SecureForm: Secure Client-Side Browser Encryption

Steven Allen, Praynaa Rawlani, Nikki Shah, Emily Su

---

1. Introduction

1.1 Threat model

With rising security concerns about sending and receiving sensitive information over the internet, internet users increasingly value secure client-side encryption measures. We present a design for a Firefox add-on to protect user data from both code running on the web server and on the website. We assume a threat model where the adversary can steal sensitive data from the server and/or run third-party code on the client-side. We assume that the browser is trusted.

A number of current systems and services attempt to use client-side encryption. For example, Senditonthenet [1] (a file sharing system) and several password managers [2] utilize client-side encryption. A recent approach has been the introduction of Mylar [3] by Raluca Ada Popa, et al., which is a platform that provides secure web applications in presence of a malicious server. However, there are no well-known implementations of client-side encryption that do not trust foreign, website specific code. These existing systems would therefore violate our threat model. We propose the following Mozilla Firefox add-on, in an attempt to remedy the shortcomings of current client-side encryption.

1.2 Requirements

Our add-on runs on v25 and v26 of the Firefox browser. We currently do not offer key generation and therefore rely on and require the user's pre-generated PGP keys. However, we provide a key ring to upload and store the public and private keys, which will be used to encrypt and decrypt the contents of the forms. Our Firefox add-on works with any website that uses `SecureForm` objects on their web page. The extension is available publicly at https://github.com/Stebalien/SecureForm.

In our paper, we use the term "form" to include any arbitrary fields where a user can input some data, regardless of whether or not they are included in a `<form>` tag. Section 2 overviews our design and implementation details. Section 3 discusses how our extension addresses and defends against various threats. In section 4, we present some use cases for our extension. Section 5 considers our design limitations and suggests some ideas to extend our work in the future. At the end of this paper, we include an appendix with API details, screenshots demonstrating the use of our

extension for securing a course submission website and our key ring implementation.

2. Implementation

A website can secure any input fields, e.g. as part of a survey, email or arbitrary forms, by specifying a `SecureForm` object with two URLs: a controls URL and a form URL. The form URL specifies the actual form, which the website can indirectly modify. However, the website cannot read the contents of the form or listen to events initiated from the form. The controls URL allows the website to specify controls for the secure form to allow some interaction with the sandboxed form e.g. adding or removing form fields. The website can directly interact with the controls.

When a user opens a secure form, the browser extension creates an overlay with two separate frames, the controls and the secure form. When the form is submitted, the form data is MIME encoded and then PGP encrypted with the user's choice of public keys. We choose to MIME encode the forms, as we want `SecureForm` to be a feasible way to implement encrypted webmail. Once the website receives the encrypted data, it can send the encrypted data to the web server. This ensures that both the website and web server are only receiving encrypted data.

The website can then specify where it wants to display the encrypted data by using our custom `<encrypted>` tag. If the website embeds an `<encrypted>` tag, the encrypted data is fetched from the server. However, to view the decrypted data, the user needs to authenticate himself by providing a password associated with his private keys to the extension. If the user enters the correct password, his private key is unlocked and the encrypted data are decrypted. The data are then MIME decoded into JSON objects. Our browser extension creates an iframe and then parses the JSON objects into HTML content. The content is appended to the iframe running in the extension's context and displayed to the user. Because of the same-origin policy, the website is neither able to read the decrypted text nor access files present in the encrypted data in the display iframe. Figure 1 summarizes the encryption and decryption workflow with `SecureForms`.

