INFORMATION FLOW CONTROL FOR A MEDICAL RECORDS WEB PORTAL

Thomas F. J.-M. Pasquier
University of Cambridge
tfjmp2@cl.cam.ac.uk

Brian Shand
Eastern Cancer Registry and Information Centre
brian.shand@cbcu.nhs.uk

Jean M. Bacon
University of Cambridge
jean.bacon@cl.cam.ac.uk

ABSTRACT

In this paper we present a web application prototype[1] developed for brain tumour patients to access their medical data. The prototype has been developed using existing technology for easy and cost effective deployment, yet supporting strong security requirements. With these constraints in mind we developed an architecture incorporating Information Flow Control for data isolation and audit.

KEYWORDS

Information Flow Control, Medical Data, eHealth

1. INTRODUCTION

It has been mandatory in England since 1971 for cancer records to be held by eight regional cancer registries. This data has traditionally been used to provide researchers and policy makers with statistics on the disease; anonymised data sets have been provided to researchers. There has recently been a move in the UK towards access by individuals to data held about them. For cancer data, a first step has been to provide this access for brain tumour patients; this paper describes a web application developed for this purpose.

This move creates a number of challenges that must be overcome. First, when data are released to medical practitioners, they work within a well defined and restricted network, where the use of standard *firewall* and *complex authentication* based techniques is possible (in the cancer registries' case, authentication is done using USB authentication tokens). Furthermore, the behaviour of the practitioners is strongly regulated. The first challenge is therefore to ensure that for personal access, the data is released to individuals securely within the system while making them aware of possible dangers when the data enters the user domain. The second challenge, is the need to use standard hardware and software to deploy the solution. This is to minimise the cost and make the service relatively user-friendly through familiarity of end users with standard software.

These challenges lead to a number of requirements for our application. The final solution should be accessible on a standard web-browser, it should not require installation of additional software, it should not require the use of an authentication token visible to the user and it should be deployable on Platform as a Service (*PaaS*)

[1] test on a selected group of real patients is scheduled to start in 2013.

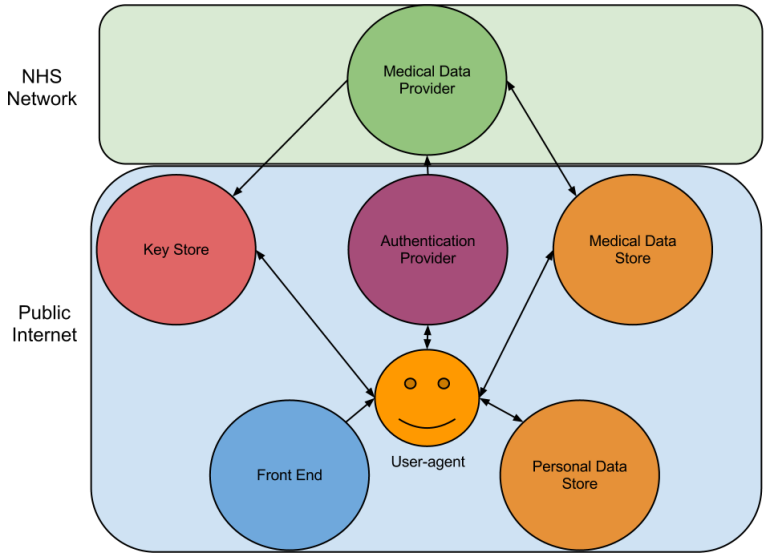

Figure 1: The system architecture

cloud solutions (such as Amazon EC2[2] or Rackspace[3]).

Furthermore, literature on the subject either assumes that the servers between the Medical Data Provider (MDP) and the end user can be trusted [Sunyaev et al. 2010a] or use complex encryption schemes [Alshehri et al. 2012, Benaloh et al. 2009, Li et al. 2010]. We argue that third parties should not be trusted and that if the purpose is to provide access and visualisation of medical records it is unnecessary and potentially risky to let third parties access the data. Also complex encryption schemes prevent large scale deployment because of the need to deploy specific software or even dedicated hardware.

