SafeWeb: A Middleware for Securing Ruby-based Web Applications

Petr Hosek[1], Matteo Migliavacca[1], Ioannis Papagiannis[1], David M. Eyers[2], David Evans[3], Brian Shand[4], Jean Bacon[3], and Peter Pietzuch[1]

[1] Imperial College London, {ph1310,migliava, ip108, prp}@doc.ic.ac.uk
[2] University of Otago, dme@cs.otago.ac.nz
[3] University of Cambridge, {*firstname.lastname*}@cl.cam.ac.uk
[4] ECRIC, National Health Service, brian.shand@cbcu.nhs.uk

Abstract. Web applications in many domains such as healthcare and finance must process sensitive data, while complying with legal policies regarding the release of different classes of data to different parties. Currently, software bugs may lead to irreversible disclosure of confidential data in multi-tier web applications. An open challenge is how developers can guarantee these web applications only ever release sensitive data to authorised users without costly, recurring security audits.

Our solution is to provide a trusted middleware that acts as a "safety net" to event-based web applications by preventing harmful data disclosure before it happens. We describe the design and implementation of SafeWeb, a Ruby-based middleware that associates data with security labels and transparently tracks their propagation at different granularities across a multi-tier web architecture with storage and complex event processing. For efficiency and ease-of-use, SafeWeb exploits the dynamic features of the Ruby programming language to achieve a low performance overhead and require few code changes in ① applications. We evaluate SafeWeb by reporting our experience of implementing a web-based cancer treatment application and deploying it as part of the UK National Health Service (NHS).

1 Introduction

Web applications in areas such as healthcare, financial processing and government services must selectively expose sensitive data to authorised sets of web users. For example, a cancer researcher may want to query a centralised patient database over the web for anonymised health records of patients that have a given type of cancer. The costs of inadvertently disclosing confidential data to the wrong users due to implementation errors in web applications are high—hospitals and medical practitioners in the UK are legally liable for unauthorised disclosure of patient data without prior consent. Due to new legislation introduced in 2010, organisations can be fined up to £500,000 for security breaches [10]. In this paper, we address *how to implement secure web applications that are guaranteed to comply with data protection policy.*

Enforcing a data protection policy *end-to-end*, i.e. across an entire multi-tier web application, is challenging. An implementation error in any tier of a web application may result in unauthorised data disclosure. Developers may introduce software bugs inadvertently, or based on misunderstandings of requirements. Achieving correctness is even more challenging for web applications that process different types of data from multiple domains, such as hospitals, laboratories and insurance providers, each with their own security requirements.

Current best practices include manual source code auditing for new applications, which is error-prone and costly. Tools for static analysis such as Pixy [11] or symbolic execution such as KLEE [2] validate that the implementation satisfies given invariants. Their use, however, requires expert knowledge to formalise invariants and they cannot handle large heterogeneous distributed web applications due to these applications' size and complexity.

We make two assumptions: (1) In terms of the threat model, we assume that the external environment is hostile, but that application code is not explicitly malicious, even if threats might be caused by bugs in the implementation. (2) We assume that stakeholders are willing to accept some performance overhead—in terms of request throughput and latency—for increased security.

Our solution is to propose a middleware that implements a "safety net" by providing a data-centric security approach that integrates well with multi-tier web applications. Our middleware is based on two key ideas: It decouples the processing of confidential data from the handling of web requests. In addition, it tracks data as it flows through the web application in order to ensure its confidentiality and integrity. This means that implementation bugs in the web request handling logic cannot cause any unauthorised confidential data to be disclosed. By tracking data propagation by means of security labels, the middleware performs automatic and appropriate compliance checks at the boundaries between application components, without relying on developer support. This reduces the effort required for security audits.

We demonstrate the practicality of this approach by describing SafeWeb, a Ruby-based middleware that enforces data flow policy across web applications. SafeWeb consists of an *event-processing backend* that processes data asynchronously from a confidential data store according to application-specific business logic. Events are associated with security labels, which are tracked by SafeWeb as they propagate between event processing components. This mechanism is implemented efficiently through isolation of processing components using Ruby's safe levels. A separate *web frontend* serves processed event data in response to web requests, while maintaining security labels at the variable level using Ruby's meta-programming features. Before sending data to web users, SafeWeb validates the associated security labels against user privileges, thus preventing violations of the data protection policy.

SafeWeb is designed to integrate well with existing security practices at an organisational, architectural and infrastructure level. From an organisational viewpoint we complement existing security practices, such as network partitioning through firewalls and security code auditing, without requiring signif-

icant changes. At the infrastructure level, SafeWeb can easily be applied to production environments. In contrast with classical label-based approaches, we avoid complex changes in the language runtime. We are able to provide tracking purely at the middleware level, through careful exploitation of Ruby's meta-programming and security features.

We evaluate SafeWeb in a real-world healthcare environment by developing *MDT web portal*, a web application that provides information about ongoing cancer treatment of patients to teams of medical practitioners at hospitals. We discuss the deployment of this application using SafeWeb as part of ECRIC—an organisation in the UK National Health Service (NHS) that collects relevant patient-sensitive oncological data. We show that SafeWeb can guarantee that medical records are only exposed to authorised users—even with implementation bugs in the processing logic of the MDT web portal application. It integrates well with existing information systems and introduces only minimal overheads in terms of application development effort and performance.

In summary, the main contributions of the paper are: (1) a middleware for securing web applications that uses event processing to decouple queries from sensitive data; (2) an application of information flow control techniques across all tiers of a web application to prevent non-compliant disclosure of sensitive data to users and an efficient implementation of data tracking using Ruby language mechanisms; (3) an evaluation of this approach in a healthcare environment using a realistic web application for supporting cancer treatment at hospitals.

In §2 we provide background on the security requirements of web applications dealing with confidential data. Then, we present a general data-centric security mechanism addressing these requirements, explaining why existing technologies cannot be applied in production environments (§3). The SafeWeb middleware is described in §4, focusing on the different components of its architecture. We evaluate our approach in §5 through an implementation of a web application using SafeWeb, describing its security properties and providing a performance analysis. In §6, we discuss related work, and we draw conclusions in §7.