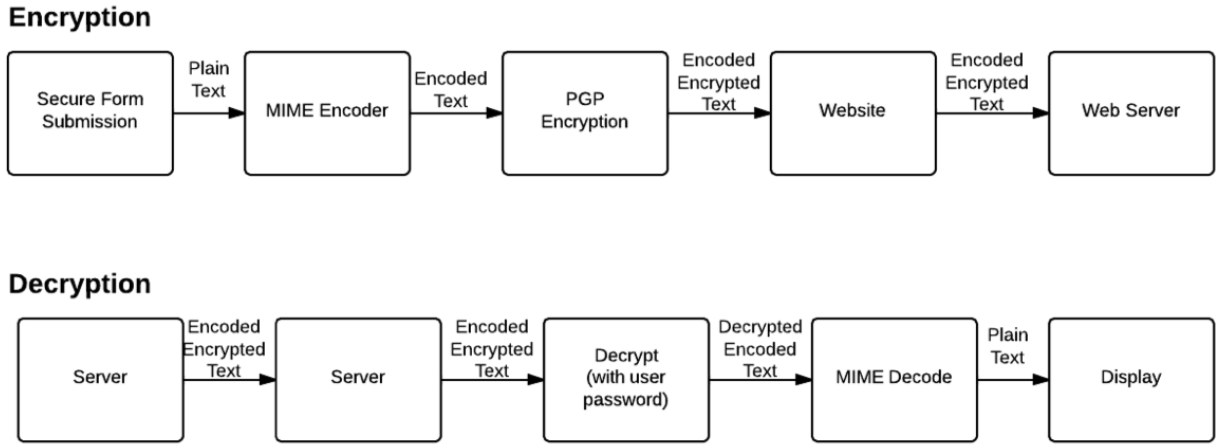

Figure1: The above figure shows the control flow for the encryption and decryption using the Firefox browser extension.

3. Security

3.1 Padding Data

In addition to plain text inputs and file uploads, our extension supports radio buttons, checkboxes, and selection menus. Given that the website knows the values of these input types, an attacker could discern what a user has selected by the length of the encrypted data, particularly if a form contained no variable-length input fields. In order to avoid this type of security attack, the browser extension pads these types of selections with whitespace before encrypting it. The encrypted text always has the maximum length possible for the form, disregarding the variable-length inputs.

3.2 Imitation Forms

An attacker could attempt to imitate our secure form and phish for user data. In order to prevent this, when a user opens our secure form, the browser will display the word `Secure` on the add-on bar, informing the user that they are in a secure form. In addition, the various components of our overlay have colored borders that indicate whether the website or browser specified the component. We draw a green border around browser specified components (encrypt/decrypt dialogs), a blue border around website specified components (the secure form), and an orange border around website controlled components (the controls).

3.3 Stealing User Input

In order to prevent an attacker from adding listeners to click events and retrieving which radio buttons or checkboxes the user has chosen, we shadow the website form and overlay the window with our `SecureForm` document that an attacker cannot retrieve or interact with. The `SecureForm` is sandboxed, so the adversary cannot listen to any events on the secure form itself. Therefore the adversary also cannot tell what the user types into the form.

3.4 PGP Keys and Key Ring

If the server is compromised by an adversary, he will not be able to decrypt the submitted form data, as that will require the adversary to know the user's private keys. We assume that the browser is trusted and thus the key ring cannot be compromised. We actually must trust the browser, since once the private key is unlocked, it remains unlocked for the duration of the browsing session. The browser can technically then navigate through objects to access the user's private key once it is unlocked.

3.5 Shadowing DOM

To ensure that the website is not able to read the contents of the form specified by the form URL of the `SecureForm`, we shadow the provided website's (slave) form to generate a secure (master) version of the form. We use a `MutationObserver` on the slave form to listen to all DOM changes and replicate these changes on the master form. When replicating changes, we remove `<script>`, `<style>`, `<link>`, and `<a>` tags, since these tags would allow the website to insert arbitrary scripts, stylesheets, or links that it could use to read or infer the contents of the form or intercept click events. We disable JavaScript on the master form, so that JavaScript attached as attributes also cannot run. As shown in Figure 2, the user will interact with the master form, and the website can only interact directly with the slave form. This allows the website to insert, modify, and delete form elements without being able to read the contents of the form.