In this context, we designed our system to protect patient privacy and reduce the risk of data leakage by considering the hosting platform to have a rather low level of trust. In order to enforce end-to-end privacy we rely on some well known principles and techniques: encryption, partitioning of data and Information Flow Control (IFC). It is important to note that protecting the data within the patient's computer is out of scope for this paper. Our focus was to provide end-to-end privacy from when the information leaves the MDP to when it reaches the patient's user agent (the web browser).

We now discuss what we mean by encryption, partitioning and IFC. Encryption has been used to ensure that the data can only be read by the patient and that no third party can gain access to it while it is stored on *PaaS* or in transit through the network. If the data happened to be disclosed, for example, by key compromise, a bug within the software and/or storage method, the medical and personal data are stored and accessed from two different separate virtual machines (VM) in the cloud. Medical data are de-identified medical records that minimize the risk of re-identification [Bertino et al. 2005]; personal data are patient-supplied information and/or information that allows the patient to be identified. In order to gain useful information about a patient an attacker would have to combine the two sets of information, which is made difficult by the fact they are stored on two different VMs. Finally, IFC techniques are used to ensure that data associated with a user are delivered only to the user presenting the corresponding credential. This is a software mechanism which ensures that, even in the case of buggy software, the server will refuse to serve data which does not belong to the currently authenticated user; more detail is given in section 3.

2. ARCHITECTURE

Our achitecture is centered around the user-agent. Indeed as we need to encrypt the data and partition them, while not fully trusting the infrastructure between the user-agent and the *MDP*, it must be the responsibility of the user-agent to combine and decrypt the different pieces of information concerning the patient.

As shown in figure 1, the system is divided into separate entities. A central component is the Authentication Provider which behaves as a single sign-on mechanism where the user authenticates in order to gain access to

[2]http://aws.amazon.com/ec2/

[3]http://www.rackspace.com/

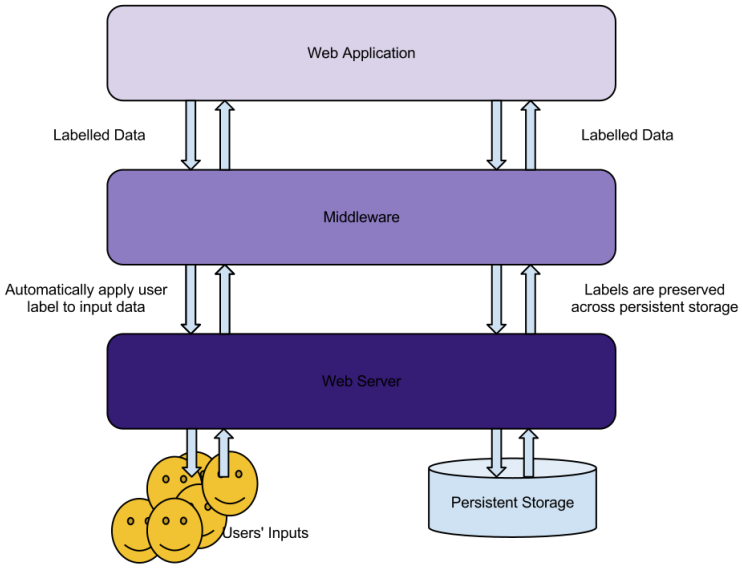

Figure 2: Architecture of the Data Store

the different pieces of information stored in the separate servers. Two data stores are used, one for medical data (Medical Data Store), the other for personal information (Personal Data Store) and a Key Store is used to store the keys needed to decrypt information. Note that the persistent medical data is held in a private, secure database; the Medical Data Store holds data constructed in response to users' requests. Finally a Front End server provides the interface and web page environment to display the information to the user. For example, this might be incorporated into a cancer charity's web presence.

