IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661,p-ISSN: 2278-8727, Volume 19, Issue 3, Ver. V (May - June 2017), PP 53-66
www.iosrjournals.org

Securing Sensitive Digital Data Techniques

Tony Karavasilev[1], Svetoslav Enkov[2]
[1](Computer informatics, Plovdiv University "Paisii Hilendarski", Bulgaria)
[2](Computer informatics, Plovdiv University "Paisii Hilendarski", Bulgaria)

Abstract: *This paper identifies the correct and wrong approaches of securing digital data. It also proposes further ways of increasing the overall security of data transfer and storage. The variety of data can be encrypted either for comparison only purposes or for further active reuse and update, which implies the need of having a different approach when processing the data flow. Several effective models and techniques of encrypting sensitive digital information are presented in this paper, that excludes all reviewed bad practices.*

Keywords: *cryptography, security, secure, data, compression, hash functions, encryption, pseudo-random number generators, hash collisions, rainbow tables, salt, HMAC, PBKDF2, SHA-2, SHA-3, AES, RSA, SSL.*

I. Introduction

We use computer applications every day, but is our private digital data truly secured enough or is it a white lie we tell ourselves to sleep better? It is very important that the people behind the project and the servers are doing things the appropriate way or there will be no point of including security features at all. When talking about sensitive digital data, we cannot skip the importance of the data itself. Having that in mind, we can ensure which kind of data should be under extra solid protection and which under optimal security measures only. Overkill is not an option, but mathematically secure is always a must. For example, the information for your credit card number or password is always more important than hiding your gender details. Of course, we should not neglect the privacy of the human user in our application and keep all data hidden behind maximum security at all costs. The protection of all types of user data is a top priority.

Digital data can store a lot of different semantic information, but the way it is being used is the most important factor to take in mind when protecting it. For example, passwords should be stored for authentication purposes only and the plain text must not be easily retrievable. While other types of data like credit card numbers, imply the need of heavy reuse and the ability to retrieve the initial stored plain text. Having this in mind, the architect of a software system should classify each set of data he is going to store and choose the right set of techniques for securing it. Also, different extra measures may be needed when protecting sensitive data both in-transit and at-rest.

Project designers, managers and unskilled developers sometimes underestimate the security level needed for your data and try only to catch the schedule for next release date. This article is all about how not to be this type of IT professional and identifies the correct and wrong techniques in implying cyber security. After all, it is a must that our data stays hidden and does not include any thread for the user's online wellbeing. Have in mind, that securing sensitive digital data is an important part of your system, but security should not be seen just as an add-on to our application's features. It should be well embedded in the whole application development cycle (design, development, testing, bug fixing, releasing, maintenance, etc.).

The sections of this paper include the differentiation of user data and its nature in both immobility and transfer states. It identifies the correct ways of implying security techniques and how to avoid ruining data protection because of fake personal advice or the heavy rush imposed by upcoming deadlines. There are several effective security models proposed in this article, that provide efficient security level and maintain the integrity of sensitive digital data via cryptographically secure algorithms, protocols and network rules.

II. Securing Data Using One-Way Encryption Techniques

This section reviews the accurate ways of protecting the password data type. The next subsections are about how to secure data used for authentication or comparison purposes only. They also show the most common mistakes that can ruin and compromise the overall data protection.

When designing a system, one of the most important things is authentication. There comes the need of storing user passwords or tokens that must not be stored as plain text. When keeping such information, the encryption involved should be one-way. Meaning, when getting the encrypted value, we should not be able to decrypt it easily and should use it for comparison only. This is done only to prevent someone from the software support of stealing the user's password and to create a protection if the database data is stolen by a hacker.

A. The correct way

The proper way of securing this type of data is using a hash function. Cryptographic hash functions are any one-way functions that can map data with arbitrary size to a fixed bit string output, also called a hash or digest. It is the easiest way of doing a valid digital fingerprint of the given input data and is usually represented as a hexadecimal number, stored as a string. [1]

A well-known manner of attacking the security of such functions and trying to decrypt the output to plain text is generating rainbow tables. This is done by generating and storing digests for all possible inputs and used for easy lookup later by a given digest. Another faster approach is the dictionary attack that contains the digest of words, phrases, common passwords and other strings that are likely to be used as a password. This way the attack is less computationally expensive than the brute force approach. [2] Having this in mind, most of the hash functions are used with pre-generated strings, also called salt strings, that are concatenated to the input data. This is a basic way of ensuring that the output hash is not in the begging of a lookup table. Of course, just using salt will not significantly boost your security if you are not using it with the proper algorithms. For example, some functions are broken because having collisions, meaning where two inputs have the same digest. When wanting to store and secure some sensitive data for authentication or comparison purposes, the best way of using hash functions is:

1. Generate a long random salt;
2. Prepend or append the salt to the input data and hash it with a secure hashing function;
3. Save both the salt and the digest in the user's database record.

When wanting to authenticate, we will want to compare the digest with a given input. This is done by:

1. Retrieve the user's salt and digest from the database.
2. Prepend or append the salt to the given password and hashing it using the same hash function.
3. Compare the hash of the given input data with the hash from the database. If they match, the authentication should pass, if not the user’s attempt should be rejected.

Currently, there are three effective and cryptographically secure ways of implementing this:

- HMAC hash functions;
- Slow computable hash functions;
- Encrypt with symmetrical algorithm before hashing.

The next subsection will explain why these solutions are active and how to use them properly.

1) HMAC hash functions

A keyed-hash message authentication code (HMAC) is a specific type of message authentication code (MAC) involving a cryptographic hash function and a secret cryptographic key. Any cryptographic hash function may be used in the computation of an HMAC. The cryptographic strength of the HMAC depends upon the cryptographic strength of the underlying hash function, the size of its hash output, and on the size and quality of the key. [3]

The main plus of these functions than just using a hash function is that HMACs are substantially less affected by collisions than their underlying hashing algorithms alone. Also, having a password as a parameter changes the way the function computes the output digest and leads to making it harder for creating a rainbow table that has all input to password combinations. Note that the HMACs do not remove the need of salt strings, because someone can still find a way to compromise the input password.

When using an HMAC for encrypting your data for storage and comparison:

- Use more advanced hash algorithms like SHA-2 or SHA-3 functions, rather than MD5 or SHA1;
- Prepend or append long salt strings to the input data;
- Use an extremely long password for computation;
- The longer the password and the salt strings, the better.

Implementing a system which uses a strong HMAC algorithm with message salting is a cryptographically secure way of protecting your sensitive data.