2 Data Confidentiality in Web Applications

Organisations in the public and private sector collect, process and analyse personal data to improve the quality of their services that they offer. Due to the sensitivity of personal data, maintaining its confidentiality is crucial. As a consequence, it is necessary to verify that applications are not vulnerable to compromise from external attacks and that confidential data is handled in compliance with the policies set by organisations.

Current best practices to maintain data confidentiality in applications are costly, error-prone and time consuming: organisations adopt a series of security measures including risk assessments, internal security code reviews and external security consultations. These measures are intertwined with project development to the point that development of new services is limited. For example, health-

care organisations struggle to develop new medical applications that have the potential to improve patient care quickly and cost-effectively.

Middleware can be used to reduce the cost of development and deployment of new applications by moving security auditing effort from applications to reusable middleware components. In this paper, *we describe a middleware that can increase the trust placed in applications that process and provide access to confidential data by placing applications within a "safety net" that, within the constraints of a production environment, protects data from compromise and accidental disclosure.* The goal is to satisfy the following two security requirements—both of which are discussed in the context of a healthcare example in the next section:

S1 Access to confidential data by external users should be static and one-way; it should not be possible from the outside to change which confidential data items are exported from an internal network to the public Internet, or to alter existing data stored within the internal network.

S2 Confidential data should be protected end-to-end; implementation errors in an application should not result in the disclosure of confidential data and violations of the specific security policy for that application.

2.1 Case Study: A Cancer Registration System

In this work, we consider the following real-world case study. The Eastern Cancer Registry and Information Centre (ECRIC) is part of the UK National Health Service (NHS). ECRIC aims to produce a comprehensive picture of cancer cases in the East of England. It receives patient data from many sources including so-called Multidisciplinary Teams (MDTs), hospital Patient Administration Systems (PAS), pathology laboratories and the Office of National Statistics (ONS).

ECRIC's software infrastructure has recently been chosen as the national cancer registry platform for England. Most of its software systems are implemented in the Ruby programming language for ease of development and due to existing developer expertise. The main cancer registration database, hosted in a secure private network, holds structured information about patients, tumours, and associated treatments. Data are imported into the main database from different sources and processed with the help of the domain knowledge of staff. ECRIC staff can also operate off-site by using an external web application server that has been extensively audited for security.

Based on discussions with ECRIC, we identified a new application that they would like to offer: an MDT web portal that provides relevant data that can support the operation of MDTs at hospitals. MDTs treating oncological patients currently provide reports to ECRIC about their patients through secure email and paper forms. ECRIC wants to feed back both summary and detailed patient information to MDTs, letting them benchmark their data quality against their peers and explore the underlying data to discover the cause of any discrepancies. At present, resolving discrepancies is laborious, because staff at ECRIC have to manually extract and release the relevant records for each MDT. In summary, the MDT web portal should satisfy the following functional requirements:

F1 Doctors and MDT co-ordinators that are members of an MDT can log into the MDT portal using a web browser and consult the details of patients treated by that MDT, with the option of providing feedback (handled externally, e.g. via secure NHS email).

F2 Doctors and MDT co-ordinators can consult various metrics about their patients, e.g. the level of completeness of the provided information or projected survival statistics of patients under treatment.

F3 MDT co-ordinators can put those metrics into context by comparing them with each MDT's average in the same region or with regional aggregates.

The *security policy* for the MDT web portal is as follows:

P1 Details about patients can be consulted only by members of the MDT that treats them. MDT-level aggregates can be consulted by all MDTs in the same region. Regional-level aggregates can be seen by all MDTs.

A design of the MDT web portal as a standard web application using the main ECRIC database would not be acceptable. The MDT web portal requires interactive access to patient-level data, in violation of security requirement *S1*—an implementation bug could compromise the integrity and the confidentiality of the whole ECRIC database. Furthermore, errors in the MDT web portal could violate its security policy, conflicting with *S2*.

Enforcing the MDT security policy *P1* with a traditional web application architecture is challenging. The MDT web portal has a data flow path for confidential data that involves multiple components at different layers: data must be extracted from the ECRIC database, processed in an application-specific way, and finally presented by a web front-end. Any component in the layers involved could cause unauthorised data disclosure. The security policy is difficult to enforce through a composition of local mechanisms because it is an *end-to-end* property. The large amount of source code that needs to be trusted increases the risk of defects and incurs a high code review effort.

In addition, the mechanisms used to protect from policy violations must operate at a fine data granularity. For example, the application must distinguish between confidential information at the level of single patients treated by an MDT and access to aggregated data at MDT or regional level.

In summary, we need a security mechanism that (a) is able to enforce end-to-end data flow guarantees, reducing the amount of trusted source code, and (b) allows for fine-grained, data-centric protection of confidential information. In the next section, we describe how controlling the propagation of security-labelled data through the application, at the middleware level, can achieve this.

3 Controlling Data Propagation

Traditional access control models achieve security by restricting *access to resources*: a principal, such as a user who owns data can delegate to other principals a subset of operations. In such a model, it is easy to control information release

but difficult to control its propagation. Once a user has delegated the privilege to read data to another user, information cannot be protected anymore—e.g. the second user can write the data to a public Internet site. Thus, under a traditional (discretionary) access control model, secure data processing means that processors must be trusted—a data processor authorised to read confidential patient data must not disclose it to unauthorised parties.

The problem of unauthorised data disclosure is addressed by *information flow control* (IFC), a mandatory access control model originally developed for military systems [1, 6]. IFC protects the *propagation* of data. We can model an IFC system as a set of *inputs*, *outputs* and *processing components*. An input component, acting on behalf of a principal, can attach a tamper-resistant *security label* to the data—e.g. a label can be used to protect the confidentiality of a patient medical report. The security labels can then be used to track the propagation of data through the system and middleware can enforce end-to-end restrictions on the permitted data flow. For example, if a component producing patient records labels every record, labels can prevent a mailing component from including records in emails, independently of the processing.

IFC systems can guarantee that security labels are preserved by controlling all data flow paths between components. When labelled data is copied or transformed by a component, the IFC system maintains the labels. When labelled data is processed or combined with data with different labels, the resulting labels are a composition of the previous labels, i.e. the system preserves all the data flow restrictions of the original labels. To achieve this, IFC systems require components to be "sandboxed," i.e. isolated from one another and from the external environment. Components can only communicate through primitives that are under the control of the IFC system.