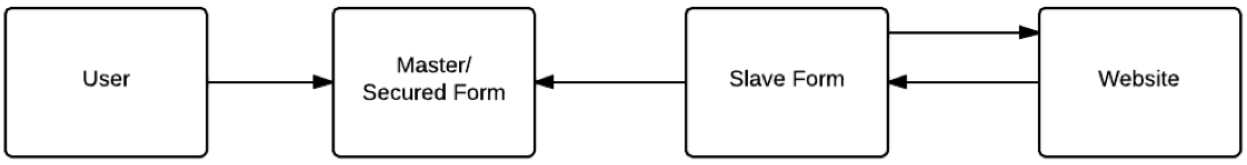

Figure 2: The above figure displays the interaction between the website, slave form, and master (secured) form.

4. Use Cases

We believe that once the implementation is refined, our browser extension has many uses. It can easily be integrated for use with dead drops and Dropbox, allowing users to secure their files from untrusted websites. Because we implement MIME encoding, `SecureForms` can be used to implemented web email clients. We hope such client-side encryption will become part of popular webmail clients, such as Gmail and Hotmail. Many times, users also worry about their privacy when taking surveys or questionnaires, so this extension is directly applicable to surveys as well. The uses for an easy to use and adaptable browser extension that provides client-side encryption are extensive. We leave it to developers to come up with further use cases.

5. Future Work/Known Issues

5.1 Authenticity and Integrity

Our current implementation only provides privacy and neither provides authenticity nor integrity. We guarantee a malicious website or web server cannot tamper with the encrypted data, but the user has no way to know who sent the data and whether or not what he received is what was actually sent. We can provide authenticity and integrity by implementing PGP signatures.

5.2 User Interface

For this system to be deployable in the real world, we would require a better user interface. In our prototype, it is extremely easy for a user to not realize whether or not he is using a secure form. If the user closes his add-on bar, he would have no way of discerning whether or not the page he is visiting is secure. (Ideally, the user will not close the add-on bar, since the link to the key ring management page is only on the add-on bar.)

We currently only support plaintext input. Providing a rich text editor would be useful for websites that implement an email client.

5.3 Display Options

In the future, the website should be able to affect the display output in a meaningful manner, assuming the website knows something about the format of the data. If the website knows which input fields exist in the form, it should be able to select which fields it wants to display and choose where it wants to display them.

We currently also only support display in plaintext, but in the future, HTML display is again required for reasonable display of webmail.

5.4 Key Ring Improvements

Our key ring in its present state is very basic and only allows the user to add keys. In the future, we want provide better key ring management by allowing the user to delete keys and search for keys.

The encryption/decryption functionality also needs to be improved in order for it to work with older PGP key formats.

6. Conclusion

Our Firefox extension guarantees privacy for any arbitrary input objects, including forms, emails and input fields. By sandboxing the form in a secured environment, we ensure that an adversary that compromises the website or the web server will not be able to intercept data inputted by the user. We believe that the extension is useful for many real-life applications, as we demonstrate in our test code. It can be extended to provide integrity and authentication in the future.

7. References

1. http://www.senditonthenet.com/auditing
2. http://ajaxpatterns.org/Host-Proof_Hosting
3. Raluca Ada Popa, Emily Stark, Steven Valdez, Jonas Helfer, Nickolai Zeldovich, and Hari Balakrishnan. *Securing web applications by blindfolding the server* http://es.csail.mit.edu/mylar-f6d8571c.pdf

Appendix

We demonstrate use of `SecureForm`s for a course submission website. A website can easily initialize `SecureForm` objects as follows:

```
var form = SecureForm({
  formsrc: "/path/to/my/form.html",
  [controlssrc: "/path/to/my/controls.html"]
});
```

And specify the behavior on submit by defining an event listener:

```
form.addEventListener("submit", function() {
  alert("You submitted: \n" + this.value);
});
```

To display the form, the website just needs to call:

```
form.show();
```

To display a PGP encrypted MIME encoded message, a website needs to place the encrypted tag specifying the source as follows:

```
<encrypted src="/path/to/armored-message.pgp"></encrypted>
```

The screenshots below demonstrate control flow for a website that employs a `SecureForm` object. We also include figures to illustrate the functionality of our key ring implementation.

