with a small set of rules. These rules then capture the behaviour of the access control checks that are potentially spread throughout the code base of the application. Since UDA policies are dynamic, i.e. they update access control lists based on changes to the policy from within the application, the UDA policy for a given application must only be written once and can then be reused across application deployments with different users, groups, data objects and access permissions. Finally, UDA policies can be enhanced over time, e.g. by adding new data objects supported by an application to achieve a more comprehensive tracking of user-generated content.

A drawback of UDA policies is that they rely on the specific format of URLs and HTTP requests and responses. This means that UDA policies must be updated if an application changes URLs or fields referred to in a policy. Since such changes are often accompanied by changes to the application functionality itself, we believe that is reasonable. Note that rules do not get triggered or matched if URLs or field names have changed, which can only lead to false negative detections until the policy is updated.

4. FLOWWATCHER DESIGN

FlowWatcher is designed as a reverse HTTP proxy that sits in front of an unmodified web application and observes requests from clients to the application and responses sent back to them (see Figure 1). It consults with the UDA policy provided by the application developer in order to update a *shadow access control policy* (or *shadow policy*) that mimics the access control policy implemented within the application. Since web application deployments already use reverse proxies for caching, load-balancing and monitoring, and such a proxy can be configured to handle HTTPS traffic, it is a natural choice for incorporating a "safety net" layer for black-box authorisation.

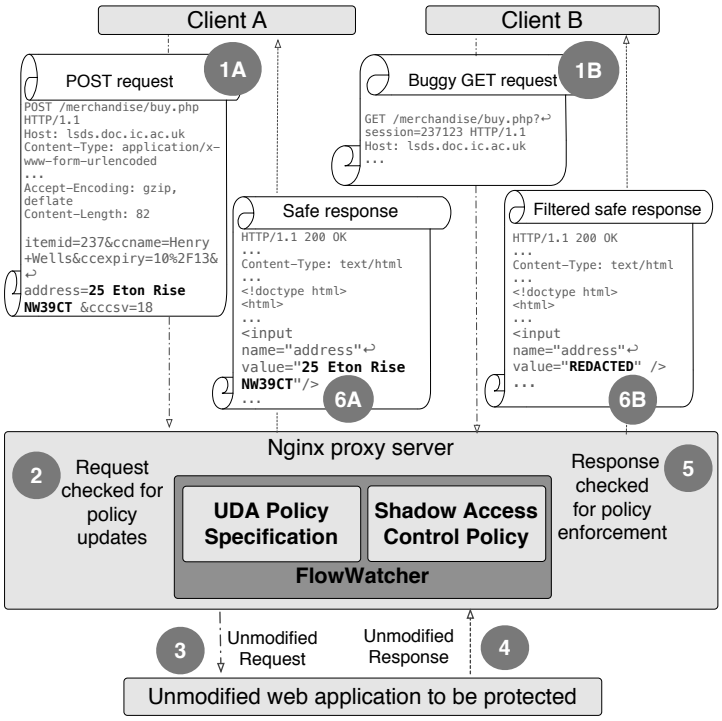

Figure 1: Overview of the FlowWatcher operation

At a high-level, FlowWatcher works as shown in Figure 1. When a client A sends an HTTP POST request (step 1A), FlowWatcher intercepts the request. Based on the UDA policy rules matched by the request parameters, FlowWatcher updates the shadow policy (step 2; described in §4.2). It then forwards the request to the application (step 3) and waits for the response. FlowWatcher intercepts the response (step 4) and consults the shadow policy to determine if the response contains data that should not be seen by the user who made the request (step 5; described in §4.1). If not, the unmodified response is returned to the client (step 6A); otherwise, if another client B sends a request that exposes a data disclosure vulnerability (step 1B), the response is modified to remove these data items (step 6B).

Next we describe how FlowWatcher performs the above two operations, enforcing the shadow policy and updating the shadow policy, in more detail.

4.1 Policy enforcement

To enforce the shadow policy as described by the UDA policy, FlowWatcher represents the current access control state using four data structures (see Figure 2):

- *user_auth* maps users to their authentication tokens;
- *group_members* maps users to their groups;
- *object_acl* maps data objects to users/groups with access; and
- *data_items* maps data objects to the associated data items.

In addition, *static_data* contains a whitelist of data that is part of the application and may be returned in HTTP responses to any user. To eliminate false positive detection of data disclosure, this data should never be tracked as unique user-generated data. The whitelist is initialised with all localised strings, scripts and other static content returned by the web application in HTTP responses.

Before the data in an HTTP response is returned to a user, FlowWatcher determines if the content of the response complies with the shadow policy through the following steps:

(a) *Identify requesting user.* FlowWatcher extracts the authentication token from the associated HTTP request, and looks up the token in *user_auth* to identify the user u_i that made the request.

(b) *Match response content.* FlowWatcher matches the tracked data items from *data_items* against the response header and body. To prevent XSS attacks, all HTML tags are first stripped from the response body. It then records all data objects o_i for which the data items $D(o_i)$ were matched.

(c) *Check access control list.* It looks up each matched data object o_i in the *object_acl* access control list to determine if user u_i is part of the list (or a member of a group in the list by considering the group membership mapping, *group_members*). If multiple data objects refer to the same set of data items, they are not unique and thus the most permissive access control decision is made.

(d) *Consult whitelist.* For matched objects without authorised access, FlowWatcher checks if the associated data items $D(o_i)$ are found in *static_data*. In this case, the data also belongs to the application and the corresponding data object o_i is ignored.