To output data protected by a label, a component must have a *declassification* privilege [12]. This enables the component to remove the label from the data and use these data without restriction. The original owner of the data can restrict the data flow of, for example, a patient record by assigning declassification privileges only to components acting on behalf of treating doctors.

Labels can also be used to protect data *integrity*, which is the dual of confidentiality. The creator of an integrity label delegates to other components an *endorsement* privilege to add this label to data. Components can then trust only data that is "guaranteed" by this integrity label.

3.1 Applying Information Flow Control

IFC is a good fit for developing secure web applications because it can detect and contain the effect of application bugs which could otherwise violate a security policy (cf. security requirement *S2*). Note that we do assume that application code is not intentionally malicious; this problem can be tackled by organisational safeguards such as only allowing trusted developers to develop applications. Instead, we focus on protection from unintentional software bugs.

We describe how IFC achieves our security goals in the context of the MDT web portal application from §2.1. Consider the security policy *P1* for the MDT

application. After creating a label for each patient, a component can (i) attach the label to each patient's data as it enters the system and (ii) assign the declassification privilege over the label only to components that execute on behalf of MDT principals treating the patient. Based on IFC enforcement, this guarantees that each patient record can only be accessed by the correct MDT, independently of the processing that happens between these two endpoints.

In the more complex case of MDT-level aggregate measures, which should be visible only to MDTs in the same region, label tracking is overly conservative because aggregates are considered as sensitive as the data of all involved patients, preventing access by any MDT. Therefore a component trusted with patient data must (i) remove all patient labels from aggregate data, (ii) relabel the aggregate data with an MDT-specific label and (iii) assign a declassification privilege over the MDT label to all MDTs in the region. The same mechanism can be used for regional-level aggregates by defining regional-level labels.

In summary, the security policy of the MDT application can be enforced by applying these three kinds of labels with corresponding privileges. Any component that is not policy compliant due to implementation errors cannot violate the MDT security policy. For example, a component for computing statistical aggregates would be constrained in terms of the data that it can disclose publicly. Even if its implementation is too large and complex to be audited, a software bug in, say, its logging function, which might otherwise reveal confidential patient data in externally accessible log files, would be prevented by IFC enforcement.

3.2 Practical Information Flow Control for Web Applications

In practice, applying a strong security model such as IFC to web applications is challenging. For IFC to be used, it must have low impact on developers, and thus integrate with familiar architectures, programming models and languages.

Recently, researchers have proposed IFC techniques to improve the security of applications. Jif [13] extends the Java type system to include labels and checks them statically. DIFCA-J [31] rewrites Java bytecode to propagate labels in JVM operations. Trishul [14] and Laminar [20] modify the JVM to track labels. DEFCon [12] isolates threads allowing communication only through labelled data.

All of these systems adopt a "strict" tracking model that is invasive on the programmer and the system architecture. IFC tracking in these systems strives to avoid false negatives, as a single false negative could compromise the security of the whole platform if exploited. However, this strict application of IFC leads to false positives that require applications to be restructured to remove ambiguity in data flow tracking. Strict IFC also requires complex implementations: mature, industrial-strength implementations of IFC systems are currently lacking. Adoption of research prototypes in a production environment is undesirable because they are difficult to verify, requiring expert knowledge of JVMs, runtime libraries and bytecode rewriting techniques. Maintenance is also problematic when new versions are released by upstream developers.

In contrast, our IFC tracking approach is inspired by Resin [30], which only targets source code that does not actively try to evade data tracking. It is thus

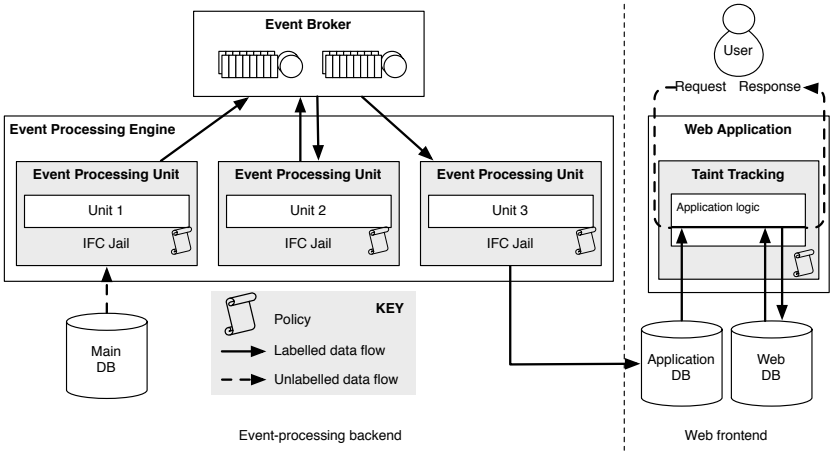

Fig. 1. Overview of the SafeWeb middleware architecture

possible to reduce the number of false positives, accepting a few false negatives, which are unlikely to be exercised by non-malicious code. IFC is thus used as a safety net to catch unintended implementation bugs, instead of acting as the primary method of security policy enforcement. This approach integrates well with current security practices adopted by organisations.

Our work differs from Resin in two key aspects. (1) We target enterprise web applications instead of stand-alone applications built directly on a specific database. In enterprise settings, it is important to support messaging services in the backend with an appropriate information processing model. To the best of our knowledge, we are the first to integrate IFC with two different processing models at different granularities (cf. §4)—a proactive event-based model in the backend with data tracking at the event level and a reactive web language model in the frontend with tracking at the variable level. (2) We manage to support IFC in the middleware without significant changes to the language runtime. By using the Ruby language support for meta-programming and its powerful security primitives, we avoid changes to the Ruby interpreter, leading to an IFC implementation that is easy to understand, verify and maintain.

4 SafeWeb Middleware Design

In this section, we describe SafeWeb, a Ruby-based middleware that separates the processing of confidential data from presentation aspects, while enforcing data flow constraints throughout the application. As shown in Figure 1, SafeWeb consists of two parts: an *event processing backend* (left), which realises the application logic, and a *web frontend* (right), which handles users' requests based on processing results. Application logic in SafeWeb is implemented in an event-based fashion through one or more processing units, which produce and/or consume events. This architecture largely decouples the processing of confidential

data from the handling of web requests [29] and creates a unidirectional data flow from the backend to the frontend, in compliance with security requirement *S1*.

