securityandtrust.lu

Towards a more secure and usable cloud storage for e-Health

Sankalp Ghatpande, University of Luxembourg
Gabriela Gheorghe, SnT, University of Luxembourg

31 August 2015

ISBN: 978-2-87971-165-2

1 Introduction

e-Health is a "system defined as the healthcare practice supported by electronic processes and communication"[23]. There are numerous high quality e-health platforms available like Mediboard [26], THIRRA[25], that aim to provide access to available heath related data which can be accessible from anywhere. These systems allow medical users to create medical cases for individual patients and share them with other medical users for consultation.

Many of the currently existing e-Health systems offer easy storage and access to health record from virtually any device [2]. Additionally they allow to explicitly share specific information in the health record, or the complete information with different users or groups. For example, the Microsoft HealthVault [24] allows access to either the complete record or part of the record by another user if authorization has been granted. This would allow medical professionals to share patient records securely with other professionals. Figure A.1 in Appendix shows an example how a medical professional accesses the medical record in an e-Health system.

e-Health systems store health-related data to allow development of an effective medical treatment plan. However there is the important issue of protecting the confidentiality of patients as any disclosure may result in personal and financial losses as well as violation of data protection laws such as the European Union's Directive 2011/24/EU on patients' rights in cross-border healthcare[3] or HiPAA[4].

We propose and analyse a system for access to these records that allows the data to be encrypted in the web browser. Using the web browser makes it possible to access the data from any device connected to internet. This data cannot be decrypted on the server or by the anyone at the data storage centre but can only be decrypted by the end users. Additionally, our system also allows for changing the encryption key without violating the confidentiality of the data, which we haven't seen in any other system.

2 Security and Requirements

2.1 Threat Model

The proposed security model should allow the user access the secured data using only the self-chosen password, which means the most vulnerable point of this model is the server which stores the wrapped key of the user derived key. In general, attacks can be categorized into a few different groups:

- Network Eavesdropping
  All the connections from server to device are assumed to be using SSL/TLS.

Since all connections will be over SSL/TLS, the channel would be protected against passive eavesdroppers.

- Man-in-the-middle (MITM)
  Using SSL/TLS does not prevent against active adversaries. Active adversaries can forge an identity and also tamper with the messages passed between the user and server. Since the user device may not have adequate information to establish to identity of the server, this would lead to the adversary being able to read and even change the communication exchanges between server and user by impersonating the legitimate server or vice-versa.

- Storage Server
  The biggest threat is the cloud/server that holds the Key-Cipher (KC) along with other information such as user-id and the associated cryptographic data, both of which can be used by potential adversary to derive the user-password and thus by extension the derived key. This type of access to server can only be possible for IT administrators, someone who has successfully managed to access it or the back-up copy of the data.

- Social Aspect
  Another threat for the system is the human factor i.e. users or admins of the server who can be forced/blackmailed to hand over the copy of data or of the backup disk.

Ideally, in the model, these attacks are only possible when the adversary has knowledge of the parameters of the key derivation function, the salt and a pair of ciphertext and its corresponding plaintext, which itself is a highly impossible task. Nevertheless when evaluating the security of the proposed model, it is being assumed that the adversary had already completed this impossible task.

2.2 Requirements

We define some security requirements that needs to be satisfied by an ideal e-Health system, namely:

1. Confidentiality: All the data must be kept absolutely secure from all adversaries including the server itself.

2. Integrity: The operation of encryption and decryption on the data should be possible without requiring to relay on any third party.

3. Authorized sharing: Data shared by one user must be read only by users that are explicitly authorized.

4. Usability: The security of the system must not decrease the usability of the system. It should have the functionality of recovering forgotten password or changing of password without violating other requirements of the system.

3 Related Work

The medical data is always encrypted on the user's device using a password known by the user, while the server only acts as an encrypted data storage space. All the operations of the system take place on the user end, which performs all encryption and decryption, sending the encrypted data to server and sharing it with other users or groups only when permitted.

The server is never able to read the decrypted data; even if an adversary gets access to server, he cannot recover the plaintext without big computational efforts to brute-force through every user's possible decryption key. In order to understand the current implementation of similar systems, we studied some of the current secure storage-based applications. These applications/systems were categorized under two sections: cloud-based encryption storage and electronic health record systems.

1. Cloud-based encryption storage services like SpiderOak[32], which offers a remote encrypted backup folder which is synchronized across all of the user's registered devices. The user may explicitly share specific files with other users or groups via web link. Application such as BoxCryptor[21] and Truecrypt[33] add client-side encryption which is then hosted on storage services such as Dropbox[40], iCloud[41] and many more [22]. These systems are not designed for the EHR but claim to be very secure. However, the security require placing of trust on the services.

2. Electronic Health record system like Microsoft HealtVault[24], which offer a easy storage and access for health record from virtually any device. The health-professionals using it may explicitly share specific information in the heath record, or the complete record with different users or group by giving them appropriate permissions. Systems like these fall short of the security requirements defined in earlier section.

The Table A.2 in appendix, shows a brief summary of the features implemented by a number of applications. In particular, it also notes the cryptographic algorithms and mechanisms used. We noticed during the study that while many of the applications provide adequate security using either symmetric or asymmetric system(PKI) they relay heavily on trusting a third-party for cryptographic key-management.