(e) *Redact response.* For all remaining data objects o_i that user u_i is not authorised to view, FlowWatcher redacts the data items $D(o_i)$ from the response, and alerts the system administrator.

4.2 Policy updates

FlowWatcher infers changes to the shadow policy from the observed HTTP request-response sequences. The definition and update rules in the UDA policy (described in §3.2) allow FlowWatcher to maintain the shadow policy for enforcement. As users interact with the application and new data objects and groups are created, the shadow policy is updated correspondingly.

Figure 2 shows how FlowWatcher updates the shadow policy when a user that owns an article gives access rights to a new group. When an HTTP request-response sequence is intercepted (step 1), FlowWatcher first verifies that the response indicates a successful

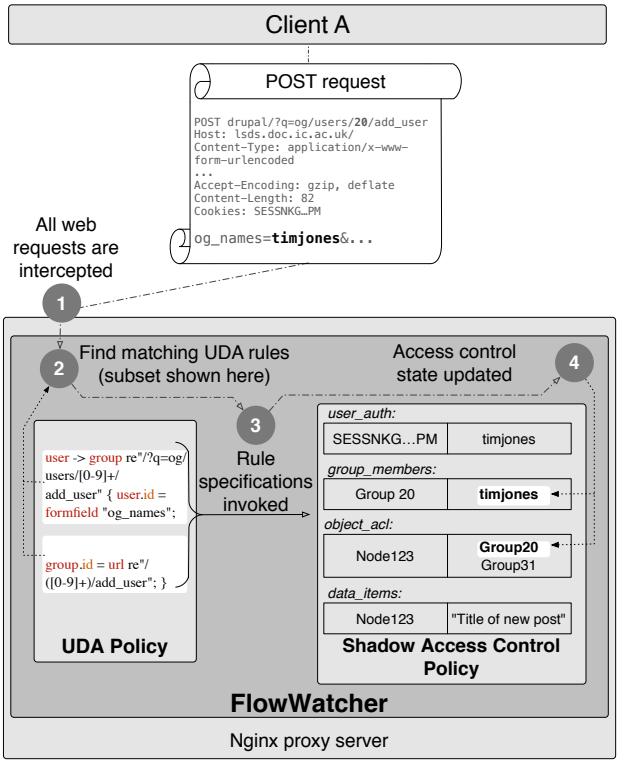

Figure 2: Updates to shadow access control policy

request. It then checks the request URL and request and response parameters and looks up the definition and update rules in the UDA policy that are triggered (step 2). After that, it invokes the rule specification for each matched rule (step 3), updating one or more of its corresponding data structures, i.e. *group_members*, *object_acl* and *data_items* (step 4). For example, if the update rule `user -> group` in Listing 1, line 16 is triggered, as shown in Figure 2, FlowWatcher adds the user `user.id` as a member of the group `group.id` to the *group_members* mapping.

4.3 Data tracking

To detect data disclosure, FlowWatcher relies on the uniqueness of tracked data objects submitted in form fields. Tracking data objects that are not unique, i.e. that were submitted independently by multiple users in HTTP requests, is not useful in identifying data disclosure because the origin of the data is ambiguous. In addition, FlowWatcher must not track user-generated data that is the same as static application data served in HTTP responses because this would lead to false positive detection of data disclosure.

FlowWatcher adopts three strategies to reduce the number of tracked data objects while maintaining uniqueness: (i) when writing a UDA policy, developers should select data items that are likely to be unique (and hence useful to track) due to their semantics; (ii) FlowWatcher only reports a data flow between users if *all* data items, $D(o_i) = \{d_1, d_2, \ldots\}$, that are part of a data object $D(o_1)$ are observed in an HTTP response; and (iii) FlowWatcher only tracks the value of a data item d_i if it has more characters than a global *minimum uniqueness length* parameter α, i.e. $|d_i| > \alpha$. As we evaluate in §5.3, in practice, the above strategies manage to track most data that is unique to each user, while still including a large portion of all user-generated data entered in form fields.

To reduce the amount of data that is stored as part of *data_items*, as an optimisation, FlowWatcher only stores a prefix of each data item of at most β characters, together with the length of the complete item and its hash. The prefix of the data item is used for matching against response content (see §4.1, step b). When content is redacted (step e), the length indicates the amount of data to be redacted and the hash is used to verify the match.

As a result, the size of *data_items* grows linearly with the number of unique data items added to the application, but it is independent of the size of the items. In addition, the data removal rules ensure that FlowWatcher discards removed data.

4.4 Threat analysis

Completeness of policies. The goal of FlowWatcher is to act as an additional line of defence against unauthorised data disclosure. As such, a UDA policy does not need to be complete, i.e. cover the whole access control model of an application. If a UDA policy only refers to a subset of all data objects, however, FlowWatcher may exhibit false negative detections, i.e. it may miss the unauthorised disclosure of data that is not included in the UDA policy. In addition, it may cause false positive detections if tracked user-generated data is not unique. As we show in §5, however, tracked data items are unique with high probability, and it is simple to write UDA policies that cover most data objects of applications.

Correctness of policies. A benefit of UDA policies is that they are application- and not deployment-specific and thus can be written by application developers. A developer can ship their application with a correct UDA policy, which does not require further changes when the application is deployed. The declarative, rule-based nature of the UDA policy language makes it easier to spot mistakes compared to the implementation of the access control model as part of the application itself.