The event processing backend hosts the application logic for processing confidential data. *Events* are created from confidential data retrieved from a data source (illustrated as the *Main Database* in the figure) and labelled appropriately. *Event Processing Units* act as generators, filters or processors of events and exchange labelled events through an IFC-aware *Event Broker*. Units are constrained in their operation by the *Event Processing Engine*. Its *IFC Jail* controls communication of units with the environment and preserves labels during event communication. *Privileges* for units over specific labels are configured through a data flow *Policy*. Result events are stored with appropriate labels in an *Application Database* after processing.

The web frontend serves synchronous web requests from users by accessing the application database. State that is specific to a given web session is stored in a separate *Web Database* to isolate it from application data. Labels from the application database are propagated in the web application by SafeWeb's runtime *Taint Tracking* library and are checked when generating responses. As a result, security labels are associated with data throughout the processing pipeline and checked at boundaries between components with respect to the application policy, thus satisfying requirement *S2*.

4.1 Events with Security Labels

Event processing units communicate through events. Data models for events can vary widely [7]. For ease-of-use, we adopt one of the simplest yet popular choices: events in SafeWeb consist of a set of key-value attribute pairs and an optional data payload. The keys, values and the body are untyped strings.

SafeWeb associates a set of security labels with each event. There are two types of labels, *confidentiality* labels and *integrity* labels. Confidentiality labels prevent sensitive data from escaping a given system boundary, whereas integrity labels are used to prevent low-integrity data from entering parts of an application. Confidentiality labels are "sticky"—once they are associated with an event, all events that are derived from that event will also contain those labels. In contrast, integrity labels are "fragile"—they are applied to an event only if all the events that this event was derived from also contain the same integrity labels.

Labels are represented as URIs. For example, `label:conf:ecric.org.uk/patient/33812769` could be used as a confidentiality label to protect the data of a specific patient, while `label:int:ecric.org.uk/mdt` could act as the integrity label for all data contained within the whole MDT application. In the MDT application, an event processing unit periodically reads unlabelled patient records from the main ECRIC database and produces events, which are labelled according to the encountered patient ID. This operation does not require privileges—it is always possible to add extra confidentiality labels to events. MDT-level aggregates, such as survival statistics or measures of information completeness (cf. §2.1), are labelled with a confidentiality label specific to that MDT.

Label enforcement is managed using associated privileges. Two types of privileges are used for confidentiality labels. The *clearance* privilege is used to access information protected by a confidentiality label. The privilege to make labelled information public by removing the label is referred to as *declassification*. Analogous privileges for integrity labels exist: *clearance* to low integrity and *endorsement*. To simplify presentation, we consider only confidentiality labels and the associated privileges in the rest of the paper.

Privilege assignment and checking is performed in SafeWeb by the event processing engine and the web frontend. Privileges associated with labels are assigned directly to units (in the backend) and requests (in the frontend) through a policy specification file. For more complex policies with dynamic privileges, a label manager could delegate privileges to units at runtime.

4.2 Event Broker

Units communicate by publishing events and by subscribing to events that they are interested in. To dispatch events among units, SafeWeb uses an event broker that matches subscriptions with published events. To support fine-grained data processing, SafeWeb uses a topic-based subscription language with optional content filtering on event attributes within a topic [7].

The event broker filters events according to their security labels. This is used to restrict the set of events that units can receive: for an event to be delivered to a subscriber, the set of its confidentiality labels must be a subset of those labels for which the subscriber possesses clearance privileges.

The event broker uses a modified version of the *Streaming Text Oriented Message Protocol* (STOMP) [24]—a simple, extensible, HTTP-inspired message protocol. It is language- and platform-agnostic and an open-source implementation [25] exists for Ruby. In STOMP, each request consists of a command, such as `CONNECT`, `SEND` or `SUBSCRIBE`, a set of optional headers and an optional body. A `destination` header is used to match subscriptions with publications by topic. An optional SQL-92 selector header specifies content-based subscriptions.

The implementation of our IFC-aware event broker is based on the STOMP implementation but has been extended with SSL support at the transport layer. At the dispatching layer, we have changed the matching semantics to respect labels, which are encoded as event headers with special semantics in `SEND` and `SUBSCRIBE` messages. In addition, subscriptions include unique identifiers to simplify the handling of subscriptions issued by different units. The client side of the STOMP implementation uses the event-based EventMachine I/O library [8].

4.3 Event Processing Engine

The event processing engine in SafeWeb provides a framework to support and control unit execution. Its key functions are (1) control of unit execution by checking and tracking security labels, (2) assignment of privileges to units, and (3) restriction of access to the environment. An event processing unit is realised as one or more classes that implement the business logic of the application.

```
1  subscribe '/patient_report', 'type=cancer' do |event|
2    list = get 'patient_list'
3    list push event[:patient_id]
4    set 'patient_list', list
5  end
6  subscribe '/next_day' do |event|
7    list = get 'patient_list'
8    publish '/daily_report', list, :remove => $LABELS,
9      :add => ['label:conf:ecric.org.uk/patient_list']
10 end
```

Listing 1. Example unit

The engine configures, instantiates and runs units and provides communication facilities using the event broker.

Listing 1 shows a unit that computes a daily list of patients with processed reports. The unit registers subscriptions for events published on the topics `patient_report` (line 1) and `next_day` (line 6). When a subscription is issued by a unit, the engine reads the set of labels from the unit's policy file for which the unit has *clearance* privileges. The engine then issues a subscription request to the broker with this set of labels; this set is used to check that a matching event can be processed by the unit. To support stateful units, the engine provides a unit-specific key-value store with labels associated with keys. It can be used for reading (lines 2 and 7) or storing (line 4) values, thus allowing different callbacks to communicate by exchanging state between them.

The engine prevents units from inadvertently disclosing confidential data because it controls the labelled events that they publish and isolates them from the external environment. We describe the two mechanisms for this in turn.