4 Proposed Solution

For a system to satisfy the confidentiality requirement, all the operations on the system must take place on the end device. These operations include changing, adding and securing the data before being sent to the storage server. This requires that the encryption key be used on the device without having the need to store the said key. A solution to this requires a key stretching algorithm which

allows users to use passwords they are comfortable with without any restrictions.

There are many key stretching algorithms including scrypt[9], bycrypt[10] and PBKDF2[8] which output an arbitrary length key based on the password defined by the user. Due to the inherent nature of key stretching functions, even a smaller password (e.g. 4 character) would keep the encrypted data secure. Having smaller password has the additional benefit of being easy to remember and reduces the risk of the user having to memorize complex passwords.

As the system is expected to be accessed directly from the browser of any device, it must not require a high amount of computation; thus we recommend using PBKDF2 which is relatively cheap in terms of computation and memory with respect to others. One important parameter of PBKDF2 is the salt, using which prevents generation of the same key for the same user passwords. In order to have strong derived key, the salt is to be generated on the server using a Cryptographic Random Number Generator (CRNG) and stored on the server. The reason for generation of the salt on server lies in the open question about the performance of CRNG over current mobile platforms. This salt will be sent back to the user device whenever the user attempts to login and must be unique for each user.

Using the PBKDF2 derived output as key for symmetric cipher such as AES the data can be securely encrypted. Other ciphers can also be used in-place of AES including asymmetric or hybrid schemes depending on the targeted device. The key size of the derived PBKDF2 output is recommended to be minimum of 128-bits. Using 128-bits has been proven to be as secure as using 192 or 256-bits [17].

Authentication of the user is also an important aspect to distinguish between legitimate and illegitimate users. Requiring the server to generate and store a unique cryptographic salt for each user is one of the effective and feasible ways to ensure the authentication of the user at a moment and also in the future.

It is known that mistyped passwords by users could potentially make the system decrypt the data into gibberish. A problem here is that users might fail to notice this and update or enter some information, encrypt it using a wrong password and save it to the database. This would potentially render all the data useless. Such scenario may happen during an emergency health situation, where the medical user might create new data using wrong password, potentially rendering the earlier data useless without noticing. It is not possible for the encryption key to be available on the server for verification, hence a cryptographic hashing function would be used. Cryptographic hashes are functions which cannot retrieve the input to the function given its output. It is recommended to use SHA to generate a "verification hash" from the encryption key. This verification hash is to be stored on the server, as it is impossible to derive the encryption key from the verification hash. This hash is then transmitted to

the client along with the encrypted data, which can verify that the password is the correct one. If it is, then the password (and, thereby, the encryption key) is correct, and decryption can proceed correctly.

One final concern was that a user might forget or lose their password. Three solutions are proposed for this problem:

1. The use of an escrow service. This service could be provided a copy of the password, the encryption key, and verification hash which itself will be encrypted. However, it would not have access to the database which stores the encrypted data and salt. Access to which requires different username and password.

2. Using trusted third party to store a copy of encryption key which will be retrieved during key-recovery.

3. Storing the copy of encryption key on the server that is also encrypted using another encryption key. This step requires the user to remember atleast two different passwords, the second password can be replaced to set of security questions that in general are used to retrieve password. This solution requires the re-encryption of all the previous data stored on the server.

The solution of key-escrow has been examined by Schneier et al. [11] who strongly recommended against it for key recovery due to issues including operational costs, complexity of maintaining such system and trade-offs among others. Since the main goal is to reduce the use of third party, we recommend the last proposed solution to be implemented in the system.

5 Design and Protocols

This section visualizes the message exchanges between the user and the server during the initial key generation, data exchange and password recovery.

The notations used in the diagrams are the following:

- Salt = \$
- K_1 = Main encryption key
- K_2 = Second wrapping key, used to encrypt the K_1 and stored on server. Retrieved when password change request is initiated.
- PWD and PWD_2 = Main password and password for recovery (or set of security answers to some questions), respectively.
- PWD_n = Different/changed password

- KC = Key Cipher i.e. encryption of the main key which will be stored on server.
- ϵ = Wrong password
- KDF = Key Deriving functions i.e. PBKDF2

Each diagram represents the different phase of the system. First diagram visualizes the initial key generation when the user logins in the first time; the second shows a normal data exchange between user and server and the last shows the exchanges done during the key recovery.

