CS 461 / 462 / 463 - Requirements Document

Project Name: PGP for the Web
Team Name: Pretty Good Project (PGP)
Team Members:
Daniel Reichert <reicherd@onid.orst.edu
Trevor Bramwell <bramwelt@onid.orst.edu>
Geoffery Corey <coreyg@onid.orst.edu>
Client/Sponsor/Mentor: Sean McGregor <mcgregse@onid.orst.edu> - Privly Foundation

Introduction to the problem:

Recent work by a W3 Working Group plans to expose many powerful cryptographic operations for web applications. Although the planned API adds much needed functionality to JavaScript, it doesn't address the terrible security properties of JavaScript's runtime. For instance, any script running in the web application has the power to hijack the web app's content and UX. In February, 2013, a mistake in Facebook's "like" button brought down millions of web sites. Further, the trust model for serving remote code is fundamentally broken, since you can't serve JavaScript code for doing cryptography without making the server a potential attack point. The application stack known as Privly, which has been developed primarily by current and past OSU students, corrects many of these insecure properties of JavaScript. This project aims to build on Privly's new foundation with Pretty Good Privacy (PGP), a tried and true standard for encrypted communication.

Project Description:

The problem. Every time you visit a website you're running someone else's code. Privly will fix this by allowing users to run and share encrypted applications.
The solution. Create a PGP application and key management system for Privly. Our starting ground is the preliminary research done on PGP implementations for the web, and an incomplete proof of concept application.
The success criteria: A cryptographically secure email exchange between two parties using Privly's implementation of PGP, the Privly extension, Privly key manager, and an insecure webmail provider.
The cryptographic operations are being performed using the openpgpjs library. Long term this will not be the case. Instead, a transition to the W3C WebCrypto standard is planned. The W3C standard is not currently finalized, which is why we are using openpgpjs in the interim.
The key management component is the bulk of this project. Since a key management system has not been designed in this context before, a great deal of care and consideration must be put into the architecture and implementation. This is especially so because the security of the entire system greatly depends on the key management being robust.

Requirements:

1) A fully working application that allows people to send and receive PGP encrypted data via any site on the web.
2) A key management facility that allows for the user to publish keys to key servers, as well as import the public keys of other users.

- 2a) The system to import public keys of others will be designed such that it is fluid and automatic. This system will give up 100% guaranteed authentication in favor of ease of use.
- 2b) The fluid authentication system will have options available to enable greater levels of trust in authentication for users that wish to pursue this option.
- 3) Full test coverage of all the implemented code.
- PGP has a lot of ciphers built in, initially supported ciphers should include:
- For hashing, SHA.
- For asymmetric encryption, RSA.
- For symmetric encryption AES.
- With more ciphers being supported as icing on the cake, or implemented by open source contributors.
- Interactions with the openpgp.js library should be made with an approach similar to the W3C WebCrypto standard (http://www.w3.org/TR/WebCryptoAPI/) to ease transitioning to it after it is formally released.
- "Two students will begin by implementing a proof-of-concept application for handling content using a hard-coded public/private keypair. One Student will work on implementation, while the other works on maintaining full test coverage. While the two students work on the functions of the application, a third student will work on key management, including importation of public keys from various sources. Once the core application is developed, the students will focus on documenting their work so others may audit the security of the application and potentially extend it for different application types. Upon completion of the project, other developers would be able to extend the work for more dynamic applications including data visualization, images, and video"

Versions:

- *One way to mitigate risk is by releasing your software in different versions. What functionality will be grouped into what version numbers?*
- Version 0.1: Create new.html and show.html that encrypts/decrypts content with a hard coded key.
- Version 0.2: Define API for PGP functionality and implement this in skeleton form.
- Version 0.3: Write key_manager.js, which manages local key database.
- Version 0.4: Determine if we are using a good source of entropy/PRNG. Write functionality to warn the user if they are operating in an unsafe setting.
- Version 0.5: Allowing the importation of existing private keys - this facilitates a more usable experience for someone with multiple devices.
- Version 0.6: Key revocation system

Design:

- The project will be a privly application that supports PGP. The application will be a wrapper that provides PGP for other privly applications. For example, the existing PlainPost application could be wrapped in the PGP app and now support encryption and signing of messages. This will mean designing an API that is available to privly applications.
- The largest pieces of functionality are symmetric and asymmetric encryption, hashing, and interfacing with the established public key infrastructure.
- QR Codes can be used to to transfer private keys from one device to another. Note, the maximum size of a QR string is 2953 Bytes or 4296 Characters.