2) Slow computable hash functions

Although adding a long salt to a hash function will ensure that attackers cannot use specialized attacks like rainbow tables, it does not prevent them from running dictionary or brute-force attacks on each hash individually. This is possible because of the existence of high-end graphics cards (GPUs) and custom hardware capable of computing billions of hashes per second.

To make these attacks less effective some types of hash functions use a technique known as key stretching. [4] The main idea is to make the hash function slow enough, so that even when using specialized advanced hardware, the brute-force attacks are too slow for a fast generation. These functions are implemented using a special type of CPU-intensive hash function that makes calculations slow enough to avoid common attacks, but still fast enough to not cause a noticeable delay for the application user. These algorithms take a security cost factor or iteration count as an input argument that determines how slow the hash function will be. Some algorithms can have more than one parameter to fine tune the slowness of the function.

A good approach is benchmarking the function on a variety of devices so that you can find the value that makes the hash to take about a half second in your application. This will make the digest harder to break without affecting the user experience.

A professional way of using this kind of hashing functions is:

- Use algorithms like PBKDF2, Lyra2 and Argon2;
- Use strong password and salt strings, preferably without reuse;
- Fine tune your iteration counter and other parameters so that the computation is slow enough.

Adding this kind of function to your software application will make digital data protection cryptographically secure enough.

3) Apply symmetrical encryption before using hashing

Having in mind that all hash function may have a way of finding the initial value, it is a good approach to add symmetrical encryption before hashing the input. This way even if the attacker gets the initial input, he will hit a wall in attempting to decrypt another advanced algorithm.

Symmetrical cryptography is like a box with a key. After putting a message in the box you lock it with a key and send it to the target receiver. If the box is stolen while in transit, it should be unbreakable and only able to unlock with the proper key. The only problem with this type of encryption is how to keep the key a secret and hidden enough. It is called symmetrical because the box is locked and opened with the same key. Note that there are algorithms that do not work that way and may need different keys to function correctly. [5]

When implying such security measures with hash functions, we must:

- Encrypt the data before hashing it, not after;
- Use strong symmetrical algorithms like AES, over weak ones like DES;
- The stronger the key and other algorithm parameters the better;
- Find a way to keep the key at maximum safety and out of user reach;
- The stronger the hash function the better;
- Prepend or append long salt strings before hashing for extra protection.

Applying this extra layer of security to your software will keep the data locked under cryptographically secure walls.

B. Wrong approaches, misuse and common mistakes

After we have reviewed the most secure and up-to-date ways of protecting this type of data, we cannot skip giving an overview of what are the most common misuses of these techniques. The next subsections show all things that you should avoid when implementing this type of protection. This is the so-called “Do not do like us” subsection.

1) Using broken hash algorithms or insecure ones

A lot of unskilled developers think that just using a certain type of hashing will keep them safe. They tend to use the easiest and already broken hashing algorithms like MD5 or skip adding a salt string. When applying such kind of fake security measures they put the user and the company at risk.

2) Using broken implementations

Some of the developers are intoxicated with the use of their favorite technologies and hash functions. They place them in every application they work on and without checking if the current programming language, library, framework or custom implementation has a secure enough realization. A lot of the open source implementation and language libraries become corrupt, full of vulnerabilities or just deprecated and misplaced. If the data is stolen, the attacker may attempt checking if the data is decryptable by exploiting know bugs.

A common example is using a secure algorithm like some slow hash function, but the current library version you are using is full of bugs and defects. Before using a secure algorithm, check if the implementation you are using is not deprecated or broken.

3) Dealing with anti-pattern lovers

Someone at your company may obligate you and your fellow colleges of using a certain inappropriate technology because of their personal flavor. This imposes the risk of using something that is not secure enough or does not apply to the current software needs. Be careful when you choose the same technology all over again.

Another common mistake is the "invented here" syndrome. This is when someone is forcing the opinion that when reinventing the wheel or inventing their own algorithm will be more secure than if you use an already existing one. Simply do not do that, because no matter how smart you are, there are people who spend their life in breaking and developing cryptography algorithms. Your ego will never protect your data privacy.

4) Myth and legends

Do not believe anything you read in a forum, blog, etc. Just because someone is gossiping about it, it does not make it a truth. A common example is the myth that SHA-3 functions are many times slower than other hash functions like those in the SHA-2 standard. [6] This is simply not true. Always read the official references or studies and test them yourself.

5) Short password parameters when using HMAC algorithms and reuse

The worst approaches when choosing to use HMAC functions are:

- Using short and weak password parameters;
- Reusing the same password for all users and hashing calculations;
- Storing the passwords in the same database and table with other user related stuff.

6) Common mistakes when applying salt strings

The most common mistakes when using hash functions are:

- Not using salt strings at all;
- Using short and weak salt strings;
- Reusing the same salt string for all users and hashing computations;
- Storing the salts in the same database and table with your user details.

7) Key and salt generation

When having a need of generating random data for use as key or salt strings you may end up filled with the same data and open up easier for a hacker's decipher attempt. You must generate these type of strings using a cryptographically secure pseudo-random number generator (CSPRNG). [7] If you use a plain pseudo-random number or string generator, you are inserting a big venerability into your storage backyard. Use only cryptographically approved standards for pseudo-random generation of data.

8) Recursive hashing using plain functions

Some developers believe that the recursive use or nesting of algorithms like MD5 and SHA1 in combination with salting will keep them safe. They are wrong, because this will still make them vulnerable to rainbow tables. It is safe to say that using recursively HMACs is good enough, if calling them with different passwords and salt strings. [3]

9) Using just AES instead of hash functions

As spoken before, using a symmetrical encryption algorithm in combination with hash functions is a great idea. On the other hand, using just that is not secure enough for this type of one-way data. Some people attempt to use the AES encryption [2] instead of hash functions and end up with fake security because of:

- Using it with the same parameters will make the data more vulnerable;
- Creating a pattern between the input data and the encryption key;
- Using it with less secure parameters or short passwords;
- If keys are too short, brute-force attempts using modern hardware may be possible;
- Encryption keys are easy to steal from a database and decryption time is fast by design.

Using just a symmetrical algorithm instead of a hash function is not secure enough because of the danger of creating a pattern between the input and the password used, by design fast decryption and keys theft.

10) Password reset and token time to live problems

A lot of systems use encryption properly, but leave logical holes creating the ability to bypass your defenses. They try to exploit the password reset strings and authentication tokens to either reset user passwords or steal sessions. When implementing this kind of features, ensure that there is an appropriate time to live timeout and that password reset strings are not always the same for each user recovery attempt.