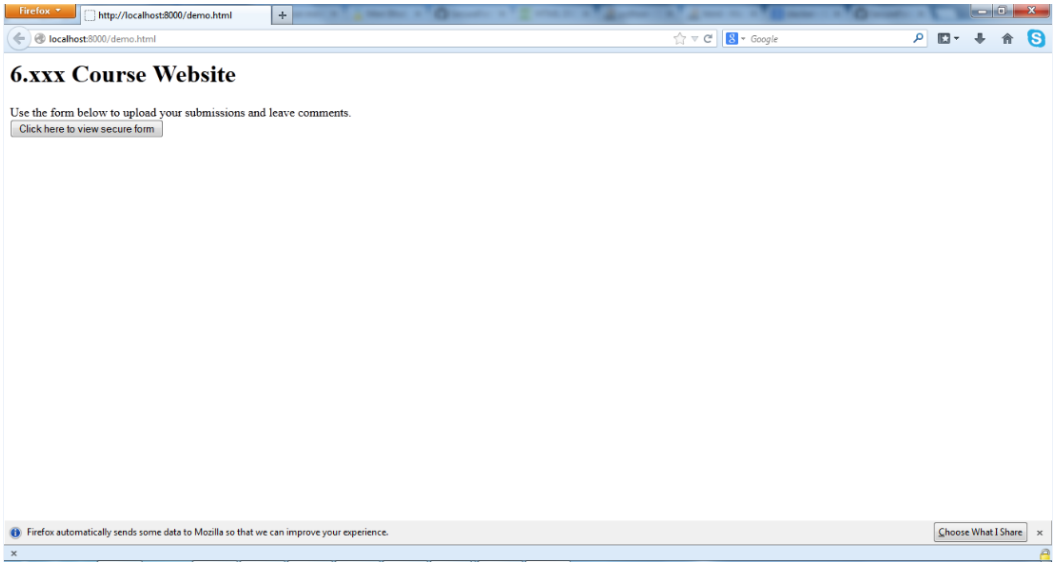

Figure 3: Our extension can secure a course website that allows students to upload their submissions securely. The user can click to view the SecureForm.