Label tracking. To ensure correct labelling, the engine associates a set of labels with the execution of a callback. This set, accessible to units as `$LABELS`, is initialised to the set of labels of the event being processed. When an event is published, the engine attaches all labels in `$LABELS` to the event. With each `publish` call, the unit can specify a set of labels to add or remove from the published event (lines 8 and 9), although removal is only permitted when the unit has the appropriate declassification privilege.

As values in the key-value store are labelled on a per-key basis, when a value is read from the store, `$LABELS` is updated to reflect its confidentiality—all the labels associated with the value's key are added to `$LABELS`. When writing to the store, all confidentiality labels in `$LABELS` are saved as the key's confidentiality, optionally adding and removing labels analogous to the `publish` call.

By maintaining labels from the received events to the published ones, and by labelling all datapaths through the shared key-store appropriately, confidentiality labels are preserved. However, unit callbacks could access other forms of unlabelled shared state, which would ignore label protection. In addition, they could bypass the event broker and use external APIs for console, disk, or network I/O, thus disregarding labels completely. To prevent this, the engine must execute unit callbacks in isolation.

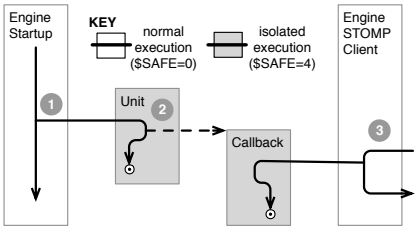

Fig. 2. Isolation of units and callbacks performed by the event processing engine

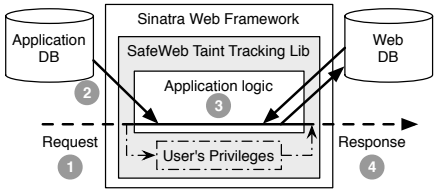

Fig. 3. Variable-level taint tracking in web frontend

Isolation. The goals of isolation are twofold: to prevent the use of I/O operations, and to prevent access to variables outside of the callback local scope, i.e. global variables, instance variables and local variables of enclosing scopes.

To isolate units from I/O and access to global variables, we use Ruby's *safe levels*. They restrict the execution mode of Ruby code and provide different kinds of isolated environments. `$SAFE` is a thread-local global variable that controls the current safe level. When set to safe level 4, it creates the most restrictive environment with the following irreversible effects on the current thread: (i) no access to I/O operations, (ii) all new objects are marked with a flag called "taint" and (iii) no write access to any object that is not tainted.

In Figure 2, we show how safe levels are used to achieve callback isolation. The engine executes units in a new thread, after setting `$SAFE=4` to prevent the unit's initialisation code (step 1) from performing I/O operations. Units register callbacks that execute when events arrive (step 2). When the STOMP client library that interacts with the event broker receives a matching event, it creates a new thread, sets `$SAFE=4` and executes the callback (step 3). The callback code cannot perform I/O or store events in global variables. It can only store events in the unit's key-value store that is tainted explicitly during unit allocation.

Isolation in safe level 4 still allows a callback to access variables of its enclosing scopes. To prevent this, we duplicate these variables when the callback is registered by using the meta-programming features of the Rubinius runtime [21].

Some units, however, need access to APIs that perform I/O, e.g. units that import and export events between the event engine and databases. To support this, the engine allows *privileged* units to execute without isolation at `$SAFE=0` and, thus, access I/O facilities. This effectively allows them to declassify any received event. To limit the power of privileged units, the engine prevents them from receiving events with certain labels by withholding their clearance privilege.

4.4 Web Frontend

The web frontend of SafeWeb presents results from the backend to users and enforces IFC without requiring changes to web applications. Web developers could be not fully aware of the policy requirements of the data that they present or, more often, they may introduce implementation bugs leading to unintended data disclosure. In the web frontend, SafeWeb's taint tracking library labels all

data to reflect the confidentiality of the principals that the data correspond to. When an application that is not policy compliant attempts to return incorrect data to the client, the operation can be aborted, preventing data disclosure.

The web frontend follows a traditional, database-driven architecture: a client issues an HTTP request, the request is served by the application server using the application database, and the HTTP response returns the result to the client. Since the application server handles requests from all users, it must have access to the data that *any* user may receive, i.e. all sensitive data in the application database. As a consequence, the web application would have to be trusted to remove all labels associated with data. Clearly, this would violate security requirement *S2* because any implementation error in the web application could result in inadvertent disclosure of data that should be visible only to a particular group of users, such as a given MDT.

To achieve the end-to-end security requirement, SafeWeb tracks data at a different granularity in the web frontend than in the event-processing backend. Instead of labels being attached to events, they are associated with individual *variables*. For example, when a variable n stores a patient name, n will carry a label that conveys the confidentiality of the patient name.

Labels are checked by SafeWeb when the web application returns an HTTP response to a client. For example, before the content of variable n is sent to a client, the client's privileges are validated to be a superset of the confidentiality labels associated with n. As described next, this is sufficient to provide end-to-end confidentiality guarantees without requiring a new application architecture, which would be challenging to adopt in a production environment [16].

Figure 3 shows the operation of SafeWeb's taint tracking library for Ruby. In step 1, an HTTP request arrives at the server. The request is authenticated and the confidentiality privileges of the associated user are retrieved from the web database. In step 2, the application queries the application database for the data needed to serve the request. SafeWeb's taint tracking library transparently adds the labels produced by units in the backend to the data fetched from the application database. In step 3, the application produces a response by carrying out application-specific processing of the data. SafeWeb's taint tracking library alters Ruby program statements and library methods to propagate labels correctly: e.g. when two strings are concatenated, the resulting string receives both operands' labels. In step 4, before sending the response to the user, the response's label is compared to the user's privileges from step 1—unless the user has the required privileges, the operation is aborted.

SafeWeb implements variable taint-tracking in Ruby using labels as follows. Its taint-tracking library redefines Ruby's `String` and `Numeric` subclasses: (1) to store labels within each instance and (2) to propagate labels correctly across method invocations. For example, SafeWeb's taint tracking library should propagate labels upon string concatenation. For this, it declares a new concatenation method in the `String` class called `nconcat()`. The taint tracking library then aliases the existing "`+`" method to call `nconcat()` and propagate labels. From then onwards, the runtime transparently invokes the redefined "`+`"

method when two strings are concatenated. Since we only support non-malicious code (§3.2), these changes are enough to effectively propagate taint.