III. Securing Data Using Retrievable Encryption Techniques

This section reviews the appropriate ways of protecting data that needs heavy reuse of the initial plain text. A basic example would be a credit card number stored and used to finish online payments when needed. The next subsections are about how to properly encrypt data that is under active use. They also show the most common misuses that can compromise the stored data.

When having a need to store sensitive digital data that needs plain value retrieval, one of the most important techniques is applying symmetrical cryptography algorithms. As reviewed before, these algorithms are like an unbreakable box with a key. Only people with the proper key or an exact copy of it can unlock the box and get its content. These algorithms have misplaced other old and insecure historical symmetrical algorithms that involve character permutations and by string shuffle manipulations.

A. Correct way

No doubt, the best way of protecting reusable sensitive data is by using symmetrical cryptography functions. There are some other important parameters they use and that we have not reviewed yet. They are the block mode of operation, the initialization vector and the padding bytes. Most modern algorithms work in a binary block processing matter of some fixed length. They split all the data into blocks with a predefined size and encrypt them one by one. The way encryption is applied and how they involve the previous blocks is called mode of operation. There are a lot of operation modes, but only a few have been proven truly secure. Because every mode needs a previous block to use data from in some matter, here comes the need of an initialization vector (IV) with the needed fixed length or also known as starting variable (SV). Having this in mind, using a unique two pair combination of a key and an IV will make the algorithm harder to be cracked. [7] [8]

Also, these functions have another loose end called padding. It must not be easily neglected. When dividing some input binary data into blocks, the last available block may be shorter that the size required. This creates a need of adding some binary data to fill in the blanks and is also known as padding bytes. The simplest example would be filling zeros, but there are other standards which are more secure and add data in some predefined matter.

It is very important that the key, IV, mode of operation and padding are carefully chosen or the algorithm may be compromised. The key and IV must be kept a secret at all costs and we must use only standardized symmetric functions. Modern hardware support really fast encryption and decryption, but the brute forcing of the key and IV pair may take millions of years, even when using multiple computer systems.

When wanting to store and secure some sensitive data for active reuse, we must first choose a symmetrical algorithm and its block size. After that, we select a mode of operation, padding, define a strong key and an IV with the chosen block size.

When we are ready, the encryption process of input data can start and is as follows:

1. Store the key and initialization vector;
2. Initialize the algorithm with chosen parameters;
3. Pass the data for encryption;
4. Save the output cypher data to a database or a file.

The decryption process of encrypted data is:

1. Read the stored key and IV pair;
2. Read the enciphered bytes from a database or a file;
3. Initialize the algorithm with chosen parameters;
4. Pass the data for decryption and retrieve the plain text or binary data.

Most of the time, keeping the key and initialization vector is the greatest challenge. Sometimes they are hardcoded into application code or placed into a more secure database. Of course, you can encounter a hybrid approach with multiple keys and IVs on both sides of your software.

In our modern age, there are three correct and cryptographically secure manners of implementing this:

- Using the AES standard with strong parameters;
- Multi-layer symmetrical encryption with a set of different keys and IVs;
- Combining symmetrical encryption and basic steganography principles.

The next subsection will illustrate why these solutions are secure and how to use them accurately.

1) AES usage

The Advanced Encryption Standard (AES), or also known by its original name Rijndael is an electronic data encryption specification established by the U.S. National Institute of Standards and Technology (NIST). It

uses a block size of 128 bits, but has three different key lengths: 128, 192 and 256 bits. AES has been adopted worldwide as the official encryption standard. It supersedes the Data Encryption Standard (DES) used earlier. The algorithm can be used with a lot of different mode of operations and padding standards. There can be some possible attacks on it when not using the proper and most secure parameters. [2] [5]
When using AES for securing your data, you must:

- Choose either AES-192 or AES-256 for top secret information;
- Use a strong key and IV combination;
- Use either CTR or CBC modes of operation, found as most secure;
- Use padding standards PKCS7 or ANSI X.923, and stay away from zero padding;
- Do not give technical feedback information to the user if decryption fails.

Using this algorithm, the right way will guarantee the quality of encryption and keep your sensitive data safe and secure.

2) Multi-layer data encryption

Another well-known approach is using different symmetrical algorithms or the same one multiple times. Keep in mind, that encryption and decryption should happen with different amount of strong keys and IVs. To ensure the security of this approach, you should use only standardized and proven algorithms with the right parameters chosen for mode of operation and padding. A good idea is always to include at least one AES encryption at first and after that apply the next ones.
When using this approach, you have to:

- Always include AES-192 or AES-256 with secure parameters;
- Do not apply weak algorithms or there will be no point of applying them at all;
- Use different and strong keys and IVs for each layer of encryption/decryption;
- Do not apply a pattern between your keys, like rehashing the initial key, etc.;
- Do not reveal technical information to the user on which layer decryption breaks and why;
- Use at least three layers of high-end encryption.

When used in an accurate manner, this approach will ensure data security and give a lot more trouble for an attacker to deal with.

3) Coupling steganography and symmetrical encryption

Combining steganography and encryption can be a very powerful weapon. Steganography is the practice of concealing a file, message, image, audio or video within another file, message, image, audio or video. This is the art of hiding things in plain sight. If we hide the real message in a dummy one and after that encrypt it, even if an attacker is able to decrypt the data, he would not see the real data hidden inside. [9]
When doing the coupling, you must:

- Encrypt the real message with some secure algorithm;
- Inject it in some dummy message or image, etc.;
- Apply another encryption on top with a different key and IV pair;
- Use only standardized encryption methods with strong and different key and IV pairs.

It is a great way of tricking the attacker and misleading him, if he has managed to break the encryption applied. Even if he is smart enough to find the injected information, there is a second layer of top encryption guarding it safely. Of course, more layers of encryption may be applied in this situation.

B. Wrong approaches, misuse and common mistakes

After we have reviewed the most secure ways of using symmetrical encryption, we must also make an overview of what are the "Do not do like us" tricks of the trade applied over these techniques. The next subsections show all things that must be avoided when implementing the protection of actively reusable sensitive data.

1) Using broken algorithms or insecure ones

Some developers think that applying some randomly picked algorithm will do the job. They are wrong and imply fake security measures. An example of this is choosing an already broken algorithm like DES or some kind of publicly developed one that tends to change a lot. This may lead to easily decryptable data or even loss of data when switching to a new version of an algorithm. Use only standardized and security proven functions.