A typical access to a page with sensitive information from the Front End is done as follows (we assume the user is already authenticated):

- the user accesses a front-end web page containing references to sensitive information that should be displayed on this page;
- the User Agent retrieves those data from the appropriate server (medical and/or personal);
- the User Agent retrieves the corresponding encryption key(s) from the Key Store;
- the User Agent displays the data within the page layout as described by the Front End;

Those operation, from the user point of view, are carried transparently. Indeed the front-end through the use of a dedicated library automatically generate javascript code, which will perform those operations and cache data as needed, providing navigation experience close to a standard website.

In order to build the central authentication system we adopt a reverse OAuth 2.0[4] approach similar to the one described by Gonzalez et al. [Gonzalez et al. 2009]. When a user tries to access one of the data stores he is redirected to an authentication page on the Authentication Provider, accompanied by a randomly generated string to ensure the freshness of the request. Once the user is authenticated, the Authentication Provider redirects to the end-point specified by the data store concerned and includes a signed authentication token and an identifier (ID) representing this user uniquely. This unique ID which carries no real identification value is used by the data store to keep track of different users. The authenticity of the token is ensured by the random string freshness method and a standard private/public key mechanism. In addition to providing a delegated single sign-on mechanism, reverse OAuth 2.0 allows the user identity to be abstracted from the data stores.

In order to push medical data, the MDP needs to retrieve from the Authentication Provider the list of newly registered patients, their details and their pseudonymised ID used for access to the data stores and key store. Their registration details, once retrieved by the MDP, are deleted from the authentication portal as they are no longer of use and could compromise patient privacy. Those details are used within the MDP infrastructure to identify patients within the MDP database. Once the patients have been identified, the data are transformed for publication by anonymisation, formatting and encryption. The key used for encryption is then pushed to the

[4]http://oauth.net/2/

Key Store and associated with the correct patient, the same is done for the medical data. The medical data is given a label which is associated with the user. The labelling mechanism is further discussed in section 3.

All accesses to the system are logged in order to detect suspicious behaviour. Furthermore, those logs (or at least human friendly simplified versions) are made available to the patient to promote transparency and involve the patient with security related problems. If suspicious behaviour at a single patient level is detected there are several possible actions:

- the association between a patient and his labels can be deleted rendering the data inaccessible. If the case is cleared later, this association can be restored; if not, the data can safely be deleted;
- if there is serious doubt of compromise all data are deleted;

3. INFORMATION FLOW CONTROL (IFC)

Information Flow Control tracks how information propagates within a program during its execution to ensure that the information is handled in a proper fashion. In the case of a web application, we want to ensure strict isolation between different users' data [Bacon et al. 2010, Hosek et al. 2011]. Therefore, we ensure, through a taint tracking mechanism, that every piece of information is associated with a specific label and let only the patient associated with that label access those data. Other protection mechanisms such as access control, assume a well behaved and trusted process to handle the data, but cannot prevent this process leaking data. In the case of a webserver handling confidential information, it would be extremely problematic if such a process leaked information from Alice to Bob through an unexpected bug.

In our solution the IFC label can be described as follows $L = \{name; o_n : r_i, ..., r_j\}$ (there is a strong similarity with labels as described by Myers [Myers 1999]). The name is a user set name used to identify a specific label from others, o_n is the owner of the label, $r_i, ..., r_j$ are the users allowed to access information associated with this label. The simplest label is $L = name; o_n$, meaning that only the owner o_n of the label is able to access the data associated with this label.

Consider a simple example: Alice wants to push into our data store some contact information. Alice sends to the server the following information: $\{key : \text{“}contact\text{”}, value : \text{“}[...]\text{”}, label : \text{“}alice\_label\_1\text{”}\}$. The server first transforms the label name using a hash function such that $stored\_name = H(alice\_id, label)$. We now have created the following label $L = \{stored\_name, o_{alice}\}$. If Bob tries to access the data, the server will fail. If Alice wants to give Bob access to her information, she needs to send a request to the server to modify her label such that $L = \{stored\_name, o_{alice} : r_{bob}\}$. Our model does not permit addition or deletion of ownership as we are dealing with medical records with a clear owner: the patient.