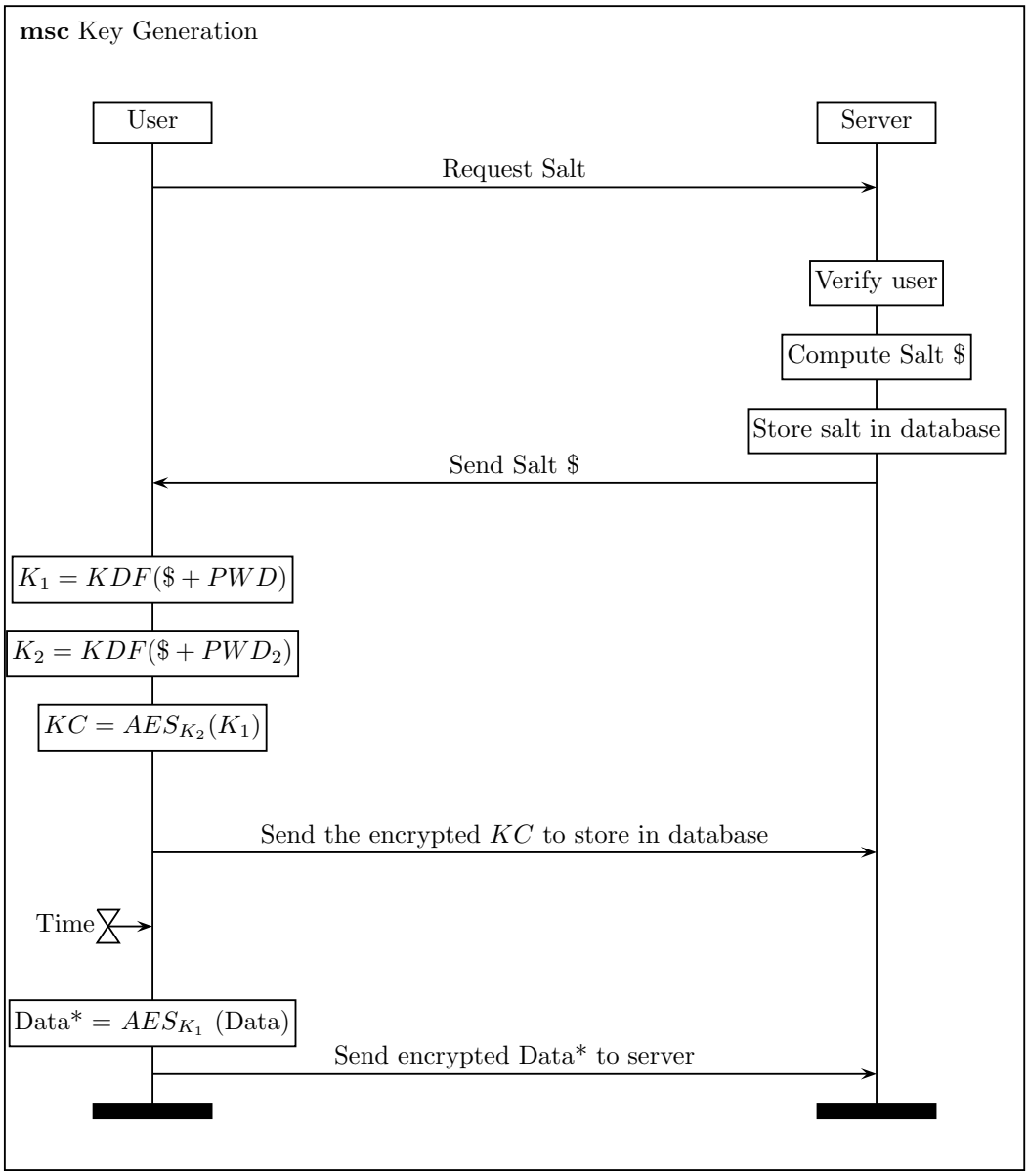

Figure 1: Key Generation

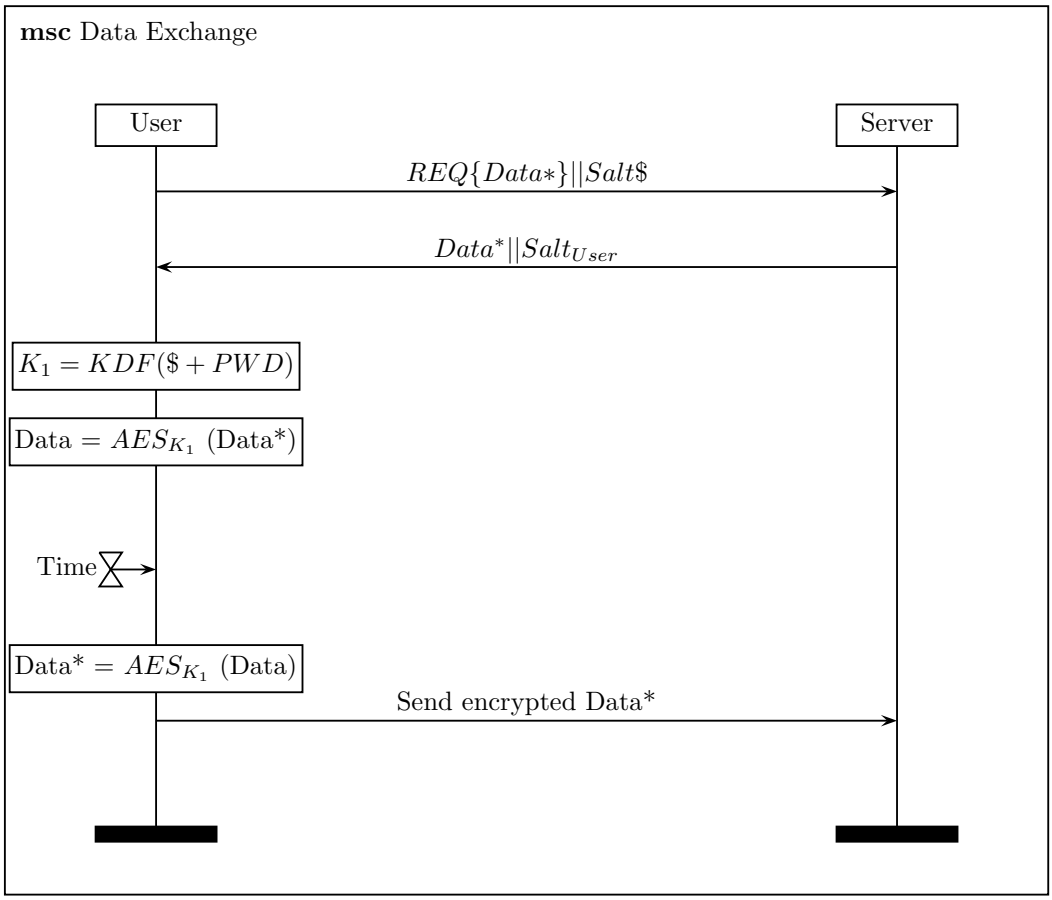

Figure 2: Data Exchange

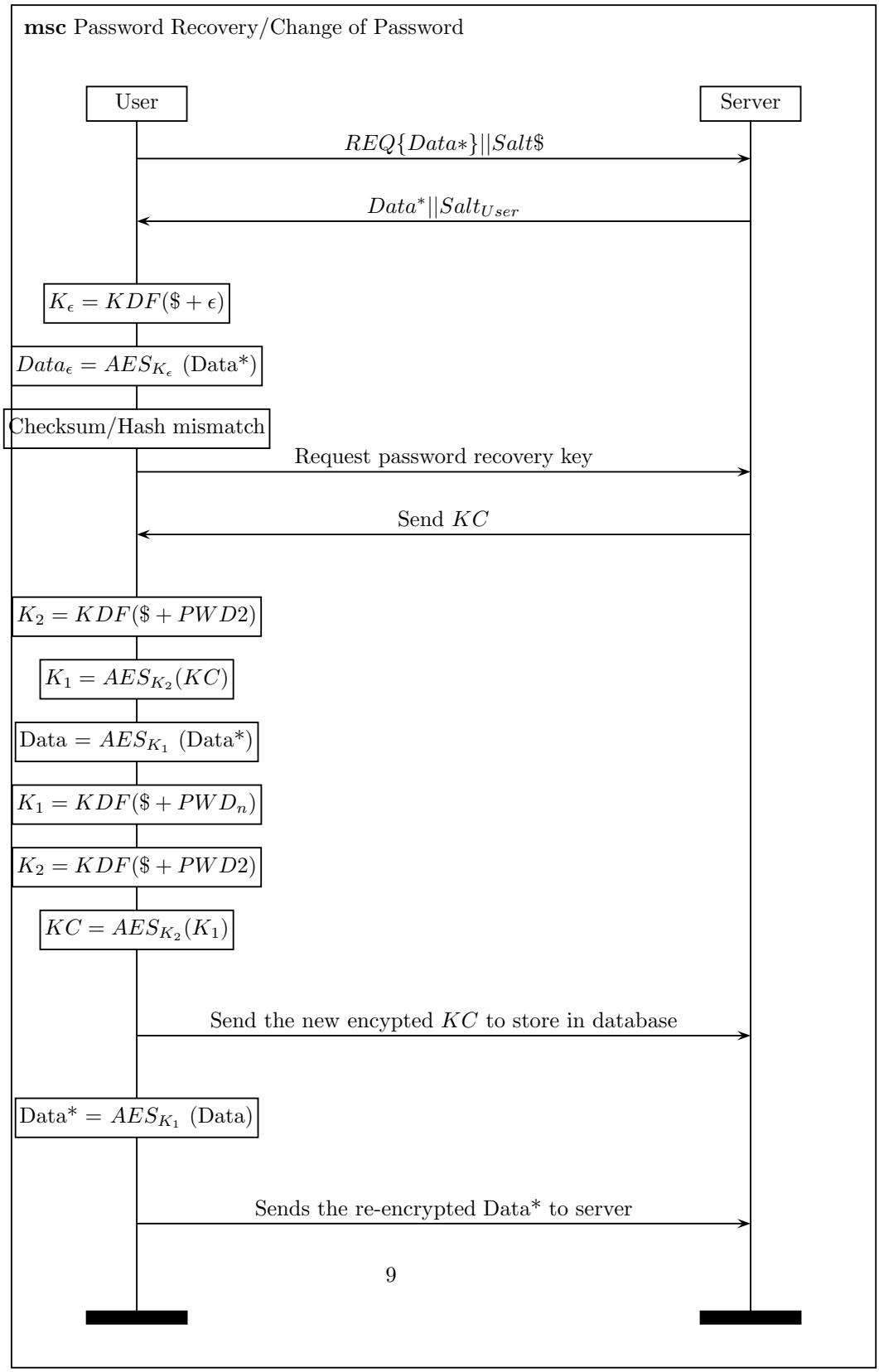

Figure 3: Password/Key change

6 Evaluating the proposed solution

6.1 Performance Testing

A high iteration count in the PBKDF protects the derived key against brute-force attacks by stretching the potentially weak password. The higher number of iterations, the harder it becomes for the adversary to brute force the key.

In order to determine the number of iterations over different devices, we conducted benchmarking test using Javascript libraries including Anandam PBKDF2[37], Stanford JavaScript library (SJCL)[39] and crytp-js[38]. Table 1 summarizes the number of iterations possible for the PBKDF2 in 500 ms on average. The benchmarks of Android considers both the available browsers Google Chrome and Firefox.