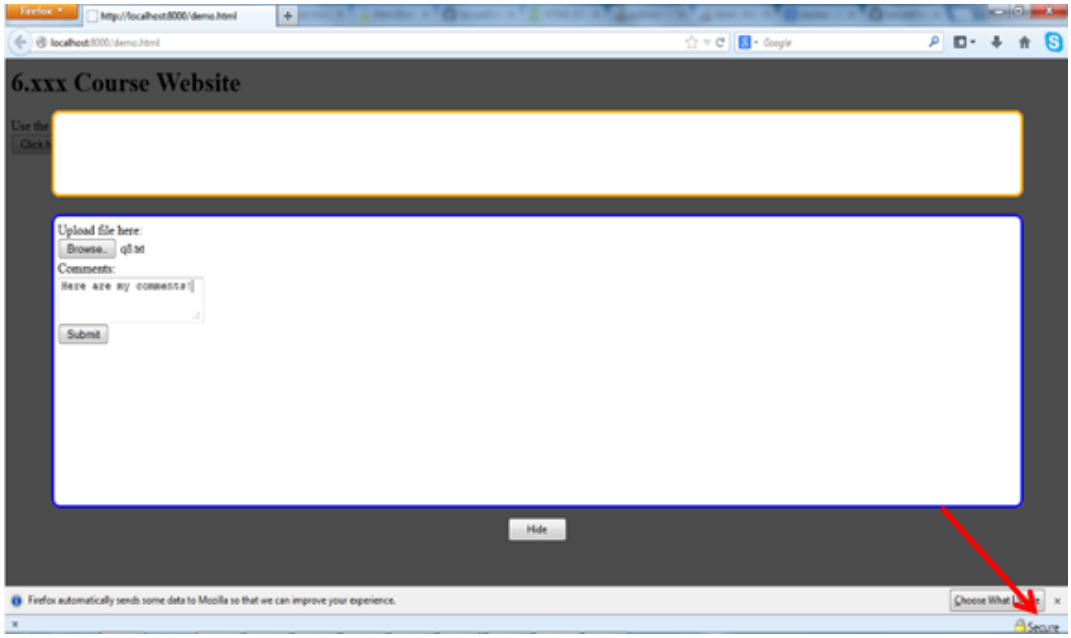

Figure 4: The figure shows the SecureForm overlay. The controls section is on top and the secure form is in the bottom iframe. Note that the control and form windows are bordered with orange and blue colors, respectively, and that the browser shows “Secure” at the the bottom right to indicate the user that form is secured under our extension.

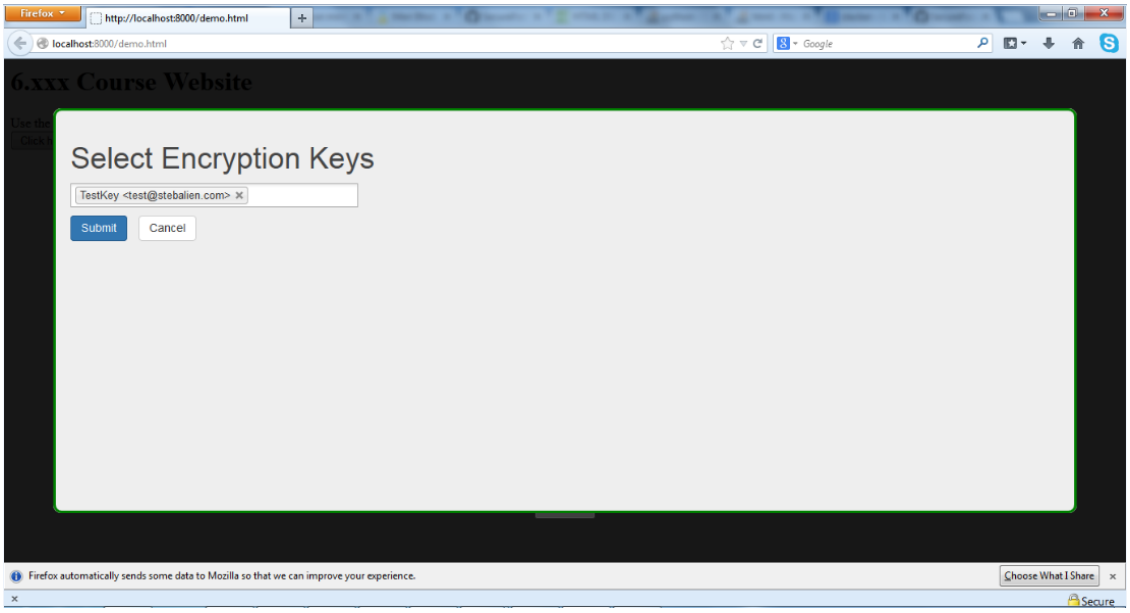

Figure 5: Once the user submits a form, he will be asked to select PGP keys, stored in his key ring. The form data are encrypted with these keys and then returned to the website.