The IFC mechanisms are not enforced and managed within the web application, but in a middleware layer as described in figure 2. Therefore, even an application containing unexpected or buggy behaviour would not be able to grant access to one patient's information to another. The main application's access to the database is filtered through the middleware which verifies labels' associations and enforces them, generating a failure when inappropriate accesses (read or write) are requested. At this stage of the project the application is built on top of a standard SQLLite database. The middleware database interface for every read and write verify the labels associated with the retrieved data match with the labels associated with the current user.

Another layer, associates any data pushed by the server with the user specified labels (it is important at this point to know that the framework does not provide a facility to remove a label). With these two layers we ensure that the data going in and out of the application through the expected input/output of the application carry valid tags. Finally, a taint guard mechanism as described by Hosek et al. [Hosek et al. 2011], ensures that if the application tries to push data with inappropriate labels out of the application the server process will fail. We are able to guarantee that, given an appropriate authentication token, a user is unable to access data he did not create himself or which he has not been granted access to.

4. IMPLEMENTATION

4.1 Data Store

The data store is totally agnostic to the format of the data it stores. However, in practice to interact easily with the javascript generated by the front end (described in section 4.2), the data is stored under the json format. The interface used for our key-value store is intentionally simple. There are four functions:

- get(key)

```
{"data":[
  {
    ``full_name":``John Doe",
    ``job":``MD",
    ``address:``Somewhere",
    ``phone":``991",
    ``email":``john.doe@test.com"
  },
  [...],
  {
    ``full_name":``Jane Doe",
    ``job":``Nurse",
    ``address:``Somewhere else",
    ``phone":``991",
    ``email":``jane.doe@test.com"
  }
]
}
```

Figure 3: Stored unencrypted data format

- get_all
- set(key, value, label)
- delete(key)

The provided interface is close to standard key-value store interfaces with the exception of the set function which takes an additional label attribute. The label attribute is used to mark the data and allow access by a third party to values bearing the indicated tag. The label is a string value and a stronger per-user unique label is built server side from the user's unique identifier and the string provided. In order to facilitate the communication with the javascript running within the user-agent, the data store communicates with it by *POST* with the json formatted value.

Additional functionality needs to be added in order to be able to share data. Each label has an owner, who can associate other users with this label or remove this association. Users associated with a label, but who do not own it, are not allowed to associate a further user with it. Therefore, only the original owner is able to decide which users should be able to see his data. This adds the two following functions to our interface:

- grant_access(user, label)
- revoke_access(user, label)

For further security the data store generate the javascript used to perform those API call and enforce *same domain-policy*[5]. The front-end web page communication with it through *HTML5 web-messaging* [6] and by including an html iframe pointing to the data store.

4.2 Front-end

The front end has been realised in Ruby-on-Rails. This lightweight framework has been designed to hide from the developer the details of the interaction between the generated webpage and the data store. In consequence, the developer need only to specify the origin of the data and how they should be displayed. As discussed in section 4.1, the data are stored in the json format and we assume the programmer is aware of the structure of the data stored. Figure 3 presents how contact information would look like once unencrypted.

In order to display the data represented in figure 3, the programmer would simply write the code presented in figure 4. This code specifies that the value associated with the key *contact_info* should be retrieved from the *PERSONAL* store and that each of the elements of this value should be applied to a given html pattern. The javascript code retrieving the data from the data store, unencrypting it and inserting the data following the pattern specified is then automatically generated.

In order to prevent cross-site scripting attacks modern browsers prevents web-pages from executing script from another domain of origin. This is known as the *same origin policy*. Therefore we include where needed

[5] http://www.w3.org/Security/wiki/Same_Origin_Policy

[6] http://www.w3.org/TR/webmessaging

```
<%= display_each_as(PERSONAL, :contact_info,
'<div class=''contact">
  <h3><!full_name!></h3>
  <ul>
    <li>Job Title: <!job!></li>
    <li>Address: <!address!></li>
    <li>Phone: <!phone!></li>
    <li>Email: <!email!></li>
  </ul>
</div>'
) %>
```

Figure 4: Ruby front-end code to display contact data

iframe pointing to the relevent data store. Information are then exchanged with those iframe through *HTML5 messaging API* and the data store trusted code within those iframe perform the data store API call. Finally, in order to maintain good reactivity of the website and to provide the user with a seamless experience we use *HTML session storage*[7] as a local cache store.