2) Using broken implementations

This is a common problem, as spoken before, when the developer chooses the right approach, but installs a defective realization of the desired function or functions. Always check the version of the chosen implementation of the features you need. If you are not careful, you may end up with a bunch of easily exploitable vulnerabilities or even data loss.

3) Dealing with anti-pattern lovers

As said before, some people may fall in love with certain algorithms and their realization. They may convert the project into a "Stovepipe system". This is a barely maintainable assemblage of ill-related components. A lot of developers come with prejudices about what the project needs or does not. Be careful what things you implement and what are the truly needed features for the project. No matter how smart you are or think you are, ego and philosophy will not overcome reality. Be a true professional.

4) Myth and legends

Using overrated and bloated technology instead of official and secure algorithms, should be a crime itself. Just because there are some rumors or false advertising from a concurrent private product, this does not mean that it will keep your data safe. Read the official documentation and test for yourself before you judge or brag about something. Do not be fooled by misleading opinions. Prepare for the worst and hope for the best.

Also, you will encounter paranoid human beings that will try to convince you that cryptography is absolute and try to prove some conspiracy that government agencies, quantum computers and parallel computations already have your data available in milliseconds. Do not trust them, quantum computers [10] are at their birth and even if there are machines available to decrypt everything in seconds, the most important thing is that the average security professional will not be able to get your data. Also, there is more risk involved in discovering bugs in cyphers than the above. Do not believe everything you read, verify your trust.

New algorithms are invented every day by specialists to keep your data immune from any new threat that may arise. Use standardized algorithms in the right manner and your data will be secure. Have no fear, it leads only to unhappiness.

5) Using wrong encryption parameters

You must relook the up-to-date state of different parameters you tend to use, like mode of operation and padding standards. As reviewed before you should use block modes like CTR or CBC, because for example using a mode like ECB will certainly lower your security. Also in some contexts CTR may be more secure than CBC and a lot of systems have migrated from using it. But in other situations CBC is the more secure.

Today, CTR seems to be the more secure option, plus it has both parallel encryption and decryption, while CBC has only parallel decryption. [11] Some experts recommend using modes like EAX or GCM that internally use CTR or CBC, then using them by themselves. Be careful what combination of other parameters you set up or just using a secure mode will not keep you safe.

You must never use zero padding in symmetrical encryption, because it is not appropriate for professional purposes. Instead use PKCS7 or ANSI X.923. Have in mind that some of those standards may be withdrawn in time like the well-known standard ISO 10126.

The most important things you always have to do:

- Review the states of the algorithms you use;
- Use them in the appropriate context;
- Choose the most secure ones;
- Set them up it the correct way.

6) Key and IV pair quality and reuse

When choosing your key, you should create a long and character rich one. The more complex, the better. As for choosing an initialization vector, you are obliged to have a fixed block length one. It is a good idea to make it more sophisticated. Have in mind, that key reuse may be a bit tolerable, but IV reuse should be a no go zone.

Also, the user should not have a word in choosing their key and/or IV pair. Still, if you implement this kind of feature in your system, use some rehash algorithm over their IVs and implement a unique generated salt string system (to make their passwords more secure by appending or prepending the generated data sets).

7) Key and initialization vector generation

As reviewed before, if you have a need to generate a key or IV, you must always generate them with a cryptographically secure pseudo-random number generator. [7] Anything else is just not professional.

8) Key and IV pairs storage

As said in the previous subsections, someone may generate lower grade keys and IVs or reuse them. But there is another problem that comes up – their storage. The good thing is that in most cases they must be somewhere inside the system and are not transferred anywhere. Symmetrical algorithms are made for this kind of usage.

The question is whether to hardcode them or softcode them? Truth is, you may end up using both. A lot of systems have at least one hardcoded key and IV in the source code itself, this way even if the encrypted data is stolen from the database, it will not compromise security. Some data like IVs can be softcoded and stored in a different database, connected only via a limited SQL user account on a different instance and after that transformed in the application's code. This way even if the IVs and encrypted data are stolen, the hacker will not have the key and the true IVs data, because the initialization vectors should be transformed before use and that algorithm is somewhere in the source code. When using multiple encryption layers, you may store some keys in source code and others in a different well-protected database accessible via different credential set. Have in mind that stealing the source code is also possible.

Truth is, that when you are an internal employee and have access to both the production's source code and database, you would be able to steal the information no matter how complex security is implied. Beware who you give full access to the system or you will be sorry.

9) Including compression – should you or should you not?

Another good approach of both strengthening security and saving some space is adding compression before encrypting the data. Of course when decrypting, you would have to decompress to be able to use your data again. The pros of using it are that you eliminate some duplicate patterns in the input data and save a lot of storage space. [12] [13]

Have in mind that if you decide to encrypt first and compress afterwards, you may end up with even bigger size of the cypher text than when not using compression at all. This is because both compression and encryption remove duplicates and patterns. When you apply compression after encryption, it would not be able to make the size much smaller and will need extra space to store the header information of the used archivation algorithm.

Compression may be a double-edged knife, because when used with AES or weak parameters, it may introduce some side-channel attacks [14] that can recover your key in a matter of seconds. Use it only with strong encryption measures and in the correct manner. Another bad thing is that compression algorithms may be a bit CPU intensive when using ultra compression modes, but on professional hardware it should not be a problem after fine tuning.

10) Use asymmetrical algorithms instead

There is another class of cryptography algorithms that use asymmetrical manners. The most used in practice algorithm of this type is RSA. It is one of the first practical public-key cryptosystems and is widely used for secure data transmission. In such a cryptosystem, the encryption key (public key) is publicly available and differs from the decryption key (private key) which is kept a secret. RSA is based on factoring the product of two large prime numbers. As opposed to symmetrical algorithms, you have one key for locking and another for unlocking your message. RSA and other algorithms like it are designed for safe communication and secure transfer over networks. Usually, the used RSA key lengths are between 1024 and 15360. [8]

These types of algorithms are created to encrypt only data smaller than the key size used. This makes it not really suitable for longer binary data. While on the other hand, symmetrical algorithms are made to encrypt practically unlimited size of data. Even more, not only the data should be less than the key size of RSA, but there is a specific amount of reserved bits that should also be taken out of the given input maximum length.