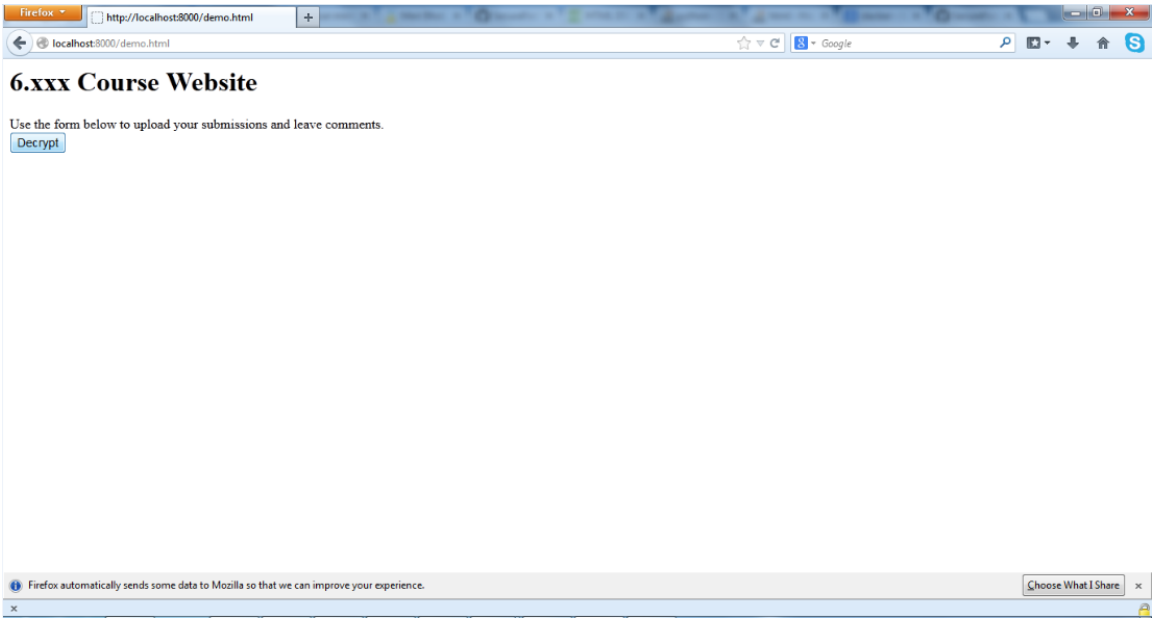

Figure 6: The user is shown the option to decrypt and view his submitted form.

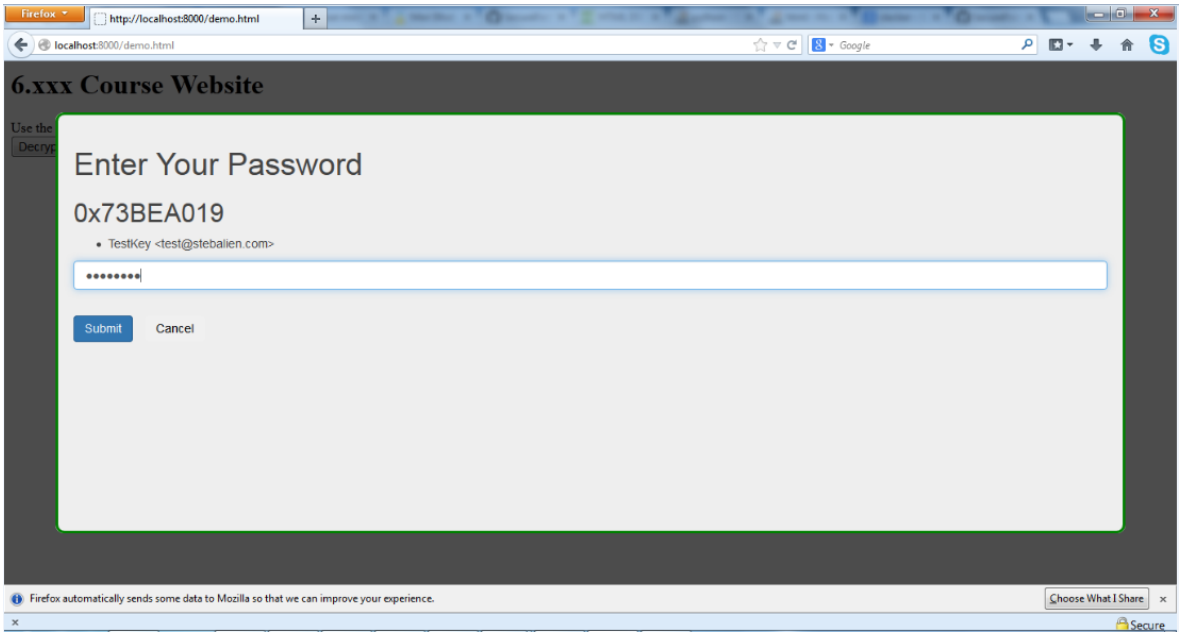

Figure 7: When the user chooses to view the decrypted form, our extension prompts the user for his password to unlock his private key. The private key is then used to decrypt the SecureForm.