The implementation exploits a standard meta-programming feature of Ruby and Ruby's pure object-oriented foundations: Ruby classes are open, all operators are defined as methods, and method definitions can change at any time. Implementing a similar taint-tracking library in other popular web languages, such as Java [4] or PHP [30], would require more extensive changes to the language runtime, making maintenance difficult in a production environment.

SafeWeb supports Ruby web applications running on the Rubinius runtime [21] and using the Sinatra web framework [23]. We use Rubinius due to its ability to manipulate the regular expression variables (`$~`, `$1`, etc.) directly. This is necessary to add taint tracking to Ruby's regular expression methods. Sinatra is used for its well-defined interception points of HTTP requests and responses. This allows SafeWeb's taint tracking library to intercept all communication to and from the client and attach label checks or fetch user privileges.

We do not introduce explicit features to prevent traditional Cross-Site Scripting (XSS) or SQL Injection (SQLI) attacks. Ruby objects support a `taint` method that marks a given object as originating from the user. The Ruby runtime stores this information per object and propagates it when strings are processed, similar to our label propagation. In the context of web applications, this mechanism can be used to enforce that every string is sanitised before being used in a sensitive operation, i.e. an HTML response or an SQL query.

5 Evaluation

The goals of our experimental evaluation are to explore the effectiveness of the SafeWeb middleware in preventing unauthorised data disclosure and to measure its performance overhead. We evaluate its security properties as part of the prototype implementation of the MDT web portal application.

5.1 Case Study: MDT Web Portal Application

As shown in Figure 4, the MDT web portal application uses three units: (a) A *data producer* obtains data from the main ECRIC database, leveraging the existing ECRIC framework for data access. It reads fields from different tables, labels them appropriately according to MDT and patient ID and publishes them as events to the event broker. For the sake of simplicity, we use only MDT-level labels as these are sufficient to satisfy our security requirements. (b) A *data aggregator* continuously collects all events related to individual cancer cases and combines their data. It produces aggregated records required by the MDT web application to satisfy functional requirements *F1–F3*. Implementation errors will not disclose data because of the isolation mechanism of SafeWeb. (c) Finally, a *data storage* unit, which has declassification privileges for all MDTs, handles data persistence. It stores processed records with their security labels in the

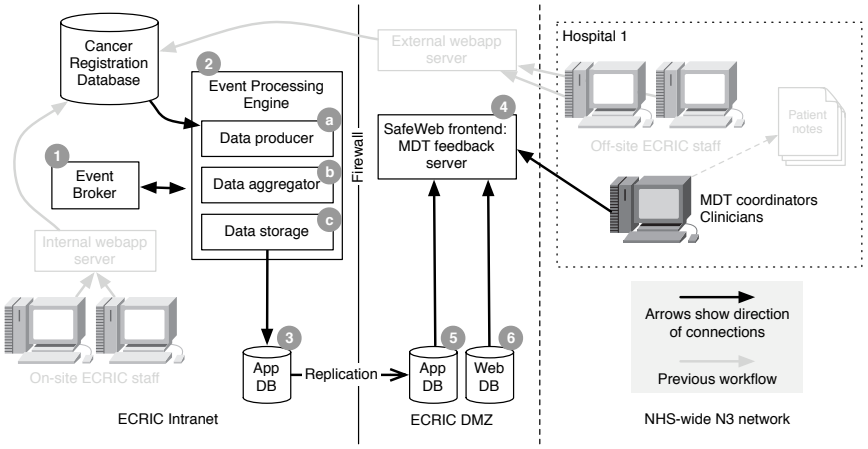

Fig. 4. Deployment of the MDT web portal application using SafeWeb within ECRIC's infrastructure

CouchDB application database. Security features of SafeWeb (i.e. IFC and strong isolation) allow the application to satisfy the security policy *P1*.

The Sinatra-based web frontend of the application uses CouchRest [5] to access CouchDB, and ERB for embedding Ruby in web pages. SafeWeb's taint tracking library enforces authentication centrally by adding hooks to all defined Sinatra rules. User accounts and their label privileges are stored in the web database. Currently, the web frontend uses HTTP basic authentication and TLS. We plan to add support for authentication using NHS smartcards in the future.

Deployment. Figure 4 shows how ECRIC's network is separated into three zones: an *Intranet*, a demilitarised zone (*DMZ*) and the NHS-wide *N3 network*. The Intranet is a restricted zone separated from the DMZ by a firewall, which permits only unidirectional connections to the DMZ. Core ECRIC infrastructure such as the main database is accessible only from within the Intranet.

The MDT web portal application is deployed within ECRIC as follows. The event broker (1) acts as a secure event bus for event processing units and is deployed within the ECRIC internal network. The units belonging to the application execute as part of the event processing engine (2). The MDT application uses a CouchDB application database (3), which contains the result data from the event-processing backend and provides result data to the web frontend (4). Because ECRIC's firewall only permits connections from the Intranet to the DMZ, we run two instances of the application database: in the Intranet (3) and in the DMZ (5). The application database is replicated periodically between the two instances using CouchDB push replication. The DMZ instance is read-only in order to prevent modifications by the web frontend, thus satisfying requirement *S1*. Data specific to the web frontend, e.g. session and usage data, is stored separately in a local web database (6) using the SQLite database engine.

```
1  require `sinatra'
2  require `safeweb−tracking'
3  get '/records/:mid' do
4    content_type :json
5    return nil if !check_privileges(params[:mid])
6    r = Records.by_mid(:key => params[:mid])
7    process r
8    r.to_json
9  end
```

Listing 2. Example of a rule in the web frontend of the MDT web portal

```
1  def check_privileges id
2    m = Measurement.find(id)
3    @is_admin or Privileges.count(
4    :conditions => {
5      :u_id => User.find_by_name(@username).id,
6      :hospital => m.hospital_id,
7      :clinic => m.type
8    }) > 0
9  end
```

Listing 3. An access control check used by the MDT web portal

5.2 Security Properties

Given the lack of third-party SafeWeb applications, we assess the security properties of SafeWeb by analysing its effectiveness in defending against known types of implementation errors. We inspected the Common Vulnerabilities and Exposures (CVE) database for vulnerabilities classified as "Information Disclosure", "Access Control" or "Design Error" and organised them into generic subcategories that share the same underlying cause. We then inject similar vulnerabilities to our MDT application and observe if SafeWeb can prevent them.