In addition, FlowWatcher effects policy updates only when a request is successful, so that the shadow policy does not diverge from the application's own access control policy. To this end, we assume that all well-developed applications indicate unsuccessful requests for policy changes through an appropriate error message in the response, which can be checked by FlowWatcher.

Vulnerabilities in FlowWatcher. Since FlowWatcher acts as an additional security layer, it cannot disclose data to unauthorised users that is not already part of an HTTP response to be delivered to a user. In the worst case, a bug in the FlowWatcher implementation can lead to false negatives or positives—FlowWatcher's small code base makes bugs less likely.

Correctness of authentication logic. FlowWatcher assumes that the authentication logic of the application is correct. We believe that this is a reasonable assumption in practice: since the authentication logic is implemented typically in a single module of an application (such as the `User` core module in Drupal) and, unlike access control checks, is not part of *all* modules, bugs are less likely to exist. Orthogonal to FlowWatcher, approaches that use dynamic data tracking to track user credentials during authentication, e.g. through a modified interpreter [10], can still be applied.

Denial-of-service attacks. A malicious user could attempt a denial-of-service attack, in which they cause FlowWatcher to redact data incorrectly in a response. If a malicious user creates a new data object that overlaps with the data object of another user, FlowWatcher notices the overlap and applies the most permissive policy, i.e. permitting the original user to access their data without causing a false positive (see §4.1, step c); if they create a new data object with public data, which is e.g. part of the web application, FlowWatcher only generates a false positive if the data is not part of the static data whitelist (see §4.1, step d); finally, if they create a new object that matches user-generated data of another user but that is not

Application	Type	Policy size (LoC)	Rules	Bug	Description
Drupal [16]	Content management	43	7 definition 6 update	CVE-2012-2081 CVE-2013-4596	Exposes private group titles to non-members Enabling extra module exposes unpublished articles
OwnCloud [40]	File sharing	21	4 definition 3 update	CVE-2013-2043 CVE-2014-3834	Any user can download another user's calendar Any user can download another user's contacts
DokuWiki [12]	Wiki	26	2 definition 5 update	CVE-2010-0287 CVE-2009-1960	Directory traversal exposes private filenames File inclusion bug leaks private page text
phpMyAdmin [44]	Database administration	9	2 definition 1 update	CVE-2014-4987	Unprivileged user can see MySQL user list
WordPress [54]	Content management	23	3 definition 4 update	CVE-2010-0682	Trashed posts are exposed to other users
phpBB [43]	Forum management	18	4 definition 2 update	CVE-2010-1627	RSS feeds exposed to unauthorised users
Dropbox [14]	File sharing	14	2 definition 1 update	[15]	Inadvertent sharing of private links leaks data

Table 2: Data disclosure bugs in web applications mitigated by FlowWatcher

tracked by the UDA policy, FlowWatcher also reports a false positive. This can be prevented by having FlowWatcher by default track all user-generated content and check for overlap.

In all cases, a malicious user must be authenticated to mount a denial-of-service attack using FlowWatcher, because only authenticated users can create new tracked data objects. In practice, a system administrator will be alerted of any repeated detection of data disclosure caused by the same user, and can block the user.

5. EVALUATION

We describe our experimental evaluation of a prototype implementation of a FlowWatcher web proxy (§5.1). Our results show that FlowWatcher is (a) *simple* to use—we managed to write UDA policies for a wide range of web applications, including Drupal, OwnCloud, DokuWiki, phpMyAdmin and WordPress, with the longest policy having only 43 lines (§5.2); (b) *effective* in protecting against real-world data disclosure vulnerabilities—we describe a range of previously-reported CVE bugs in the above applications that FlowWatcher mitigates (§5.2) and demonstrate that its data tracking approach is effective in practice (§5.3); and (c) *efficient*—our unoptimised implementation as part of the Nginx reverse proxy does not impact the throughput of a Drupal deployment (§5.4).

5.1 Prototype implementation

We implemented FlowWatcher as an add-on module for the Nginx [35] HTTP reverse proxy. We chose Nginx for its high performance [46] and ease of extension through custom modules. FlowWatcher is designed as an Nginx *filter* that can read HTTP requests and responses and manipulate responses returned to users.

The FlowWatcher implementation consists of 1834 lines of C code, which makes it easy to conduct a security audit. It uses the Redis in-memory store [45] for storing and looking up the shadow access control policy. To filter HTTP responses, it uses streaming regular expression matching over the response bodies.

To avoid re-generating the regular expressions for each request, the implementation caches the last generated regular expression string for each user. This is done in the Redis store because the Nginx module API does not permit maintaining state across requests. A regular expression string is reused until it is invalidated by the creation of a new data object that should not be disclosed to a given user. A limitation of our prototype is that it currently does not handle application transformations of user-generated data items such as HTML entity sanitisation.

5.2 Can FlowWatcher mitigate real-world data disclosure bugs?

To determine if FlowWatcher can mitigate real-world data disclosure, we write UDA policies for 7 popular web applications, specifying their core access control models. All rules were written by us in a day. We then evaluate the policies with bugs from the CVE database [9], which resulted in data disclosure for these applications, observing if FlowWatcher can mitigate the bugs. Table 2 lists the applications, details of the UDA policies and the bugs.