Other functions are available to the frond end developer such as automatic form generation to push data to the data store, pre-fetch data into the user-agent session storage to improve interface reactivity, reset of encryption key, deletion of data and others. Through those functions the programmer is able to develop a full fledged website, offering similar functionality that one relying on more standard technology.

Let examine a simple use case. The front-end provider, in our case a charity helping patients with brain tumours, provides them with access to their medical record. The MDP would like to perform quality of life survey in order to generate statistics. The front-end code will generate a form which push data to the medical store and grant the MDP provider access to the data.

This example is illustrated in figure 5. For the reader familiar with ruby on rails syntax this code is quasi-identical to the one used to generate a standard form. The only method which differs is the *form_store_tag* method which replace the *form_tag*, which takes one parameter. The second method *grant_access* will generate a button and the associated javascript for calling *grant_access* from the data store discussed in section 4.1.

5. RELATED WORK

Our work is mostly based on the SAFEWEB [Hosek et al. 2011] solution which provides a dynamic taint tracking mechanism for Ruby web applications, while not requiring modification to the language and only minor modification to the application code. There is a similar framework for python [Bello and Russo 2012, Conti and Russo 2012] or PhP [Papagiannis et al. 2011], however the last one does not support labels persistence. We decided to go one step further by separating front-end formatting from handling of sensitive data. We motivate this decision as we believe third parties do not require actual access to the data. Patient privacy is better protected by ensuring that no single third party involved is able to combine the whole set of patient data and should not see those data unencrypted under normal operation.

Separating and encrypting personal and medical data is not a new idea [Heurix and Neubauer 2011, Neubauer and Heurix 2011]. However, because we combine encryption and IFC techniques our encryption scheme can be much simpler. Access delegation in our case is not done through a complex encryption mechanism, but through a relatively simple IFC mechanism by simply providing access by a third party to a particular IFC label. Indeed, complex encryption mechanisms for delegation [Alshehri et al. 2012, Benaloh et al. 2009, Li et al. 2010], can be avoided using our method, while still providing as fine grained access control and encryption as necessary.

Microsoft Health Vault and Google Health [Sunyaev et al. 2010a;b] provide access to third parties to patient medical and personal data. Third parties behave like the front end of our application. They format or modify the information to present it to the end user. However, we argue that it is unnecessary to give those applications access to the data to perform such tasks and that it can potentially be harmful.

[7] http://www.w3.org/TR/webstorage

```
<%= form_store_tag(MEDICAL, :label=>'quality_of_life_survey') do %>
  <%= label_tag(:tiredness, ''How tired do you feel from 0 to 5?") %>
  <%= select_tag(:tiredness, options_for_select(...)) %>
  <%= label_tag(:depression, ''How depressed do you feel from 0 to 5?") %>
  <%= select_tag(:depression, options_for_select(...)) %>
  [...]
<% end %>
<%= grant_access(MEDICAL, MEDICAL_PROVIDER, :label=>'quality_of_life_survey') %>
```

Figure 5: Ruby front-end code for quality of life survey

6. CONCLUSION

We have designed a system capable of protecting patient privacy between the Medical Data Provider and the User Agent. This is achieved by using the Information Flow Control mechanism to ensure that only the correct end user can access a given set of data. The de-identification, partitioning and encryption of data ensure that in order for an attacker to re-identify a useful number of patients, this attacker needs to compromise several servers.

However, our solution has a number of limitations. First, we did not address security within the user agent. To our knowledge, there is no standard web browser providing *application* sand-boxing. At the moment we trust the front-end to provide a script which does not transmit the data to third parties and we trust the user not to have software running that is capable of capturing the information displayed. With a sandbox ensuring that no data can be transmitted to unspecified i/o (in our case the display) and that no third party application can access the data displayed, we would be able to ensure end-to-end privacy (investigation on enforcement of Information Flow Control in Javascript exists [Austin and Flanagan 2012]). Information Flow Control is not enforced within the database, but at the application level; we believe this is also an area worthy of investigation.