Omitted Access Checks. The most common problem that leads to information disclosure (e.g. CVE reports 2011-0701, 2010-2353 and 2010-0752) is the omission of access control checks. To emulate this, we remove the MDT privilege check from the patient filtering routine that normally precedes the filtering of patient details (Listing 2, line 5). Without SafeWeb's taint tracking library (line 2), sensitive information disclosure occurs in line 8. However, when the taint tracking library is included and an MDT requests data they are not allowed to see, the library correctly taints the JSON string (line 8) and displays an error message.

Errors in Access Checks. Even when an access control check is present, it may not specify the correct security policy and may result in information disclosure. Often, these errors involve specially constructed input and do not manifest themselves under normal operation, making them hard to discover (e.g. CVE reports 2011-0449, 2010-3092, and 2010-4403). To introduce such a problem, we modify the user lookup operation (listing 3, line 5) to ignore the case of the username. This may lead to two MDTs sharing privileges. To test this, we create two MDTs with usernames `mdt1` and `MDT1` but with different privileges. SafeWeb's taint tracking library, when included, successfully prevents access of the second MDT to all the patient details that only the first MDT should see.

Inappropriate Access Checks. Security policies are often complicated and developers may not fully understand them. This category of vulnerabilities covers correctly applied checks that do not enforce the intended policy (e.g. CVE reports 2010-4775 and 2009-2431). To emulate such issues, we remove the check for clinic equality from `check_privileges` (Listing 3, line 7). This effectively enables any MDT to see the data of all the patients in the same hospital. Again, the error does not result in information disclosure: SafeWeb's taint tracking library detects the taint of the output, generates an error and prevents access.

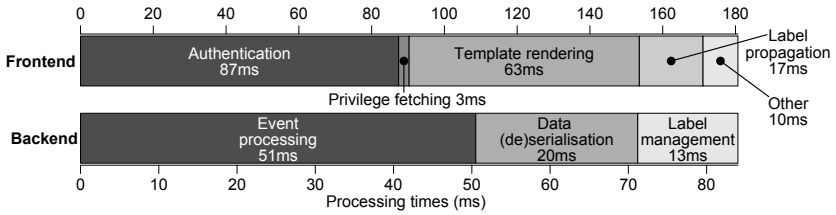

Fig. 5. Processing latency within the MDT web portal application

Design Errors. The last category captures vulnerabilities due to design errors in the application's business logic (e.g. CVE reports 2011-0899 and 2010-3933). Such errors involve cases in which the application processes sensitive data in unexpected ways that lead to data disclosure. We modify the data aggregator unit to ignore the hospital of origin when matching events. As a result, the unit generates records that mix data of different MDTs. SafeWeb's event processing backend allows access to these events but requires that the output records have labels of both MDTs. Thus, when the frontend attempts to display these records, access is prevented because no MDT has the necessary privileges.

Trusted Codebase. SafeWeb enforces security policies but it does not eliminate trusted code: (1) SafeWeb's taint tracking library must be trusted to correctly authenticate users, associate privileges with their requests, propagate labels and check the labels of each response. (2) The event backend must isolate non-privileged units and label their output. (3) Privileged units must label events that they publish or store in the application database correctly. (4) The policy file that specifies user privileges in the web frontend and unit privileges in the event backend, as well as the scripts that edit it, must be audited for errors.

A code audit is still required, but it can focus on the SafeWeb implementation and only includes a small application-specific part. SafeWeb's taint tracking library consists of 1943 LOC and the event processing engine has 1908 LOC. After this trusted codebase has been audited once, data confidentiality only depends on the correctness of a small part of each application. For the MDT web portal, the code that has to be audited involves the two privileged units in the backend (138 LOC) and the code that assigns privileges to new MDTs in the frontend (142 LOC). The confidentiality of patient data does not depend on the other 2841 LOC of the MDT application: no further security audit is required.

5.3 Performance Overhead

In this section, we measure the performance overhead of SafeWeb in terms of latency and throughput. All measurements are taken on an AMD Opteron 6136 2.4GHz system with 16 GiB of RAM running Ubuntu 10.04. The 95% confidence interval for each value we report extends to each side at most 5% of the value.

For the web front-end, we measure the page generation time of the MDT application's front page with and without SafeWeb's taint tracking library. We issued 1000 requests and measured the time required to render the response.

With SafeWeb's taint tracking library, the page generation time increases by 14% from 158 ms to 180 ms. For the back-end, we measure the average latency of individual events from the data producer to the data storage unit during the processing of 1000 events. With SafeWeb's isolation and label checks, the latency to process a single event increases by 15% from 73 ms to 84 ms. Overall, this is an acceptable overhead for a web application with strong security requirements.

Figure 5 shows a break-down of the overall latency when SafeWeb is enabled. In the front-end, HTTP basic authentication takes 87 ms to which privilege fetching adds 3 ms; processing of the `ERB` template takes 63 ms, to which label propagation adds 17 ms. "Other" includes operations like network transmission and database access. For the back-end, processing an event takes 51 ms plus 20 ms for serialisation, to which SafeWeb adds 13 ms for label management including label (de)serialisation and checking.

SafeWeb provides ample throughput for the low event rates of the MDT portal. We employ a synthetic benchmark with two units, an event producer and an event consumer, to measure the throughput reduction that SafeWeb incurs. We measured the end-to-end event throughput between the two units by having the producer publishing events at the maximum sustainable rate while confirming that the memory consumption remained stable. We sampled the throughput once per second for 1000 seconds. With label tracking active, event throughput decreased (-17%) from 4455 events/second to 3817 events/second. Due to the language isolation support in Ruby, the decrease in performance is minimal and comparable to approaches that rely on low-level runtime modifications [30].

6 Related Work

The most common security problems in web applications arise from the handling of untrusted user data in the application's output. This leads to problems such as XSS and SQL injection attacks. Web application frameworks protect against such vulnerabilities (e.g. XSS, CSRF) [26]. Applications developed with SafeWeb can still benefit from this (e.g. RailsXSS [19], Rack::Csrf [18]) to avoid traditional exploits that often disclose data by hijacking user accounts. In addition, SafeWeb improves these frameworks to prevent sensitive data disclosure.