Drupal [16] is a content management framework written in PHP. The standard release of Drupal, known as Drupal core, contains basic features, but there is a growing set of community-contributed add-on modules. In recent years, many security vulnerabilities have been discovered in these add-ons [19]. Natively, Drupal only supports access control permissions for performing operations such as reading, writing and editing different content types (i.e. articles, pages, etc.) based on roles such as "administrator" or "authenticated user". More fine-grained access control between users can be realised using third-party modules.

For the purpose of our evaluation, we focus on access bypass vulnerabilities in two contributed modules. First, the Organic Groups module [18] supports the creation of groups that can restrict access to content. Due to a missing access check (CVE-2012-2081), users were able to access titles of private groups that should only be accessible to group members. Second, the Node Access Keys module [17] grants users temporary access permissions to selected content types based on custom user roles: e.g. registered users for a course can be mailed access keys allowing them to view certain types of content such as course pages. When this module was enabled, unpublished nodes (of any content type without an access key) became visible to all users (CVE-2013-4596).

We discussed the UDA policy rules for Drupal in §3. The complete policy, shown in Appendix A, consists of 43 lines with 8 definition and 7 removal rules, and covers all content types of a default Drupal installation. It specifies that data from different content types must be tracked by the shadow access control policy, and access must be restricted based on group membership, as defined when content is created or updated.

Further content types from third-party modules require extra rules to expose them to FlowWatcher. For example, the policy includes two rules (shown in Listing 1, lines 9 and 25), to enable FlowWatcher to control access to a new content type, Private Groups,

provided by the Organic Groups module. By tracking unpublished content and Private Group data, FlowWatcher can prevent the data disclosure caused by the two above bugs.

OwnCloud [40] is a cloud hosting software that allows users to store files, contacts and calendars, and share these with other users or groups. In versions before 4.5.11 and 5.0.6, the file `apps/calendar/ajax/events.php` lacked a correct ownership check for calendars, enabling users to download any calendar by manually setting the `calendar_id` parameter in the request URL (CVE-2013-2043). In versions before 6.0.3, a missing access check revealed contacts from the address book of one user to others (CVE-2014-3834).

The UDA policy for OwnCloud encodes the group membership policy and the data sharing behaviour with respect to calendar and contact information with 7 rules in 21 lines. Since the rules stipulate that calendars and contacts must only be visible to users and groups that they are shared with, FlowWatcher prevents the data leakage caused by both reported bugs.

DokuWiki [12] is a Wiki application that does not require a database. It has built-in support for user groups, and an access control model that restricts access to Wiki entries (including page titles) for users and groups. In CVE-2010-0287, a directory traversal vulnerability was introduced due to a missing check, allowing any user to view the titles of pages by navigating to a specially-crafted URL. DokuWiki stores user data as files, and if files have insufficient access control, data from one user can flow to another. A file inclusion bug (CVE-2009-1960) allowed unauthorised users to access text of any page. DokuWiki's policy has 7 rules (26 lines), and FlowWatcher prevents the disclosure from both vulnerabilities.

phpMyAdmin [44] is a PHP tool for the administration of databases over the web. It provides a user interface for managing databases, users and permissions and supports user groups. User information is considered private, and users should not be able to see information about other users, unless they are part of the same group. A missing access control check in the application permitted users to see members of all groups (CVE-2014-4987).

We create a UDA policy with 3 rules (9 lines) that models the group membership, while tracking user names and other details, such as passwords. Using this policy, FlowWatcher can identify and prevent unauthorised leaks of user information between users, including the above bug. Note that this does not constitute a complete access control policy for phpMyAdmin—extra rules would be required to control access to other data objects such as databases.

WordPress [54] is a content management system for weblogs and websites. WordPress allows users to create posts and determine who to share them with (i.e. public, private or password-protected). Private posts should only be visible to the creator of the post. Posts that are moved to the trash folder should not be visible to anyone. A bug in WordPress versions prior to 2.9.2 (CVE-2010-0682) allowed any user to view a trashed post by changing the value of the post identifier (`p=`) in the request URL.

The UDA policy for WordPress consists of 8 rules written in 23 lines—we give the complete policy in Appendix A. It expresses access semantics to posts in the trash through a special `Null` group without users. When a post is moved to the trash, FlowWatcher assigns the data object to the `Null` group, which specifies that no other users should be able to view the post. This prevents the data disclosure described in CVE-2010-0682. When a trashed post is restored, it is removed from the `Null` group.

Another feature of WordPress are password-protected posts, which should only be accessible to users with the correct password. In CVE-2014-5337, a missing access check permitted a user to view password-protected posts by navigating to `export/content.php`, without entering the password. FlowWatcher is unable to defend against this bug because the UDA policy cannot express the dynamic change to the access control list after the password was shared out-of-band (i.e. without a request by a user that can be observed by FlowWatcher).

phpBB [43] is a forum application. Forum access can be restricted to specific groups of users. Each forum contains topics, and topics can contain replies. Versions of phpBB prior to 3.0.7-PL1 had a vulnerability caused by an improper permissions check for feeds, enabling any user to access data from a private forum by requesting its RSS feed (CVE-2010-1627).

We model the policy of phpBB with 6 rules (18 lines). A `ForumData` data object has three data items, namely the forum name, topics created under the forum, and replies on a topic. Since the creation of each data item entails a visit to a different URL, the UDA policy has multiple data definition rules for `ForumData` based on the request URL. It also has two rules to update group membership and group access to `ForumData`. By tracking forum data and group access to forums, FlowWatcher can prevent the data leak through RSS feeds.