- Add the picture that shows the trusted and untrusted contexts of the privly extension, host page, and injected content. (Emailed to trevor)
- Diagrams to create:
- Flow of creating, encrypting, storing, retrieving, and decrypting data.
- Fluid key manager
- Picking up keys found in the wild
- Creating your own key string to share
- Web of trust management
- Pushing to key servers
- Revoking keys
- Establishing trust of keys in key manager
- Integration with privly application infrastructure
- Use of trust in other people's keys
- Block Diagram Components:
- Content Server
- "Cloud"
- Key Manager

Specific tasks to be undertaken:

- Tasks:
- Sending and Reading Encrypted messages with hard coded keys
- Sending Encrypted message
- Encrypting a message
- Pushing ciphertext to a server
- Reading Encrypted message
- Retrieving a ciphertext from a server
- Decrypting a ciphertext

Risk Assessment:

- Failing to come up with a way to manage private keys that requires no user action. Using a model similar to mozilla persona and the BrowserID protocol may mitigate some issues that could crop up.
- The chance that the underlying openpgp.js API we use changes radically. There really is no way to mitigate this risk, but by paying close attention to the openpgp.js mailing list, we can watch out for changes and potential bugs in new releases. If the API does change radically, we'll adjust our coding to match the new API and adjust our tests to account for the changes and to avoid any regressions in our or their software.

Testing:

- Unit tests and integration tests will be written with the goal of full test coverage.
- Tests will be written in Jasmine since it is the established testing framework for Privly, and in wide use by JavaScript developers. We will be following TTD practices. As the number of tests grow we will separate tests into quick and not-quick tests so that we will continue to run tests continuously during development.
- Integration Test Example: Spin up headless browser, install plugin, write encrypted message with application as User1, assert message is in cipher text, Switch to User2 and assert the message can be read in plain text.

- We will also be running against priv.ly's integration tests to make sure we stay compatible with their API.

Preliminary Timetable:
- Concurrent tasks:
- Design key management system
- generate
- view
- import
- Download public keys from the PGP public key servers
- Import public key from privly created identity URL for fluid trust system
- Import existing private keys created outside of the privly pgp app, or from other devices
- export
- Allow you to export public keys to interface with existing public key infrastructure.
- Create QR code system to facilitate transfer of private keys between devices
- Creation of identity URL for fluid trust system
- delete
- Remove keys from local key ring.
- Untrust previously trusted keys
- Delete private key
- Note: there should be a large hurdle when deleting a private key as a precaution against user mistake.
- send
- Interface with current PGP key servers to upload keys to the server
- receive
- revoke
- Revoke key in PKI
- verify (should we be re-implementing web of trust?)
- Allow a user to check validity of a key and sign it.
- find
- Locate keys on websites and import auto-magically
- Implement hard coded keys exchange
- Perform actions of:
- encrypting a message
- transfer ciphertext to a server
- retrieve ciphertext
- decrypt ciphertext
- See attached Gantt chart for timeline.

Roles of the different team members:
- Roles:
- Design key management system
- Trevor and Geoffrey
- Implement hard coded keys exchange
- Daniel

Integration Plan:

- Create a mock api and code against the API. Also create tests beforehand and code against those tests. Should the API requirements need to change, this will be communicated to the other team members as soon as possible.

Dataflow sequence diagram:

- Attached

User interface requirements:

- The user interface should be non-existent to the furthest extent possible. Some interface may be required. It should be as simple and intuitive as possible.

References:

- Papers, Books, or Websites on security/PGP:
- https://www.schneier.com/
- http://thoughtcrime.org/blog/
- http://www.philzimmermann.com/
- OpenPGP.js: https://github.com/openpgpjs/openpgpjs/
- RFC4880: https://tools.ietf.org/html/rfc4880
- RFC5581: https://tools.ietf.org/html/rfc5581

Glossary:

- *If useful. Are there terms that are used in this Project Description that are not generally known?*
- Privly Application: A privly application is an application built into the Privly extension that provides a specific piece of functionality that is not core to the operation of the extension.