Some people make a common mistake of creating an algorithm that splits data into sizes smaller than the key used with an RSA algorithm, after that iterate over each data chunk and encrypting it using the public key. At the end, all chunks are glued together using a homemade custom algorithm. The decryption is made by deriving the chunks, looping over each data split and decrypting it using the private key. At the end they are glued back together. There are several big reasons this asymmetric algorithm is not used in practice:

- RSA and asymmetrical cryptography algorithms are not intended for bulk data encryption;
- RSA is created only to encrypt short data like symmetrical algorithm keys, for safety transfer;
- It is not a standard method and is going to be an idiosyncratic scheme only you use;
- It probably will force you to manage a very large number of large key pairs;
- Will probably break the secure padding scheme used with those algorithms;
- The chosen RSA key length may already have become insufficient for security purposes;
- RSA is much slower than algorithms like AES, because it is designed for small data lengths;
- May create unforeseen patterns, vulnerabilities and cypher weaknesses.

The main point is, some people think that a solution can be universal no matter the context and try to twist the way it should behave although its true nature. This is totally insecure and is often used by cargo-cult programming developers that do not understand how anything works, but pretend so. Keep it real and keep your data safe. Be an example for your colleagues, not the opposite.

IV. Proposed Digital Data Security Models

This section contains several proposed abstract security models that combine the correct use of techniques and standards reviewed in the previous sections. Also, it contains pros and cons details, plus ways of evolving them even more. Implementing them in the correct matter will secure sensitive digital data and supply professional encryption protection. Also, all models should use HTTPS traffic, for secure data transfer.

Of course, you can feel free to simplify them a bit or make them even more complex. It depends only from your project's needs. Some of the algorithms can be exchanged with others from the same security level.

A. One-way encryption models

The next two proposed models are for securing one-way data needed for comparison or authentication. They implement the proper use of the encryption techniques discussed previously in this article.

1) Combining AES, HMAC and database ACLs

This security model is based on adding symmetrical AES encryption before hashing, using an HMAC function and the separation of the digest from the key and salt parameters. It is shown on Figure 1.

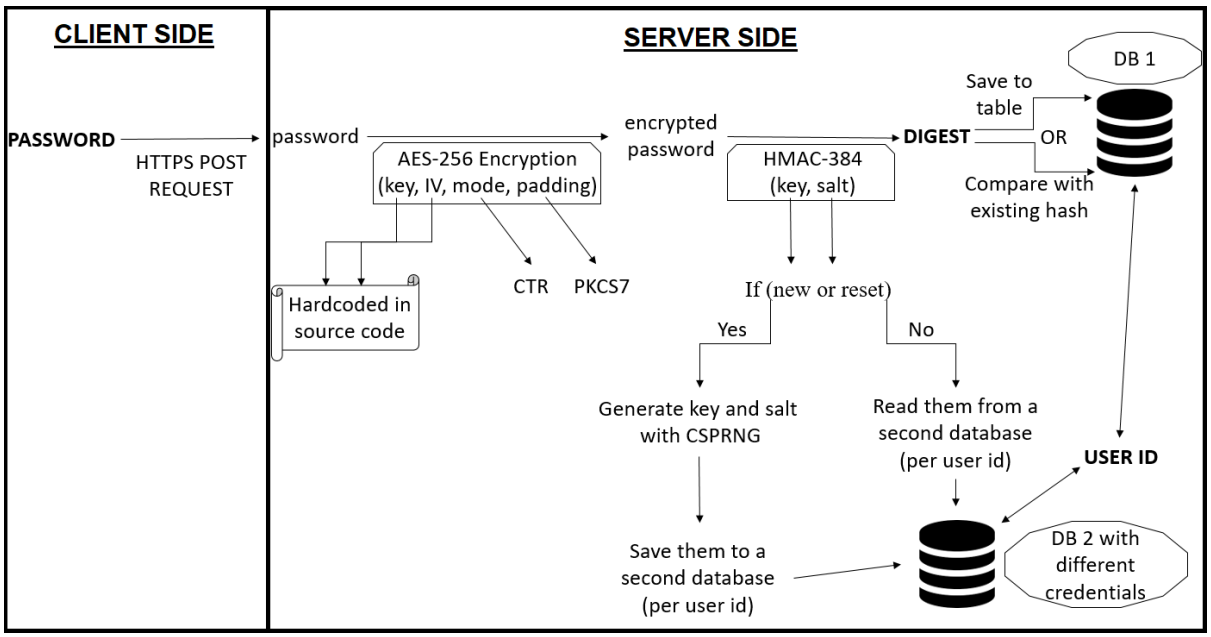

Figure 1: Architecture of the first proposed one-way model

This protection scheme of processing the data creates the following pros:

- Even if the chosen HMAC-384 function breaks, there is another layer of protection with AES;
- There is no key or salt reuse for hashing and they are randomly generated using CSPRNG;
- On password reset, the key and salt string are securely regenerated;
- The key-salt pairs are not stored in the same table and database as other user related stuff;
- Both databases are with different access users and may be on different instances;
- For attacking the system, both access to production's source code and the two databases is needed, which is always hard to get;
- If one database is stolen, it would be useless without the data in the other one.

The cons of using it in your system may involve the following:

- Temporary problems when one of the databases is down;
- If there is a need of changing the AES hardcoded parameters, you would have to force all your users to change their password at once and notify them via email;

- If you use a short password and IV pair for the AES algorithm, you introduce a risk for the user's privacy if the hashing layer protection is breached;
- On user delete some problems may occur and multi-database operations would be needed.

Ideas for improving or altering the model:

- There is another alternative way of implementing this, by using one database but with two users having different access over tables. This way you can make one user to be multi-use related and the other only used in the encryption application layer;
- You may also include one more collision-free hashing before saving to make the final size smaller for storage;
- Some systems implement a password digest per user history, so that they ensure you will never use the same password again. Implementing such policy can be healthy;
- If not implemented properly, database locks or driver timeouts may slow your authentication system down;
- Changing some of the chosen algorithms with more secure ones;
- Adding multilayer symmetrical encryption.

2) Using AES, slow hash functions and symmetrically encrypted key-salt storage

This proposed data protection scheme uses symmetrical AES encryption before hashing, using a slow computing hash function after that and encrypting the key-salt pairs being used. It is shown on Figure 2.