Dropbox [14] is a popular cloud-based file storage application that we use to illustrate a key advantage of FlowWatcher—its ability to prevent data disclosure for large complex, closed-source applications. In this scenario, we assume that FlowWatcher is deployed by Dropbox as part of its back-end service.

Dropbox has several mechanisms for sharing data, one of which is to create *private links* with which any user can access the associated content. Dropbox allows to restrict access to private links by setting an expiry date. However, inadvertent sharing of a private link can lead to unwanted data disclosure—in particular, a bug was reported whereby any Dropbox document shared using a private link could leak the link through the HTTP referer header if the document itself contained a link to a third-party website [15].

By examining Dropbox's web interface, we create a UDA policy that enforces the expiry of private links. The policy contains 3 rules (14 lines): user and data definition rules for adding a new user and a file, and a group update rule for tracking the creation of a private link to a file. The group update rule uses a special `any` group in combination with an `expiry` statement in the rule body to allow public access to the linked content until the link expiry time is reached.

5.3 How effective is FlowWatcher's tracking?

Next we explore our hypothesis that it is possible to track most of a typical application's user data, with low false positive and negative rates, by only considering data items with a length above the minimum uniqueness length α (see §4.3). For this, we employ a complete HTTP request and response trace from a deployment of GitLab [23], a web application for project and repository management, collected at our university over a period of two months.

Table 3 summarises the properties of the trace. Most of the HTTP POST request URLs are for group wikis and issue tracking, with the remainder involving user profiles and admin pages. The number of form fields in the trace corresponds to the number of potential data objects (assuming a single data item per object); the number of form field values corresponds to the number of potentially trackable data object instances.

We associate each form field in an HTTP POST request with a user session, and consider the percentage of *useful-to-track* form field values, i.e. ones that are only observed in the requests of a single user (or group). A high percentage of useful-to-track values means that tracking them through a UDA policy should lead to few false negatives, i.e. cases in which FlowWatcher misses unautho-

Characteristic	Value
HTTP requests	85,309
HTTP POST requests	461
Distinct POST URLs	59
Users	48
Groups	10
Form fields	363
Total form field values submitted	1271

Table 3: Summary of HTTP trace dataset for GitLab

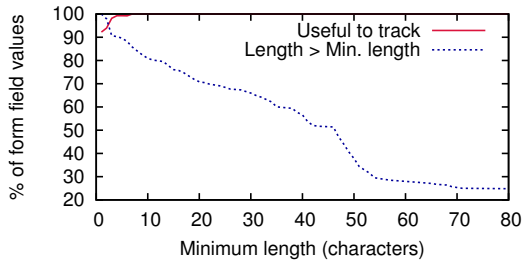

Figure 3: Impact of minimum uniqueness length α on data tracking for form field values submitted to GitLab

rised data disclosure because it did not track the involved user data. We ignore form fields that are part of a *static_data* whitelist for GitLab and cannot be modified by the user.

Figure 3 shows the percentage of useful-to-track form field values as we increase the minimum uniqueness length α. It also shows the percentage of form field values with lengths of more than α. We can see that, as α increases, the form field values become useful-to-track quickly: over 99% of values with lengths greater than 4 characters are specific to a given user or group; all form field values with more than 7 characters are useful-to-track. With $\alpha = 4$, over 90% of all form field values remain trackable by FlowWatcher; even with $\alpha = 7$, over 85% of form field values are included.

This analysis shows that, even when a UDA policy includes all form fields used by GitLab as tracked data items, FlowWatcher can achieve a substantial coverage of form field values for a short minimum uniqueness length α. In addition, since tracked data items quickly become unique, FlowWatcher does not exhibit false positives when it does not track all user-generated data due to an incomplete UDA policy.

The data stored by FlowWatcher is enumerated at the beginning of §4.1. Assuming that the number of groups remains largely unchanged over the application's lifetime, the *user_auth* map and the *group_member* map grow linearly with the number of users of the application. As described at the end of §4.3, the *object_acl* and *data_items* maps, which are conflated into a single data structure for the implementation, grow linearly with the number of unique data items added to the application, but are independent of the size of the data items. For our two-month deployment of GitLab, FlowWatcher requires less than 100 KB of memory.

5.4 What is the performance impact?

We explore the performance impact that FlowWatcher has on request throughput and response latency in a realistic web application deployment. We use Drupal [16] (version 6.31) with an Apache HTTP server and a MySQL database, which are both deployed on a server with an Intel Xeon E5-2690 CPU with 32 GB of RAM. FlowWatcher runs with Nginx as a reverse proxy, forwarding requests and responses from and to clients. The Nginx proxy executes on a machine with an Intel Xeon E5-4620 with 64 GB of

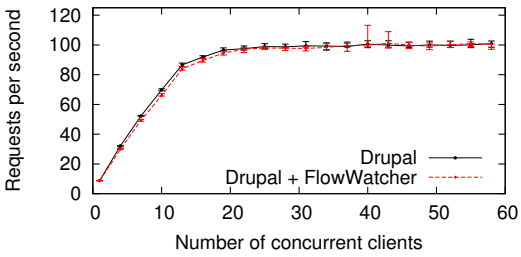

Figure 4: Request throughput with and without FlowWatcher under an increasing client workload

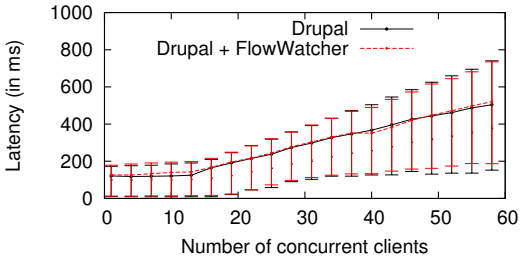

Figure 5: Latency with and without FlowWatcher under an increasing client workload

RAM. FlowWatcher uses the Drupal UDA policy from Listing 1, tracking article titles and bodies for access control enforcement.

