CS6238 Secure Computer Systems
Fall 2015
Project 2: Secure Shared Data Repository (S2DR)
Deadline: 12/4/2015 (11:55pm)

Goals & Assumptions

The goal of this project is to explore how distributed services can be secured. You will design and develop a simple *Secure Shared Document Repository* (S2DR) service that allows storage and retrieval of documents created by its multiple users who access the documents at their local machines (S2DR may be implemented in a Cloud). Thus, the system consists of S2DR clients and a server that stores the documents. The server also facilitates sharing of documents between clients as allowed by the owners of the documents. A document's owner is the client that creates the document unless ownership has been delegated to another client. In case of delegation of ownership, multiple clients can have the rights that are available to an owner.

To secure communication between clients and the server, and authenticate sources of requests, we will assume that certificates are available for this purpose. Thus, to implement such a system, you will need to set up a Certificate Authority (CA) that generates certificates for clients and the server. All nodes trust the CA. You can make use of a library such as **OpenSSL** (see resources section) for setting up the CA and to generate certificates.

S2DR Implementation Details

After an S2DR server starts, a client can make the following calls. You can assume that clients have a discovery service that allows them to find the hostname where S2DR runs.

1. **init-session(*hostname*)**: A new secure session is started with the server running at host hostname. Mutual authentication is performed, and a secure communication channel is established between the client executing this call and the server.

2. **check_out(Document UID)**: After a session is established, a client can request a document over the secure channel from the server using this call. A request coming over a session is honored only if it is for a document owned by the client that set up the session, or this client has a valid delegation (see delegate call). If successful, a local copy of the document data is made available to the client. The server must maintain information about documents (e.g., meta-data) that allows it to locate the requested document, decrypt it and send the data to the requestor.

3. **check_in(Document UID, SecurityFlag)**: A document is sent to the server over the secure channel that was established when the session was initiated. If the document already exists on the server, you may overwrite it along with its meta-data. If a new document is checked in, this client becomes the owner of the document. If the client is able to update because of a delegation, the owner does not change. You need to use some scheme to ensure that documents created by different clients have unique UIDs. The **SecurityFlag** specifies how document data should be stored on the server (see next page).

4. **delegate(Document UID, client C, time T, permission, PropagationFlag)**: A delegation credential (e.g., signed token) is generated that allows an owner client to delegate permissions (checkin, checkout or both, or owner) for a document to another client C for a time duration of T. If C is ALL, the delegation is made to any client in the system for this document. If you use a protocol that needs to securely exchange messages between clients for implementing delegation, you should use secure channels. **PropagationFlag** is a Boolean that specifies if C can further propagate the rights delegated to it. A true value permits delegation and false disallows it. Propagated delegation shouldn't be valid if the original delegation has expired.

5. **safe_delete(Document UID)**: If the requesting client has appropriate rights, the file is safely deleted and no one in the future should be able to access data contained in it even if the server gets compromised.

6. **terminate-ssession()**: Terminates the current session. If any documents that were received from the server are updated, their new copies must be sent to the server before session termination completes.

The S2DR server stores data in encrypted form when **SecurityFlag** is **CONFIDENTIALITY**. It generates a random AES document encryption key. To decrypt the data at a later time, this key is also encrypted using the server's public key and stored with document meta-data. When a request comes for such a document, the server locates its key and data, decrypts the data and sends it back over the secure channel. If **SecurityFlag** is **INTEGRITY**, the server stores a copy signed with the document key, and it also checks the signature before sending a copy to a client. Neither encryption nor signing is necessary when the **SecurityFlag** is **NONE**. Deletion of a confidential document should result in permanent destruction of the key used to encrypt it.

This project should be implemented on Linux/Unix using C/C++/Java/Python/Node.js/PHP for programming and you can work in groups of two. Submissions should be made via T-square by the team-leader/point of contact. Since this is a security class, you should use secure coding practices. You are also expected to use static code analysis tools like flawfinder, and splint (Findbugs, PMD for Java) and minimize the use of unsafe function calls (Justify any such calls you make by providing inline comments). The report should list tools you used to check your code is vulnerability free. The report should also discuss the threat model and what threats are handled by your implementation. You should also include instructions for testing your implementation in the report.

Project Deliverables

The project deliverables include S2DR code and a report. In the report, state the protocols that you utilize to implement the interactions between a client and the server. In particular, define the message formats for requests-responses and the steps taken to validate requests and delegation tokens. Required deliverables include:

- The source code along with a Makefile/Ant script for building your program. (5 points)
- A README file explaining how to run the program, along with instructions for compiling and testing your application. Specifically, include scripts/instructions for testing the following modules of your implementation. Note that some of the following operations should fail in a correctly implemented system.