Table 1: PBKDF2 Benchmarks

	PC	iOS	iPad	Android	Least
Anamadan	25000000	10000000	50000000	16666667/10000000	10000000
crypt-js	24097	642	657	687/612	252
Stanford	347222	6693	86029	69000/67000	6693

The basic idea of using PBKDF2 is to make the hash function very slow meaning that even when using a fast GPU or custom hardware, brute-force attacks time will have negligible effect. The main purpose of using this is to make the hash function slow enough to evade these attacks, but still keep it fast enough to not cause a noticeable delay for the user. As such with the results derived from the benchmarking, the Stanford library provides a 'good' amount of iterations in a small time making it the ideal choice for use over various platforms.

We also studied the benchmarking for AES, which is the recommended algorithm for encryption of the data. The results of AES (in Tables 2,3,4 and 5)indicate both the encryption and decryption time taken by the schemes for the particular library. Each table shows results of specific devices using different available browsers. For the plaintext, we used a 1.2 MB randomly generated text file, the reason for using the big file was to get an approximation of computation taken by any device when a user starts the key change process either due to having forgotten the password or having desire change of it. Since all the devices were easily able to handle the task of encryption and decryption of a big file, it proves that in the situation of a change in key, it would not be infeasible to perform re-encryption reasonably fast. We used different libraries like jcryption[34], moveable[35] and pidcrypt[36].

The following parameters were used for AES benchmarking:

1. Key size: 128 bits

2. Block size: 64

3. Plaintext: 1.2 MB randomly generated text file with letters, special symbols and numbers

There are other modes of operations [5] for AES but they were not used for testing due to various reasons; Electronic Code Book (ECB) due to its disadvantage of being deterministic when plaintext is the same; Galios Contour Mode (GCM) due to it having *each* operation based on a 128-bit multiplication in the Galois field per each block, which requires higher computational power each time; EAX would theoretically be the ideal choice for the project but due to its unavailability in JavaScript, testing it was not feasible. It must able be noted that the library pidcrypt fails to execute on both the mobile platforms and hence should be avoided if the targeted devices includes mobile device.

Table 2: AES - PC

PC	Chrome		Firefox		Opera		Internet Explorer	
	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec
cryptojs	1068	1316	1337	1384.8	1221.6	1458.2	2642	1730
stanford	503.6	547	335.4	565.6	614	550.6	513.4	824.4
jcryption	1068	1351	1198.4	1172.6	1006	1338.8	1147.6	1147.8
moveable	1069.8	1257	1253	1145.4	945.4	1291.6	1156.2	1165.4
pidcrypt	2128	3477.4	7229.6	10699.2	2303.8	3580	2342.8	3768.4

Table 3: AES Android

Android	Chrome		Firefox	
	Encryption	Decryption	Encryption	Decryption
cryptojs	2910	2808	3275	3506
stanford	1374	1596	1032	1249
jcryption	2466	3359	2037	2046
moveable	2606	2626	1711	14383
pidcrypt	-	-	-	-

Table 4: AES iPad

iPad	Chrome		Safari	
	Encryption	Decryption	Encryption	Decryption
cryptojs	18937	28861	4353	4560
stanford	5887	7557	1442	1779
jcryption	27161	26454	3821	3826
moveable	26423	28213	3666	3935
pidcrypt	31079	36451	18358	20097

Table 5: AES iPhone

iPhone	Safari	
	Encryption	Decryption
cryptojs	11806	12942
stanford	3532	4849
jcryption	12269	12538
moveable	7865	10151
pidcrypt	-	-

6.2 Security Testing

The system consists of two main components, the first one being the PBKDF2 and the other AES. In order to test the security of the system, we mounted an attack on the PBKDF2 as a compromise to this component would lead to compromise of the whole system. Additionally, the AES has already been proven secure by various researchers [12].

For our demonstration we made some assumptions: The adversary has complete knowledge of the parameters of key derivation function, salt, derived key length and a pair of ciphertext and its corresponding plaintext. During our testing we set the number of iterations as 1000, salt to be 128 bit hex and the derived key length to be 128 bits.

The adversary then tries to derive a key from the known parameters, either using some dictionary or generating each possible string as password. The computational power available to the adversary will determine the amount of time it takes to brute force the key. For testing, we used a normal computer with 2.3 GHz processor which was able to check approximately 5000 hashes per second i.e, 5 password per second.

During our testing we were able to derive a formula that provides with approximation of the effort needed by the adversary to derive the key. The *effort*[1] here is the amount of resource and the time available with the adversary. For the formula, some assumptions must be made: firstly, one needs to think in terms of the hardware an adversary will be using and secondly we define each available core as a unit. This is due to the inherent nature of the PBKDF2 which cannot be computed in parallel[2]. In order to simplify the calculation we assume that one unit i.e. core can test approximately 100 passwords per second.

A typical keyboard has 26 lower-case, 26 upper-case, 10 numbers, 32 symbols, and space; which equals to 95 characters. The number of passwords for a given size, say S, and a specific password length n is denoted as S^n. The

[1] effort = resource $\times$ time

[2] A clever adversary can spread the computation over several cores

adversary has 10,000 cores at disposal.