We preload the Drupal installation with 6000 users and 60 user groups, with each user assigned to one group. For a realistic workload, we use Apache JMeter [1] to generate HTTP requests based on the following workflow: each client (1) visits the main page and goes to the log-in page; (2) logs in and is redirected to the main page with a list of recent posts; (3) accesses the "Add post" page and creates a new post; (4) visits the new post; and (5) logs out.

We evaluate the performance of FlowWatcher with 1 to 60 concurrent clients. Each time a client performs the above workflow, it logs in as a different user. A fraction of the posts (5%) are assigned to the access-restricted group of the user making that request. This is below the read/write ratio of the GitLab trace described in §5.3, reflecting the nature of real-world deployments. Access-restricting some of the posts ensures that FlowWatcher must redact the response to a subset of all read requests. As for a production environment, we enable both PHP intermediate code caching and Drupal's page cache for improved performance. We execute five runs for two experiments—one with FlowWatcher and one without. Throughout the experiments, the CPU utilisation on the proxy machine never exceeds 10%.

Figure 4 shows the measured throughput with and without FlowWatcher. As can be seen, FlowWatcher has no discernible impact on throughput. In both deployments, the throughput flattens out after 20 concurrent clients when the web server becomes saturated. As all responses contain dynamic content, the reported throughput represents a worst case scenario for Drupal because it cannot serve cached content. Figure 5 shows the average response latency with an increasing client workload. Even with 60 clients, there is no statistically significant latency increase due to FlowWatcher. The high latency variance for both deployments is due to the different costs of requests in the executed workflow.

6. RELATED WORK

In addition to the related approaches discussed in §2.2, we discuss further related work, contrasting with FlowWatcher in terms of scope and effect.

Data flow tracking enforces access control throughout an application by associating access control state with data flows [10, 55]. CloudFence [41] uses binary rewriting and byte-level taint tracking to give data flow guarantees. SilverLine [33] provides even stronger containment using Information Flow Control to guarantee isolation of both data and control flows. It propagates taint information into the application database, but cannot effectively support valid information flows between users. However, the significant performance overheads and tight language binding limit the broad adoption of data flow tracking approaches. In contrast, FlowWatcher protects unmodified applications with low overhead but, as a black-box approach, it does not detect data disclosure when the application modifies the data.

Web application firewalls (WAFs) monitor and potentially block data passed to/from web applications. ModSecurity [31] acts as a reverse proxy and has rules to detect certain classes of data, such as credit card number patterns. However, this approach is prone to false positives, and, unlike FlowWatcher, cannot detect leakage of specific user data. WAFs also support other types of data disclosure protection [27], such as traffic anomaly detection, and HTTP request sanitising techniques to prevent injection attacks. We regard this as complementary to FlowWatcher, which could be integrated with existing proxy-based WAFs.

Secure web frameworks aim to offer comprehensive enforcement of access control policies. They are typically tied to a particular development framework, such as the model-policy-view-controller of the Hails architecture [22]. The Passe framework [5] executes isolated server processes in sandboxes, ensuring that each web page view runs with the minimum required privileges. Here again, a learning phase is needed to identify the associations between views, database queries, and data flow. Similarly, GuardRails [6] modifies an annotated Ruby-on-Rails web application, to add secondary access control checks on sensitive data. Unlike FlowWatcher, all of these approaches require specifically written or annotated application code within a given framework.

Policy languages. A large number of policy languages exist for different access control models [36, 11]. Existing policy languages focus on expressiveness, e.g. through role-based access control models [47], or on the needs of specific domains, such as data privacy. For example, to restrict data dissemination in online social networks, the UURAC model [7] allows individual users, and the social network itself, to specify which users should be allowed to access which data. In contrast, the goal of our UDA policy language is to express a dynamic access control model with access control lists in terms of observed HTTP requests and responses.

7. CONCLUSIONS

As web applications support richer functionality, it becomes challenging for developers to ensure that all application components perform access control checks correctly. The rise of data disclosure vulnerabilities in web applications shows that existing techniques to detect and mitigate bugs in the authorisation logic of applications lack widespread adoption.

We described FlowWatcher, a practical approach to mitigate unauthorised data disclosure in web applications due to bugs in the authorization logic. FlowWatcher operates externally to the application: developers specify the intended dynamic access control policy in a rule-based policy language, as a UDA policy, which enables the FlowWatcher web proxy to detect unauthorised data disclosure by tracking the propagation of data between HTTP requests and responses across users. As our experimental evaluation demonstrates, UDA policies are simple to write, can protect against a wide range of data disclosure bugs and can be enforced with low overhead.

Acknowledgements

This work was supported by grant EP/K008129/1 (“CloudSafetyNet: End-to-End Application Security in the Cloud”) from the UK Engineering and Physical Sciences Research Council (EPSRC).