    1. Initialization of the CA using the server. You need to provide script to generate certificates for clients. Client name should be provided as argument for your CA management tool. You could assume the name of the client is used as UID. Once a client is created, you need to provide a workspace for that client. The workspace (a directory) should contains the client program and the certificate and keys of that client. The client program should use the certificate to communicate securely with the server program. Clients should be operated in separate workspaces. (10 points)

    2. Checking-in with INTEGRITY SecurityFlag. (10 points)

        1. Initialize a session with the server as the first client (let's say, client_0) and check in a document "0.txt" with INTEGRITY SecurityFlag. (For simplicity, you could use the name of the document as the Document UID). The name of the document should be provided as argument of your check_in() implementation.

        2. Show where the checked-in document and its signature will be located at the server side. Provide a script for verifying the signature of the copy.

    3. Checking-out a document as its owner. (5 points)

        1. Using the same session, check out the document you just uploaded to the server and store it as "0_copy.txt".

    4. Checking-out a document as a user without the access permission. (10 points)

    1. Initialize a session with the server as the second client (client_1) and check out the document “0.txt” and store it as “0_copy.txt”.

5. Safe-deletion. (5 points)

    1. Using the first session, safely delete “0.txt” then attempt to check it out again and store as “0_copy2.txt”.

6. Checking-in and checking-out with CONFIDENTIALITY SecurityFlag. (10 points)

    1. Using the first session, check in a second document “1.txt” with CONFIDENTIALITY SecurityFlag. Show where the server copy of “1.txt” is located. The copy needs to be encrypted with a random AES key that is encrypted using server’s public key.

    2. Using the same session, check out the second document you just uploaded to the server and store it as “1_copy.txt”.

    3. Terminate the first session.

7. Updating a document. (10 points)

    1. Restart the first session, check in the second document “1.txt” with CONFIDENTIALITY|INTEGRITY SecurityFlag. Show where the server copy of “1.txt” and its signature are located. The copy needs to be encrypted with a different AES key that is encrypted using server’s public key. Use the same script to verify the signature of the encrypted copy.

    2. Check out “1.txt” and store it as “1_copy2.txt”.

8. Checking-in & checking-out delegation without propagation. (10 points)

    1. Using the first session, delegate(“1.txt”, “client_1”, 30, checking-in | checking-out, False). The delegation timeout should be 30 seconds.

    2. Using the second session (client_1), check out “1.txt” and store it as “1_copy.txt”.

    3. Using the second session, check in a different text file as “1.txt”.

    4. Using the second session, delegate(“1.txt”, “client_2”, 30, checking-in | checking-out, False). The delegation timeout should be 30 seconds.

    5. Initialize a session with the server as the third client (client_2), check out the document “1.txt” and store it as “1_copy.txt”.

    6. After the delegation expires, using the second session (client_1), check out “1.txt” and store it as “1_copy2.txt”.

    7. Using the first session, check out “1.txt” and store it as “1_copy3.txt”.

9. Checking-out delegation with propagation. (10 points)

    1. Using the first session, delegate(“1.txt”, “client_1”, 30, checking-out, True). The delegation timeout should be 30 seconds.

    2. Using the second session, delegate(“1.txt”, “client_2”, 60, checking-out, False). The delegation timeout should be 60 seconds.

    3. Using the third session (client_2), check out “1.txt” and store it as “1_copy2.txt”.

4. After the 30-second delegation made by the owner expires, using the third session (client_2), check out “1.txt” and store it as “1_copy3.txt”.

5. After the 60-second delegation made by the second client expires, using the third session (client_2), check out “1.txt” and store it as “1_copy4.txt”.

6. Terminate all sessions.

- A PDF report for your application that includes protocol details of S2DR and the security analysis that is required above. Also, if any features are not implemented or tested, highlight them in the report. The report should also briefly describe individual contributions of each member – write about who did what (function/feature implementation, integration, design, evaluation etc.) (15 points)

Notes

- All files should be submitted in tarball or zip archive.
- You should use secure communication channel between client(s) and server. Mutual authentication should be performed. This suggests that you can’t simply reuse the common authentication mechanisms in web applications, which normally only offer authentication of the web server.

Resources / Links:

1. OpenSSL: http://www.openssl.org/
2. Secure programming with the OpenSSL API: http://www.ibm.com/developerworks/linux/library/l-openssl.html
3. Java SE Security page: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html