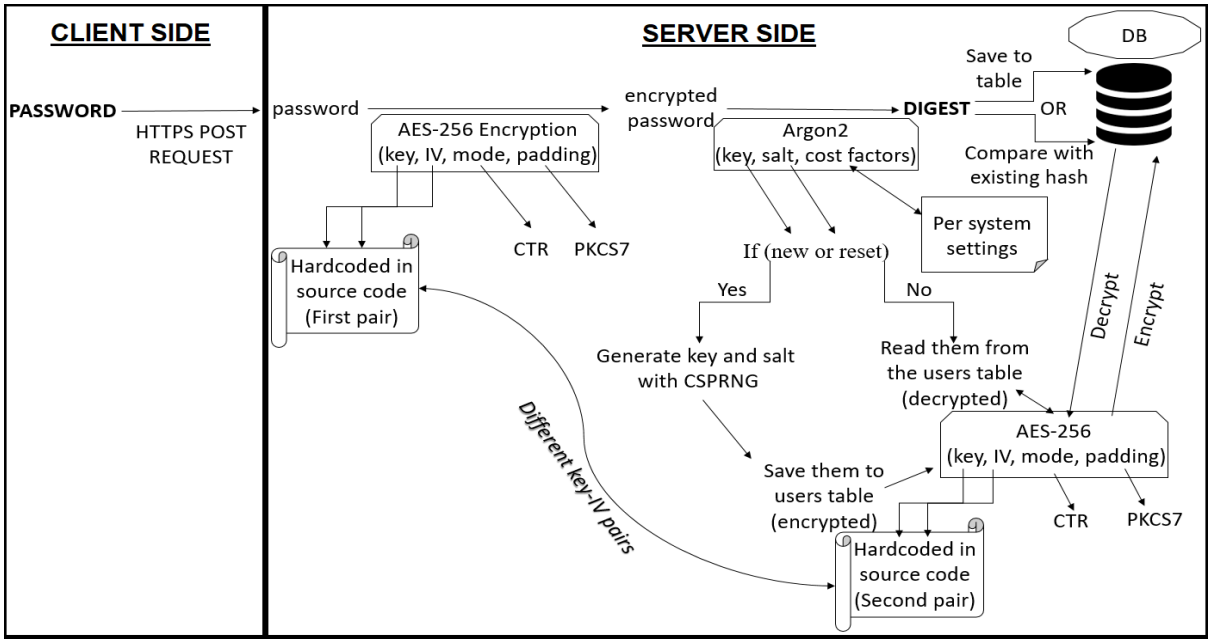

Figure 2: Schema of the second proposed one-way model

The architecture of the given model has the following pros:

- Even if there is a remote chance of breaking the fine-tuned slow hashing function Argon2, there is another layer of protection on top, using AES-256;
- There is no key or salt reuse for hashing and they are randomly generated using CSPRNG;
- On password reset, the key and salt string are securely regenerated;
- The key-salt pairs are encrypted with AES-256 before storing them into the users table and decrypted when needed back for use;
- Different set of hardcoded parameters are used for the two AES protections;
- Even if the source code and the database are stolen, you would be faced with the slow brute forcing of the hashing function.

The cons that may be introduced of its usage are:

- Changing the AES hardcoded parameters used for the first encryption, would require forcing all your users to change their password and to notify them;
- You need to carefully tune your hashing function for all the devices you may use or your protection may be compromised faster than you initially thought;

- If you need to change the AES hardcoded parameters for the key-salt pairs, you would need to temporary drop access to the user table, decrypt the data and re-encrypt it with the new parameters.

Improvements and alternatives:
- Ensuring that the user will never use the same password again by keeping digest history;
- Benchmarking it carefully on all devices that may be used;
- Changing some of the chosen algorithms with more secure ones;
- Adding multilayer symmetrical encryption.

B. Symmetrical encryption models

The next subsections propose two models that are for protecting data which is needed for active reuse. They achieve maximum security, using the accurate encryption techniques reviewed earlier in this paper.

1) Using AES-256 encryption

This security model is based on using only symmetrical AES encryption with hardcoded parameters per application. The way it can be applied, can be seen on Figure 3.