Summarizing, there are 95 character for password (S), a single core can test 100 password per second ($rate$), the password of length (n) and the adversary has 100,000 cores he can use per year ($effort$).

$$\begin{aligned} S^n &= rate \times effort \\ n &= \log_S(rate \times effort) \\ &= \left\lceil \frac{\log(rate \times effort)}{\log(S)} \right\rceil \\ &= \left\lceil \frac{\log(100 \times (10,000 \times 3 \times 10^7))}{\log(95)} \right\rceil \\ &= 7 \end{aligned} \tag{1}$$

Summarizing the result we assume that an adversary has 10,000 cores which is used to generate 100 PBKDF2 keys per second, then for a 7 length password consisting of any letters, it would take approximately one year to brute force it. Conclusively, even for password of length as short as 4, it offers security that can stand brute forcing for more than a year provided that the adversary is restricted to normal computing resource[3].

7 Discussion and Conclusion

During our initial research, we learned that are some systems which permit the encryption and decryption at the browser end, but all of them lack the option of recovering/changing the password/key. Additionally, most of these systems requires trusting third-party to manage the key. The description of our system is targeted towards protecting record data while allowing the feature to easily change the associated password/key.

An important note is that this system is only intended to protect data from misuse on the server side of the system. This is intended to be used as a part of a larger system which provides user authentication, access logs, and other protections against unauthorized access from clients.

This system was devised for the encryption of protected health information and its storage. While this paper discussed the model of the system, many points must be taken in consideration when implementing it. Along with the points mentioned below, the details in[19] must be strongly considered.

[3]This resource is a normal high-end commercial computer

7.1 Recommended good practices

7.1.1 Salt Reuse

Using the same salt for each hash can lead to complex problem. Usage of same salt may happen either due the salt being constant i.e, it is hard-coded in the program or it is generated randomly only once. This is very ineffective as if same salt are used with same common password, they will output the same hash. An adversary then can use a reverse lookup table attack to brute force through every hash at the same time. They just have to apply the salt to each guessed password before its hashed. If the salt is hard-coded in the code then the lookup tables and rainbow tables can be built for that salt, to make it easier to crack hashes generated by the product later-on. A new random salt must be generated each time a user creates an account or changes their password. It must always be unique per user.

7.1.2 Short Salt

Using short salt would lead to an adversary building a lookup table for each possible combination of salt and guessed password. To make it impossible for an adversary to create a lookup table, the salt must be long. A good practise is to use a salt that is the same size as the output of the hash function. For example, when using SHA-256 which is of 256 bits the salt must be atleast of 256 bits; for SHA-512, 512 bits and so on. It is highly recommended against using the MD5 and SHA-1 due to its relatively easy collisions for two inputs[20].

7.1.3 Cryptographic Random Generator

The generation of the salt on the server side must be cryptographically random. Using a random salt makes it less feasible to brute-forcing a single password which will not be the case when the salt is predictable. The generated salt while random must also be long enough to protect the password from rainbow tables. Use of small or even blank password can make it vulnerable to hash look-up tables. These tables contain the common list of passwords along with its hash values stores against them which means that if the salt is either small or blank the adversary just has to check the generated hash against the password in the table.

7.2 Conclusion

In this paper, we have presented a secure model for applications dealing with e-Health that uses only client-side encryption to prevent sensitive data from being available to anyone other than the user.

The proposed solution allows the possibility of changing the password in the event of forgetting the password without placing trust on third party services or requiring high computation effort. We did not find this to be implemented

in any of the systems currently available. The system was developed taking in consideration that it will be accessed from browsers of devices which has limited memory and computation resources.

7.3 Future Work

From all the requirements presented in this paper, our proposed system satisfies all of them besides the authorized sharing. Implementing group sharing schemes on the JavaScript is a challenge and something to consider for future work. For example, implementing some of the group sharing algorithms like attribute based sharing[15], would allow sharing of the sensitive data within specific users while keeping the security intact *and* work on device browsers, developing an innovative way to inform the user if the same data already exists in the server after which he can request access to that data from the original creator of that data.

8 Acknowledgement

We would like to thank Fabien Guillaume for his assistance in implementation. Also to Maria Mateo Iborra and Luis Pena for their insights.

References

[1] Morse, R. et al. *Web-Browser Encryption of Personal Health Information* `BMC Medical Informatics and Decision Making 11` 2011

[2] Fernandez-Aleman, J. et al. *Security and privacy in electronic health records: A systematic literature review* `Journal of Biomedical Informatics , Volume 46 , Issue 3` 2013

[3] Kierkegaard, P. *Electronic health record: Wiring Europe's healthcare* `Computer Law & Security Review 27` 2011

[4] Atchinson, B. Fox, D *The Politics Of The Health Insurance Portability And Accountability Act* `Health Affairs 16 (3)` May–June 1997

[5] NIST Computer Security Division's (CSD) Security Technology Group *Block cipher modes* `Cryptographic Toolkit` NIST, 2013.