Sensitive data disclosure is often caused by insufficient authorisation, missing access control checks or by errors in application semantics. Potential solutions include static analysis [9, 11], symbolic execution [2, 3] and runtime taint tracking [4, 28, 15, 17, 30]. Static analysis tools for dynamic web languages often have high false positive rates [9] or do not support all language features [11]. Symbolic execution explores all possible execution paths of a web application and can prove absence of certain errors [3]. However, devising assertions for symbolic execution is a manual task and involves many of the shortcomings of manual security audits. In contrast, our approach requires minimal developer involvement.

At runtime, access control can be enforced transparently on each operation [22] or when sensitive events are received [27]. Nevertheless, errors in the application logic may still convey sensitive data. Taint tracking systems trans-

parently track sensitive data and protect against inadvertent disclosure despite application errors. They have been provided, amongst others, for Java [4], C [28, 15] and PHP [17, 30]. Simple approaches use one bit taint per string for injection attack protection [17]. In contrast, SafeWeb's taint tracking library attaches full security labels to each variable, offering end-to-end guarantees about sensitive data disclosure. Resin [30] uses pointers to user-defined policy objects, such as IFC labels, however, it requires extensive language runtime modifications.

7 Conclusion

We have designed and implemented SafeWeb, a middleware for creating secure, event-based web applications. It provides strong end-to-end security guarantees, while integrating with existing web development practices. We have demonstrated SafeWeb as part of a web application for assisting cancer treatment practices within the UK National Health Service (NHS).

The strict data security requirements across multiple interacting organisations provided us with a set of real design constraints. The sensitivity of healthcare data required careful consideration of the parts of the middleware that push and pull data. The back-end requirements suited an event-driven design, whereas the front-end is a typical web application. We showed that information flow control can be applied to both the event-processing back-end and the web front-end as part of a middleware. This gives security assurances regarding data disclosure, and minimises organisations' code audits.

In future work, we plan to explore how SafeWeb could become the basis for wider deployment of healthcare applications at the national level. Scaling up will involve creating separate, independent regional instances of SafeWeb, which can interact with each other in a secure fashion. In addition, we want to investigate the use of SafeWeb for other classes of web applications.

References

1. D. Bell and L. LaPadula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical report, MITRE Corporation, 1976.
2. C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, San Diego, CA, 2008. USENIX.
3. A. Chaudhuri and J. Foster. Symbolic security analysis of Ruby-on-Rails web applications. In *Computer and communications security*, Chicago, IL, 2010. ACM.
4. E. Chin and D. Wagner. Efficient character-level taint tracking for Java. In *Workshop on Secure Web Services (SWS)*, pages 3–12, Chicago, IL, 2009. ACM.
5. CouchRest. `http://github.com/couchrest`, 2011.
6. Department of Defense. Trusted Computer System Evaluation Criteria, 1983.
7. P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.
8. EventMachine. `http://rubyeventmachine.com`, 2011.

9. Y.-W. Huang, F. Yu, Hang, et al. Securing web application code by static analysis and runtime protection. In *World Wide Web (WWW)*, New York, NY, 2004. ACM.
10. Information Commissioner's Office, UK. Data breaches to incur up to £500,000 penalty. `http://www.ico.gov.uk/~/media/documents/pressreleases/2010/PENALTIES_GUIDANCE_120110.ashx`, 2010.
11. N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *Symposium on Security and Privacy*, Berkeley, CA, 2006. IEEE.
12. M. Miglivacca, I. Papagiannis, D. Eyers, B. Shand, J. Bacon, and P. Pietzuch. High-performance event processing with information security. In *USENIX Annual Technical Conference*, Boston, MA, 2010.
13. A. Myers and B. Liskov. Protecting privacy using the decentralized label model. *Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.
14. S. Nair, P. Simpson, B. Crispo, and A. Tanenbaum. A Virtual Machine Based Information Flow Control System for Policy Enforcement. *Electronic Notes in Theoretical Computer Science*, 197(1):3–16, 2008.
15. S. Nanda, L.-C. Lam, and T.-c. Chiueh. Dynamic multi-process information flow tracking for web application security. In *Middleware*, Toronto, Canada, 2007. ACM.
16. I. Papagiannis, M. Migliavacca, D. M. Eyers, B. Shand, et al. Enforcing user privacy in web applications using Erlang. In *W2SP*, Oakland, CA, 2010.
17. T. Pietraszek and C. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection*, Menlo Park, CA, 2006.
18. Rack::Csrf. `http://github.com/baldowl/rack_csrf`, 2011.
19. RailsXSS. `http://github.com/rails/rails_xss`, 2011.
20. I. Roy, D. Porter, M. Bond, K. McKinley, and E. Witchel. Laminar: Practical fine-grained decentralized information flow control. In *PLDI*, Dublin, Ireland, 2009.
21. Rubinius. `http://rubini.us`, 2011.
22. P. D. Ryck, L. Desmet, and W. Joosen. Middleware support for complex and distributed security services in multi-tier web applications. In *Engineering Secure Software and Systems (ESSoS)*, pages 114–127, Madrid, Spain, 2011.
23. Sinatra. `http://www.sinatrarb.com`, 2011.
24. Stomp Protocol. `http://stomp.github.com`, 2011.
25. StompServer. `http://stompserver.rubyforge.org`, 2011.
26. J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song. An empirical analysis of XSS sanitization in web application frameworks. Technical report, UC Berkeley, 2011.
27. A. Wun and H.-A. Jacobsen. A Policy Management Framework for Content-Based Publish/Subscribe. In *Middleware*, Newport Beach, CA, 2007. ACM.
28. W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Security Symposium*, Vancouver, Canada, 2006. USENIX.
29. C. Ye and H.-A. Jacobsen. *The Smart Internet*, chapter Event exposure for web services: a grey-box approach to compose and evolve web services, pages 197–215. Springer-Verlag, Berlin, Heidelberg, 2010.
30. A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *SOSP*, Big Sky, MT, 2009. ACM.
31. S. Yoshihama, T. Yoshizawa, Watanabe, et al. Dynamic information flow control architecture for web applications. In *ESORICS*, Dresden, Germany, 2007.