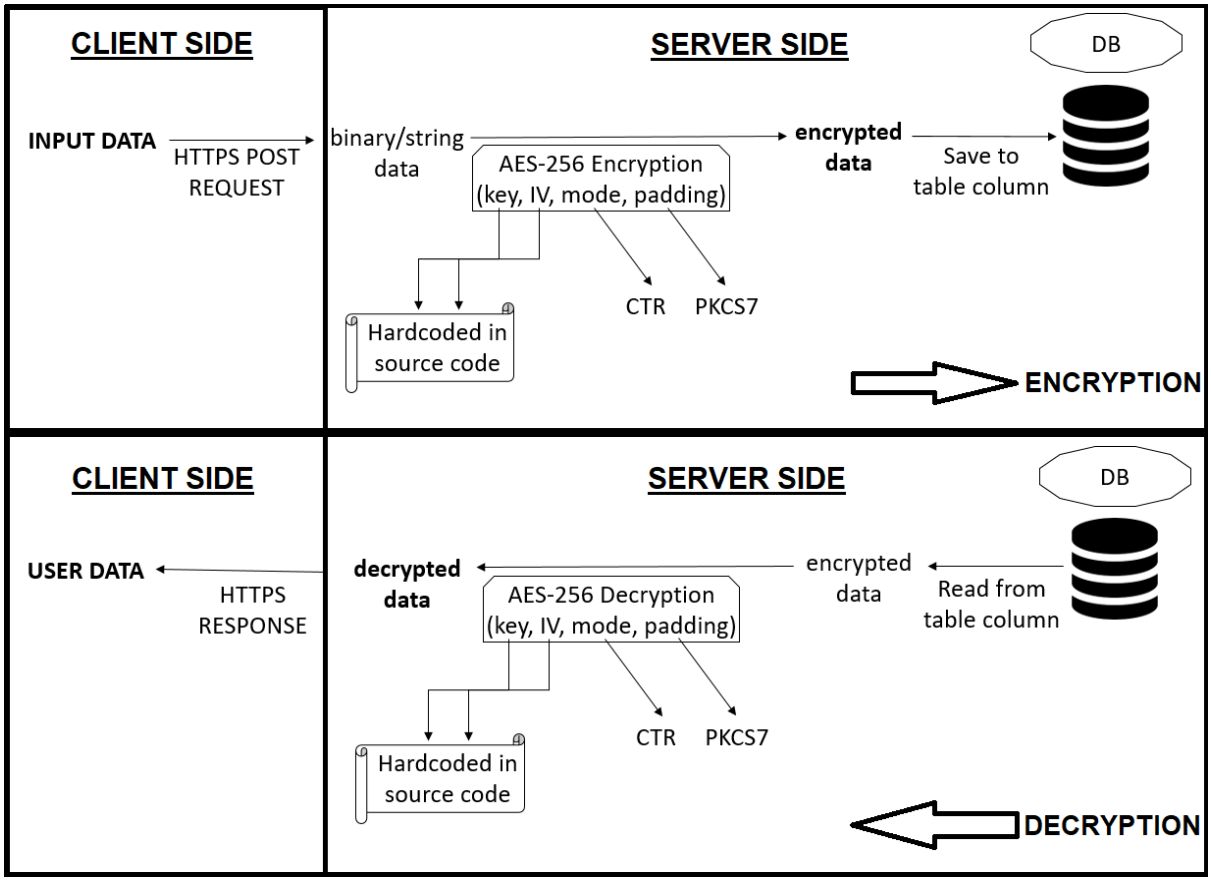

Figure 3: Architecture of the first proposed symmetrical model

This protection scheme of processing the data creates the following pros:
- Storing data under solid AES-256 protection, using CTR mode and PKCS7 padding;
- Fast encryption and decryption, but really slow brute forcing of the key-IV pair;
- Hardcoded data cannot be easily seen from external people;
- The more complex the key and IV pair, the better security.

The cons of using it in your system may involve the following:
- Stealing the hardcoded data from internal staff;
- Having to maintain different keys and IVs for your developer, staging and production environments or migrating to the softcoded approach;

- If you need to change the AES hardcoded parameters, you would have to temporary drop access to your application module, decrypt the data and re-encrypt it with the new parameters;
- If you use a short password and IV pair, you introduce a risk for the user's privacy.

Ideas for improving or altering the model:

- Moving the hardcoded key and IV values to the softcoded approach or combining both. This way you can change credentials through the system's admin panel or the database directly;
- Adding multilayer symmetrical encryption with different key-IV pairs, some hardcoded and others softcoded.

2) Multilayer symmetrical encryption

This proposed data protection scheme is based on multiple AES encryption passes with different key and IV pairs. The key-IV storage is spread between both the application and the database sides. The model can be seen on Figure 4.

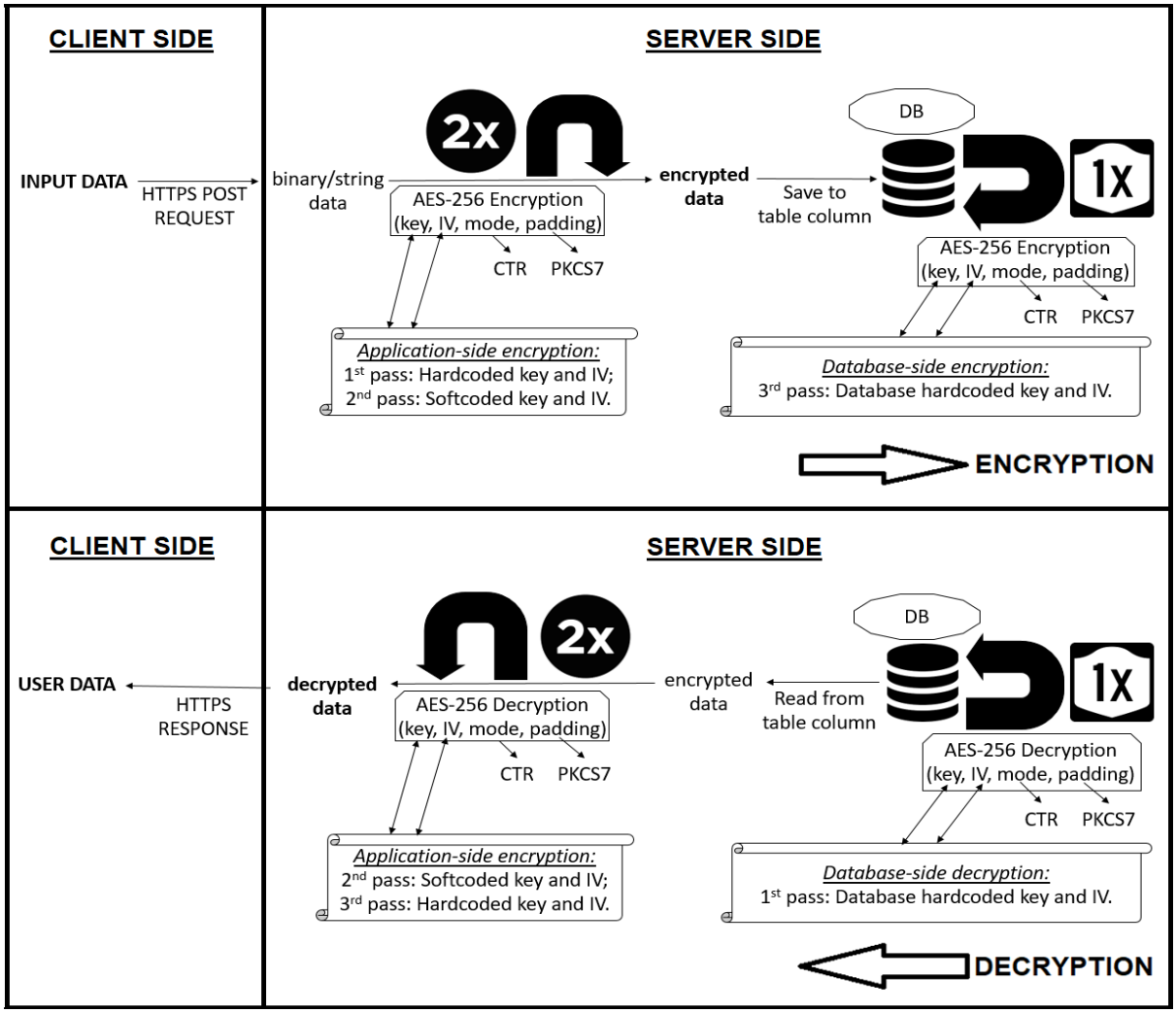

Figure 4: Schema of the second proposed symmetrical model

The architecture of the given model has the following pros:

- Multiple layer AES-256 encryption with 3 different key-IV pairs;
- To encrypt or decrypt the data, one must have access to both the application's source code and production's database. Which is not easy to have;
- The database-side AES encryption should be implemented with using a trigger and some stored procedure that has the needed parameters. This makes it harder for not authorized personnel to manage it or steal it. Especially when the application database user has no rights to look up stored procedures, etc.;

- The application side AES encryption pass reads the parameters softcoded somewhere in the protected database table. This makes it easier to change key-IV pairs on developer, staging and production environments. Also makes it harder to get, without the appropriate rights.