[6] Recommendation for Password-Based Key Derivation *NIST Special Publication 800-132* `http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf`

[7] Advanced Encryption Standard (AES)– Federal Information Processing Standards Publication 197
http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
November 26, 2001

[8] PBKDF2 *PKCS #5: Password-Based Cryptography Specification Version 2.0*
tools.ietf.org/html/rfc2898

[9] Percival, C. *Stronger Key Derivation Via Sequential Memory-Hard Functions*
http://www.tarsnap.com/scrypt/scrypt.pdf BSDCan 2009

[10] Schneier, B *Fast Software Encryption, Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)*
Cambridge Security Workshop Proceedings (December 1993)
Springer-Verlag; 191–204.

[11] Anderson, Schneier et. al *The Risks of Key Recovery, Key Escrow, and Trusted Third-Party Encryption*
https://www.schneier.com/paper-key-escrow.pdf

[12] Lynn, H. *National Policy on the Use of the Advanced Encryption Standard (AES) to Protect National Security Systems and National Security Information*
CNSS Policy No. 15, FS-1 2003

[13] RFC 2315 *PKCS #7*
https://tools.ietf.org/html/rfc2315

[14] Daemen, J. Rijmen, V. *The Design of Rijndael: AES - The Advanced Encryption Standard* Springer, 2002

[15] Ming Li et al. *Scalable and Secure Sharing of Personal Health Records in Cloud Computing using Attribute-based Encryption*
IEEE Transactions on Parallel and Distributed Systems Volume:24, Issue: 1. 2012

[16] Stark, E. Boneh, D. *Symmetric Cryptography in Javascript*
Computer Security Applications Conference, 2009. ACSAC '09. Annual IEEE, 2009.

[17] ENISA *Algorithms, key size and parameters report 2014*

[18] Kamara, S. Lauter, K. *Cryptographic Cloud Storage*
Microsoft Research

[19] Contini, S. *Method to Protect Passwords in Databases for Web Applications*
Cryptology ePrint Archive, Report 2015/387 2015

[20] Xie, T. *Fast Collision Attack on MD5*,
Cryptology ePrint Archive, Report 2013/170 2013

[21] BoxCryptor *Encryption software to secure files in the cloud*
www.boxcryptor.com

[22] Wikipedia - A Free Encyclopedia. *Comparison of online backup services*

[23] Wikipedia - A Free Encyclopedia. *e-Health*
wikipedia.org/wiki/EHealth

[24] Microsoft HealthVault
www.healthvault.com

[25] THIRRA Electronic Health Records Systems *Electronic Health Records System for ambulatory care*
http://sourceforge.net/projects/thirra/

[26] Mediboard
http://mediboard.org

[27] DropSecurePro
www.dropsecurepro.com

[28] CryptSync
http://stefanstools.sourceforge.net/CryptSync.html

[29] SafeBox
http://safeboxapp.com

[30] Tahoe-LAFS
www.tahoe-lafs.org/trac/tahoe-lafs

[31] Tarsnap
www.tarsnap.com

[32] SpiderOak
spideroak.com

[33] TrueCrypt
truecrypt.sourceforge.net

[34] jCryption - JavaScript form encryption using openSSL
http://www.jcryption.org/