8. REFERENCES

[1] Apache JMeter. jmeter.apache.org, 2014.
[2] D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna. Multi-module Vulnerability Analysis of Web-based Applications. In *CCS*, 2007.
[3] P. Bisht, T. Hinrichs, N. Skrupsky, et al. NoTamper: Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Applications. In *CCS*, 2010.
[4] P. Bisht, T. Hinrichs, N. Skrupsky, et al. WAPTEC: Whitebox Analysis of Web Applications for Parameter Tampering Exploit Construction. In *CCS*, 2011.
[5] A. Blankstein and M. J. Freedman. Automating isolation and least privilege in web services. In *IEEE S&P*, 2014.
[6] J. Burket, P. Mutchler, M. Weaver, M. Zaveri, and D. Evans. GuardRails: A data-centric web application security framework. In *USENIX WebApps*, 2011.
[7] Y. Cheng, J. Park, and R. S. Sandhu. A user-to-user relationship-based access control model for online social networks. In *DBSec*, 2012.
[8] 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.
[9] CVE. cve.mitre.org, 2014.
[10] M. Dalton, C. Kozyrakis, and N. Zeldovich. Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications. In *USENIX Security*, 2009.
[11] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Spec. Language. In *Workshop on Policies for Distributed Systems and Networks (POLICY)*, 2001.
[12] DokuWiki. www.dokuwiki.org, 2014.
[13] A. Doupé, B. Boe, C. Kruegel, and G. Vigna. Fear the EAR: Discovering and Mitigating Execution After Redirect Vulnerabilities. In *CCS*, 2011.
[14] Dropbox. www.dropbox.com, 2015.
[15] The Dropbox Blog: Web vulnerability affecting shared links. bit.ly/1FiRIsv, 2014.
[16] Drupal. www.drupal.org, 2014.
[17] Drupal Node Access Keys. drupal.org/project/nodeaccesskeys, 2014.
[18] Drupal Organic Groups. drupal.org/project/og, 2014.
[19] Drupal Security Advisories. drupal.org/security/contrib, 2014.
[20] Evernote. www.evernote.com, 2014.
[21] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *USENIX Security*, 2010.
[22] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo. Hails: Protecting Data Privacy in Untrusted Web Applications. In *OSDI*, 2012.
[23] GitLab. about.gitlab.com, 2014.
[24] K. L. Ingham, A. Somayaji, J. Burge, and S. F. A. C. Learning DFA Representations of HTTP for Protecting Web Applications. *Computer Networks*, 51, 2007.
[25] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *IEEE S&P*, 2006.
[26] C. Kruegel and G. Vigna. Anomaly Detection of Web-based Attacks. In *CCS*, 2003.

[27] T. Krueger, C. Gehl, K. Rieck, and P. Laskov. TokDoc: A Self-healing Web Application Firewall. In *ACM SAC*, 2010.
[28] M. S. Lam, M. Martin, B. Livshits, and J. Whaley. Securing Web Applications with Static and Dynamic Information Flow Tracking. In *ACM SIGPLAN Symp. on Partial Eval. and Semantics-based Program Manipulation*, 2008.
[29] X. Li and Y. Xue. BLOCK: A Black-box Approach for Detection of State Violation Attacks Towards Web Applications. In *ACSAC*, 2011.
[30] X. Li, W. Yan, and Y. Xue. SENTINEL: Securing Database from Logic Flaws in Web Applications. In *ACM Conf. on Data and App. Security and Privacy (CODASPY)*, 2012.
[31] ModSecurity. www.modsecurity.org, 2014.
[32] M. Monshizadeh, P. Naldurg, and V. N. Venkatakrishnan. MACE: Detecting Privilege Escalation Vulnerabilities in Web Applications. In *CCS*, 2014.
[33] Y. Mundada, A. Ramachandran, and N. Feamster. SilverLine: data and network isolation for cloud services. In *HotCloud*, 2011.
[34] New York Times. Russian Hackers Amass Over a Billion Internet Passwords. nyti.ms/1mjPhsL, 2014.
[35] Nginx. www.nginx.org, 2014.
[36] OASIS eXtensible Access Control Markup Language. bit.ly/1BRwfFt, 2013.
[37] Open Web App. Sec. Proj. (OWASP). Injection flaws. www.owasp.org/index.php/Top_10_2013-A1-Injection, 2013.
[38] Open Web Application Security Project (OWASP). OWASP Top 10 2013. www.owasp.org/index.php/Top_10_2013-Top_10, 2013.
[39] Open Web Application Security Project (OWASP). SQL Injection Prevention Cheat Sheet. www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet, 2014.
[40] OwnCloud. www.owncloud.org, 2014.
[41] V. Pappas, V. P. Kemerlis, A. Zavou, et al. CloudFence: Data flow tracking as a cloud service. In *Research in Attacks, Intrusions, and Defenses*. Springer, 2013.
[42] G. Pellegrino and D. Balzarotti. Toward Black-box Detection of Logic Flaws in Web Applications. In *NDSS*, 2014.
[43] phpBB. www.phpbb.com, 2014.
[44] phpMyAdmin. www.phpmyadmin.net, 2014.
[45] Redis. www.redis.io, 2014.
[46] W. Reese. Nginx: The High-performance Web Server and Reverse Proxy. *Linux Journal*, Sept. 2008.
[47] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2), Feb. 1996.
[48] 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 *IEEE S&P*, 2010.
[49] R. Sekar. An efficient black-box technique for defeating web application attacks. In *NDSS*, 2009.
[50] S. Son, K. S. McKinley, and V. Shmatikov. RoleCast: Finding Missing Security Checks when You Do Not Know What Checks Are. In *OOPSLA*, 2011.
[51] S. Son, K. S. Mckinley, and V. Shmatikov. Fix Me Up: Repairing access-control bugs in web apps. In *NDSS*, 2013.
[52] G. Wassermann and Z. Su. Static Detection of Cross-site Scripting Vulnerabilities. In *ICSE*, 2008.
[53] WhiteHat Security. Website security statistics report. www.whitehatsec.com/resource/stats.html, 2014.
[54] WordPress. www.wordpress.com, 2014.
[55] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving Application Security with Data Flow Assertions. In *SOSP*, 2009.