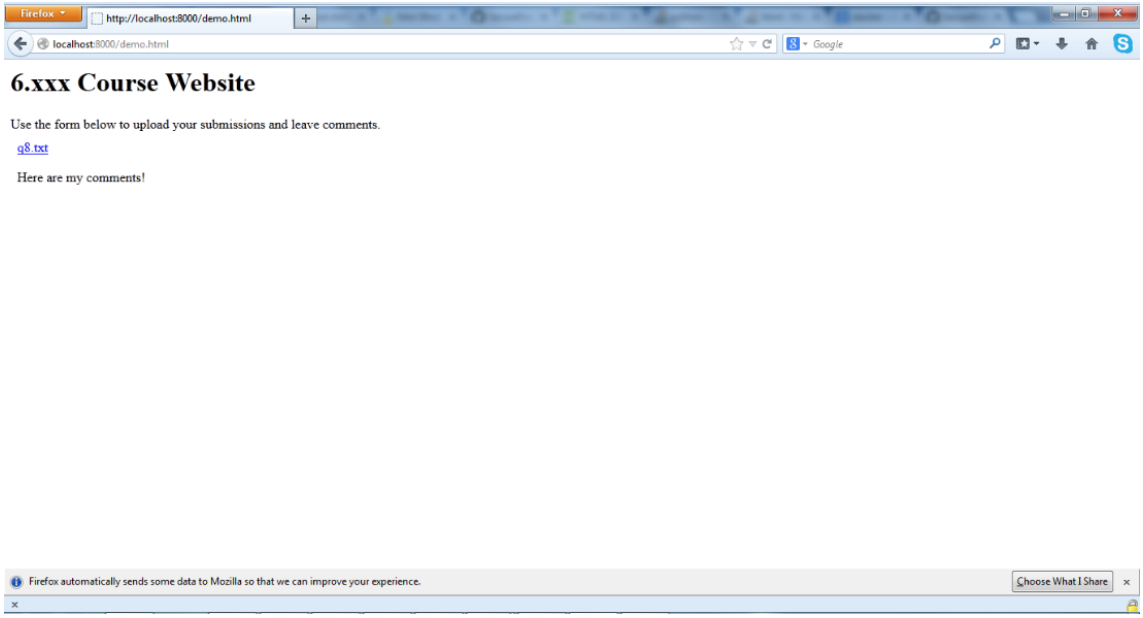

Figure 8: The form is decrypted and MIME decoded, and the contents are displayed to the user.

The user can store his PGP public and private keys in the keyring as shown below:

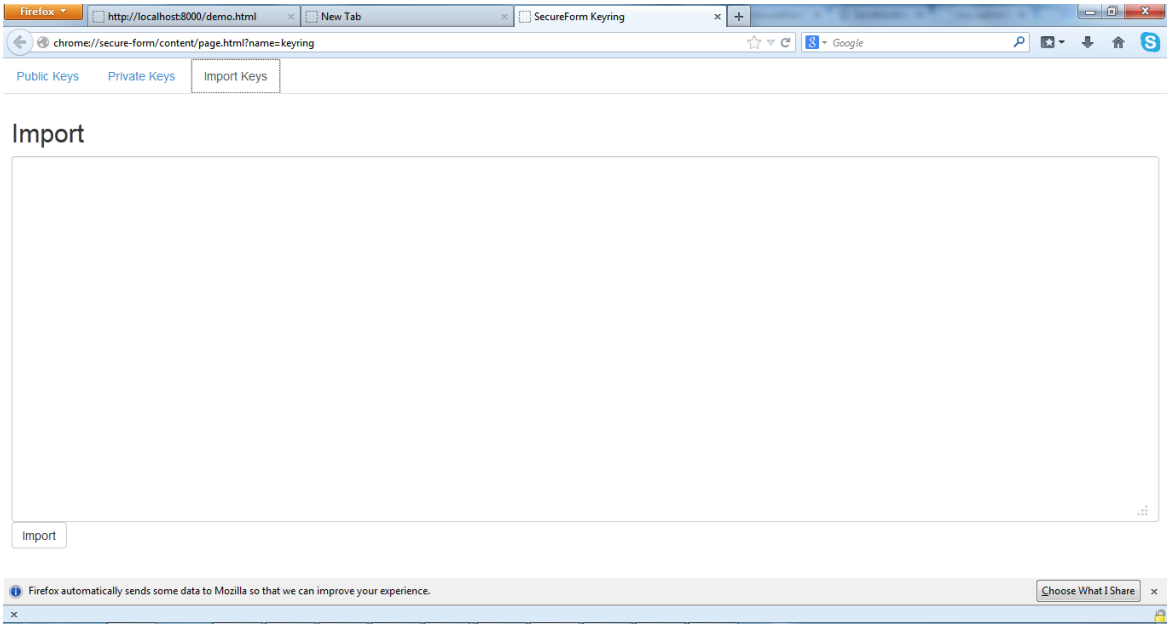

Figure 9: The user can import his private and public keys that will be securely stored in his key ring.

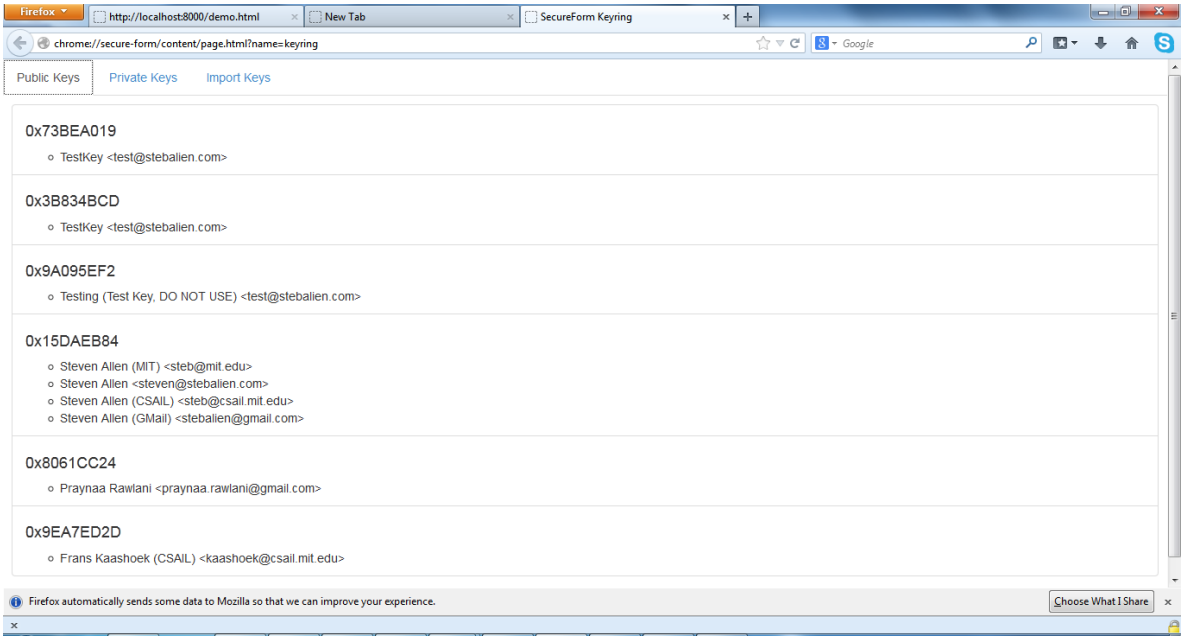

Figure 10: The user can see his private and public keys stored.