[35] Moveable - AES Script
http://www.movable-type.co.uk/scripts/aes.html

[36] pidCrypt - a Javascript crypto library
https://www.pidder.de/pidcrypt/

[37] Javascript PBKDF2
http://anandam.name/pbkdf2

[38] CryptoJS
code.google.com/p/crypto-js

[39] Stanford Javascript library (SJCL)
https://bitwiseshiftleft.github.io/sjcl

[40] Dropbox
www.dropbox.com

[41] iCloud
www.icloud.com

Appendix A

Figure and Table

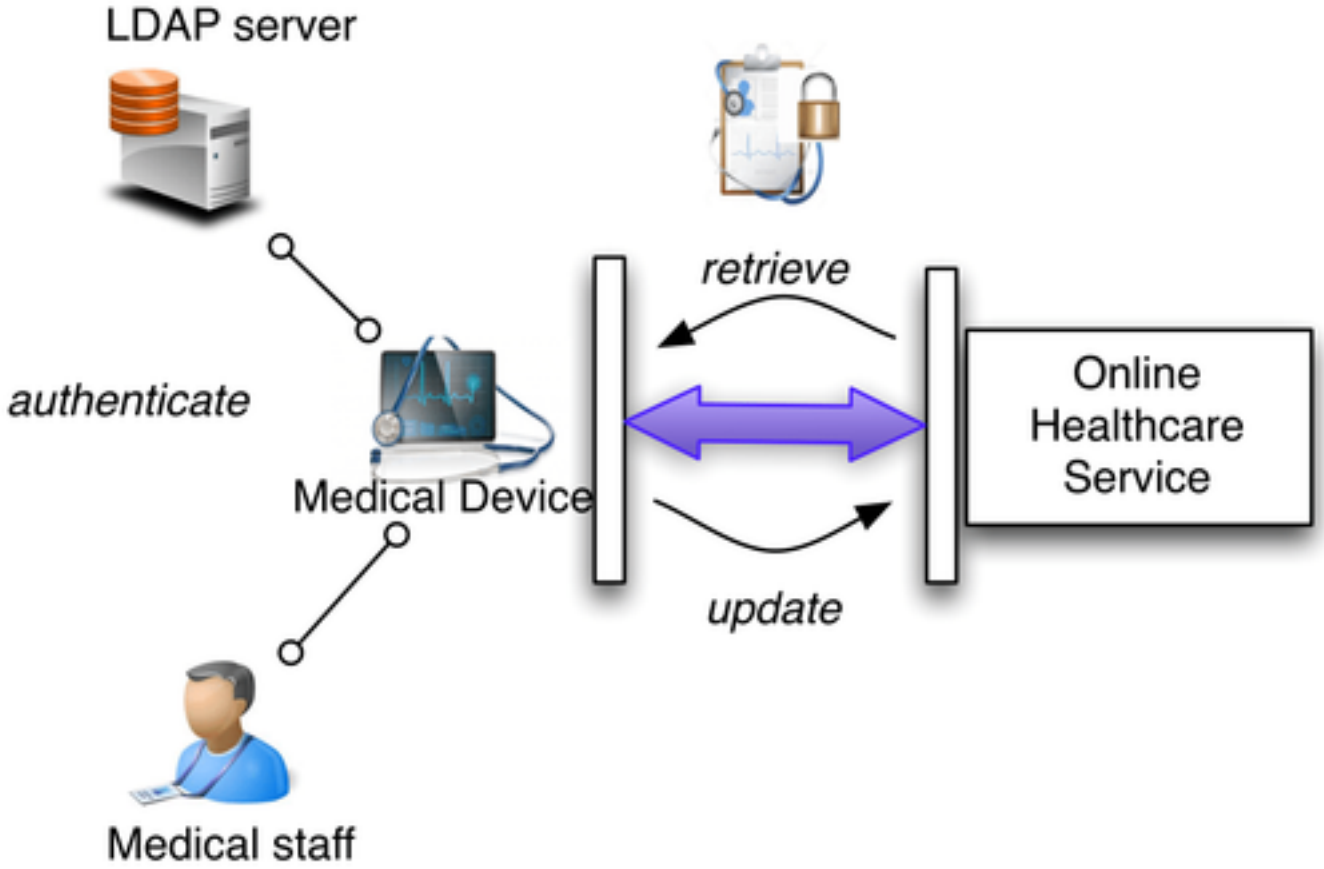

Figure A.1: e-Health System

Name	Confidentiality Guarantees (Claims)	Type	License Type	Mobile Friendly	Architecture or Design	Password or Key based	Cryptography schemes	Accountability of Actions	Support of user roles and identity	Remarks (in comparison with the project)
Microsoft HealthVault	Easy access; Privacy; possibly encrypted data	Cloud based EHR	Commercial (free for personal use)	Yes	Easily deployable; built structure for one user; can be used to manage multiple users; ability to share relevant information only	Password	-	Requires explicit information from user; provider can read/share/write(?) data	Users can read/write; permitted users can either be in read only mode or read/write but not delete; group policy supported	Possibly best choice among others in terms of usability and support for medical work; apparently also has education services
Thirra	Mobile, offline working	Cloud based EHR	?	Yes	Mostly built for medical services in developing countries	?	?(possibly none)	Organising local storing node	Possibly yes, akin of group based policies	Seems to offer same services, specifically built for medical purpose in developing countries
MediBoard	Secure	(Cloud) EHR system	Open source	Maybe(?)	Built to manage patients electronic record	Password and username	Unknown	-	No	-
Drop Secure	Extreme Secure	cloud encryption	Commercial (free on some platforms	No	Encryption at users end (on instructions) then upload to cloud. Requires user interaction every time	Password	-	Requires same password and software at both ends	-	Not recommended ; shady and incomplete description; dead links; probably back-door; [Absolute]
CryptSync	Secure; compression	cloud encryption	Open source (active)	No	requires password interaction	Password or PKI	PGP; add-on for 7-zip	User-to-user only	Cannot be read/written by anyone than user	Older version of OS not supported; compression limitations apply
SafeBox	Secure	cloud encryption	Commercial (9.31)	Maybe(?)	Needs users interactions	Password based key	AES-GCM 256 bits key (self management of key)	User only	None	-
BoxCryptor	Secure encryption	cloud encryption	Commercial (72/year)	Yes (with app)	End-to-end management; possibility to adapt to centralised situation	Password based PK + Symmetric	Aes +RSA + password	Only users themselves able to see unless group encryption used	Possibility to have access policy for different levels groups/users; readable only by – end-to-end unless master key provided	Even if commercial, the basic functionality and technical working is available; can be adapted to the case
TrueCrypt	Absolute secure encryption	cloud encryption	Open source (somewhat active)	No		Password derives	Various (AES serpent and twofish, AES-Twofish, AES-Twofish-serpent,Serpent-AES, Serpent-Twofish-AES, Twofish-Serpent)	None	Computationally can be heavy (depending on the cascading chosen); proven to be truly secure by independent cryptographer the span of two years	
Tahoe-LAFS	Secure, fault-tolerant	Cloud based storage	Open source	Yes (if configured)	Decentralised ; based on principles of least authority	Password	POLT	Based on privileges	Yes; needs to be assigned when initializing uploading the data file	Requires decentralized infrastructure Underlying idea being the RAIN - Redundant array of independent nodes
Tarsnap	Secure, Anyomous storage	Cloud based storage	Open source	No	Key file based system	File based (a single key file)	RSA + HMAC + AES	Based on single key file to access, no authentication	No	Requires storing of key file on device and needs the keys to be of 1024+ size, not feasible for e-Health model

Table A.2: Comparison Table