A. COMPLETE UDA POLICIES

We show the complete listing of the UDA policies for Drupal 6.31 in Listing 2 and for WordPress 4.2.2 in Listing 3.

Listing 2: The complete UDA policy for Drupal

```
/* Definition rules */
user+ "/*" if (res_hdr "Set-Cookie" re"SESS.*")
{ id := formfield "name", res_hdr "Location" re"/?q=user/([0-9]+)";
  token := res_hdr "Set-Cookie" re"SESS.*"; }

group+ "/?q=node/add/group"
{ id := res_hdr "Location" re"/?q=node/([0-9]+)"; }

data+ Article re"/?q=node/add/article"
{ id := res_hdr "Location" re"/?q=node/([0-9]+)";
  item := formfield "title", formfield "body"; }

data+ Page re"/?q=node/add/page"
{ id := res_hdr "Location" re"/?q=node/([0-9]+)";
  item := formfield "title", formfield "body"; }

data+ PrivateGroupName "/?q=node/add/group" if (formfield
      ↪ "og_private"="1")
{ id := res_hdr "Location" re"/?q=node/([0-9]+)";
  item := formfield "title"; }

group- re"/?q=node/[0-9]+/delete" if (formfield "op"="Delete+group")
{ group.id := url re"/?q=node/([0-9]+)"; }

user- re"/user/[0-9]+/delete"
{ id := url re"/user/([0-9]+)"; }

data- Any re"/?q=node/[0-9]+/delete"
{ id := url re"/?q=node/([0-9]+)"; }

/* Update rules */

data* Article re"/?q=node/[0-9]+/edit"
{ id = url re"/([0-9]+)/edit";
  item[0] = formfield "title";
  item[1] = formfield "body"; }

data* Page re"/?q=node/[0-9]+/edit"
{ id = url re"/([0-9]+)/edit";
  item[0] = formfield "title";
  item[1] = formfield "body"; }

user -> group re"/?q=og/users/[0-9]+/add_user"
{ user.id = formfield "og_names";
  group.id = url re"/([0-9]+)/add_user"; }

user -/> group re"/?q=og/unsubscribe/[0-9]+/[0-9]+"
{ group.id = url re"/([0-9]+/)/";
  user.id = url re"[(0-9]+)(?!\/)"; }

user -> data re"/?q=node/add/.*" if (formfield "status"="0")
{ user.id = authenticated_user;
  data.id = res_hdr "Location" re"/?q=node/([0-9]+)"; }

group -> data re"/?q=node/add/*" if (formfield "status"="1")
{ group.id = formfield "og_groups";
  data.id = res_hdr "Location" re"/?q=node/([0-9]+)"; }

group -> PrivateGroupName "/?q=node/add/group" if (formfield
      ↪ "og_private"="1")
{ group.id = res_hdr "Location" re"/?q=node/([0-9]+)";
  PrivateGroupName.id = res_hdr "Location" re"/?q=node/([0-9]+)"; }
```

Listing 3: The complete UDA policy for WordPress

```
/* Definition rules */
user+ "/wp-login.php" if (res_hdr "Set-Cookie"
      ↪ re"wordpress_logged_in_.*")
{ id := res_body re/"\"uid\":\"([0-9]+)\"";
  token := res_hdr "Set-Cookie" re"wordpress_logged_in_.*"; }

data+ Post re"/wp-admin/post\.php" if (formfield
      ↪ "visibility"="private")
{ id := res_hdr "Location" re"/?post=([0-9]+)";
  item := formfield "post_title", formfield "content"; }

user- re"wp-admin/users.php.*action=delete" if (formfield
      ↪ "action"="dodelete" and res_status="302")
{ id := url re"user=([0-9]+)"; }

data- re"/wp-admin/post\.php.*action=delete" if (res_status="302")
{ data.id = res_hdr "Location" re"/?post=([0-9]+)"; }

/* Update rules */

data* Post re"/wp-admin/post\.php.*action=edit"
{ id := res_hdr "Location" re"/?post=([0-9]+)";
  item[0] = formfield "post_title";
  item[1] = formfield "content"; }

user -> data re"/wp-admin/post\.php" if (formfield
      ↪ "visibility"="private")
{ user.id = authenticated_user;
  data.id = res_hdr "Location" re"/?post=([0-9]+)"; }

group -> data re"/wp-admin/post\.php.*action=trash" if
      ↪ (res_status="302")
{ group.id = Null;
  data.id = res_hdr "Location" re"/?post=([0-9]+)"; }
```

```
group -/> data re"/wp-admin/post\.php.*action=untrash" if
      ↪ (res_status="302")
{  group.id = Null;
   data.id = res_hdr "Location" re"/?post=([0-9]+)"; }
```