The system proposed in this paper can be applied as it stands to any situation where data should be accessed only by the individual to whom the data relates. Furthermore, when the data is sensitive we believe it is of the utmost importance to store separately the sensitive data and necessary associated personal and/or identifying data. It is likely that a great deal of data will migrate to cloud services; private clouds when the data is sensitive. Here, data is likely to be held in encrypted form. Our approach of keeping the data encrypted and protected until it reaches the end user integrates well with this cloud scenario. As discussed above, we believe that current designs where third parties are allowed access to unencrypted data may eventually lead to unintentional or malicious leakage of patient information.

References

Alshehri, S. et al, 2012. Secure access for healthcare data in the cloud using ciphertext-policy attribute-based encryption. *Proceedings of the 28th International Conference on Data Engineering Workshops (ICDEW)*. Arlington, Virginia, USA, pp 143–146.

Austin, T. and Flanagan, C., 2012. Multiple facets for dynamic information flow, *In SIGPLAN Notices*, Vol. 47, pp 165–178.

Bacon, J. et al, 2010. Enforcing end-to-end application security in the cloud (big ideas paper). *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*. Bangalore, India, pp 293–312.

Bello, L. and Russo, A., 2012. Towards a taint mode for cloud computing web applications. *Proceedings of the 7th Workshop on Programming Languages and Analysis for Security*. Beijing, China, pp 7:1–7:12.

Benaloh, J. et al, 2009. Patient controlled encryption: ensuring privacy of electronic medical records. *Proceedings of the 2009 ACM workshop on Cloud computing security*. Chicago, Illinois, USA, pp 103–114.

Bertino, E. et al, 2005. Privacy and ownership preserving of outsourced medical data. *Proceedings of the 21st International Conference on Data Engineering*. Tokyo, Japan, pp 521–532.

Conti, J. and Russo, A., 2012. A taint mode for python via a library, *In Information Security Technology for Applications*, Vol. 7127 of *Lecture Notes in Computer Science*, pp 210–222.

Gonzalez, J. et al, 2009. Reverse oauth: A solution to achieve delegated authorizations in single sign-on e-learning systems, *In Computers and Security*, Vol. 28, pp 843–856.

Heurix, J. and Neubauer, T., 2011. Privacy-preserving storage and access of medical data through pseudonymization and encryption, *In Trust, Privacy and Security in Digital Business*, Vol. 6863 of *Lecture Notes in Computer Science*, pp 186–197.

Hosek, P. et al, 2011. Safeweb: A middleware for securing ruby-based web applications, *In Middleware 2011*, Vol. 7049 of *Lecture Notes in Computer Science*, pp 491–511.

Li, M. et al, 2010. Securing personal health records in cloud computing: Patient-centric and fine-grained data access control in multi-owner settings, *In Security and Privacy in Communication Networks*, Vol. 50 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp 89–106.

Myers, A., 1999. Jflow: practical mostly-static information flow control. *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. San Antonio, Texas, USA, pp 228–241.

Neubauer, T. and Heurix, J., 2011. A methodology for the pseudonymization of medical data, *In International Journal of Medical Informatics*, Vol. 80, pp 190–204.

Papagiannis, I. et al, 2011. Php aspis: using partial taint tracking to protect against injection attacks. *Proceedings of the 2nd USENIX conference on Web application development*. Portland, Oregon, USA, pp 2–2.

Sunyaev, A. et al, 2010a. Comparative evaluation of google heath api vs. microsoft healthvault api. *Proceedings of the Third International Conference on Health Informatics*. Valencia, Spain, pp 195–201.

Sunyaev, A. et al, 2010b. Evaluation framework for personal health records: Microsoft healthvault vs. google health. *Proceeding of the 43rd Hawaii International Conference on System Sciences (HICSS)*. Hawaii, USA, pp 1–10.