The cons that may be introduced of its usage are:
- Stealing the hardcoded data from internal staff with master rights;
- If you use a short password and IV pairs, multiple encryption won't do you any good;
- Changing the parameters of any of the three AES encryption layers, will cause you headaches with decrypting everything and encrypting it all over again. This may cause you a lot of down time and not very happy clients;
- It may be too complex for the average software system.

Improvements and alternatives:
- You may move some of the layers either to the application or to the database side;
- Create a system and some scripts for automatic decryption and re-encryption when needed;
- For some systems, the hardcoded approach is better, for others the softcoded one. Do what is best for your data security;
- Include encrypted steganography injections into randomly generated image data, before or after storing your data.

V. Ensuring Data Protection Both At-Rest and In-Transfer

After having reviewed the top techniques and proposed protection models for your digital data, we must not skip the importance of data storage and data transfer. You may have done everything right as a developer or designer of a system, but get hacked because you have not secured your server and network setup or have used outdated transfer protocols. [1] [15] [16] [17]

The next two sections will be about best practices and will cover the following two data states:
- At-rest –All information stored on physical media and software related manipulations upon it;
- In-Transit – Covers the time when data is being transferred between components, locations or programs. Mostly covering transfer over the network or during an input/output process.

A. Boosting the at-rest security level

To ensure the best protection in data storage, you should:
- Avoid data loss via redundant array of independent disks (RAID) and link aggregation (LAG);
- Enable transparent full disk encryption on all your servers (including the database one);
- Use SQL cryptography aggregate function for encrypting important data;
- Use role-based access and file level permissions;
- Use system event logging and log analysis software;
- Use monitoring software to ensure everything is working as it should;
- Install all available security updates on used operating systems and their packages;
- Configure all your application services settings in security-driven matter;
- Use secure erase/deletion techniques for data removal and physical disk disposal methods;
- Buy expensive hardware, secure the physical server location and use ISP business plans only;
- Check if your real IP is not blacklisted and schedule daily antivirus/anti-malware scanning;
- Daily data backup and weekly machine configuration backups can save your life later;
- Add encrypted remote backup solutions and safe full drive cloning services.

B. Boosting the in-transit security level

To level up your protection, you should always:
- Transfer data only over encrypted traffic like HTTPS and SSL/TLS;
- Do not use self-signed SSL certificates for public use;
- Remove the use of insecure algorithms that can be negotiated from the SSL handshake;
- Disable HTTP pipelining on the server or face both security and performance issues;
- Set up proper network firewalls and local firewalls on every operating system;
- Take up hiding virtual machines behind demilitarized zones and container isolation solutions;
- Limit remote access to servers and use only encrypted traffic protocols (VPNs, SSH, etc.);
- Use network access control systems with banning, logging and limiting functions;

- Block anonymous WAN requests, ban WAN NAT Redirection and filter multicast;
- Block ports and protocols that are not needed for between server communications;
- Combine virtual LAN (VLAN), firewalls and MAC filtering for management networks;
- Ensure that there is no secret use of port mirroring, traffic sniffers or keyloggers;
- Design your system against eavesdropping and other man-in-the-middle attacks;
- Disable the spanning tree protocol (STP) or you will face PXE blocking and flood attacks;
- Implement cross-site scripting (XSS), clickjacking and SQL injection defenses;
- Use advanced protection solutions against distributed denial of service (DDoS) attacks;
- Take advantage of hotlink protections and password protected directories;
- Update your CMS or framework and use or implement IP blocking software features/plugins;
- Setup CAPTCHA-like systems designed to establish that a computer user is human;
- Use multi-factor authentication to access the servers, their data and for user verification;
- Use only complex and long passwords or preferably SSH key-based authentication;
- Enable user and group security policies between servers and local client machines;
- Develop application based password policies for users and access level control systems;
- Disable domain transfer and hide excessive information about the server and framework used;
- Use custom error pages to ensure that internal error messages are not exposed to end users;
- Ensure that developers have not included backdoors and login spoofing in your application;
- Be careful with the way you set up proxies, load balancers and clusters or they can misbehave;
- Create a custom protection for systems that do not have such (like some NoSQL systems);
- Do not store sensitive data inside cookies and files (even if the information is encrypted);
- Do not include any encryption parameters into the client-side (for both web and desktop);
- Embed security in the whole application development cycle (not just adding security features);
- Create behavior logging and connection tracing components for your software application;
- Use free or paid penetration testing tools and vulnerability scanners.

VI. Conclusion

This paper successfully gives a complete overview of the security landscape in 2017. It accurately identifies the correct and wrong approaches of securing sensitive digital data and how to ensure the safety of data both at-rest and in-transit.

The four proposed security models integrate high-end cryptography techniques and can be used in practice for every software system developed by IT professionals. Applying their use in your application will boost your data security level dramatically higher than before. The given models can be implemented practically with any programming language and SQL database engine. Further steps in improving the security of the models and benchmarking the performance of different implementations can be done in the future.

Acknowledgements

University of Plovdiv Paisii Hilendarski, 24 Tzar Asen, 4000 Plovdiv, Bulgaria

Author Contributions

Both authors contributed equally to this work.

References

[1]. J. Katz, Y. Lindell. Introduction to Modern Cryptography: Principles and Protocols. 2007. ISBN: 978-1584885511.
[2]. S. Vaudenay. A Classical Introduction to Cryptography: Applications for Communications Security. 2005. ISBN: 978-0387254647.
[3]. A. Menezes, P. van Oorschot, S. Vanstone. Handbook of Applied Cryptography. 1996. ISBN: 978-1439821916.
[4]. J. Kelsey, B. Schneier, C. Hall, D. Wagner. Secure Applications of Low-Entropy Keys. 1997.
[5]. J. Daernen, V. Rijrnen. The Design of Rijndael, AES - The Advanced Encryption Standard. 2002. ISBN: 978-3540425809.
[6]. R. Dahal, J. Bhatta, T. Dhamala. Performance Analysis of SHA-2 and SHA-3 finalists. 2013.
[7]. B. Schneire. Applied Cryptography: Protocols, Algorithms, and Source Code in C (2nd Edition). 1995. ISBN: 978-0471117094.
[8]. N. Smart. Cryptography: An Introduction (3rd Edition). 2013. ISBN: 978-0077099879.
[9]. T. Morkel, J. H. P. Eloff, M. Olivier. An Overview of Image Steganography. 2005.
[10]. D. J. Bernstein. Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete. 2009.
[11]. M. Albrecht, K. Paterson, G. Watson. Plaintext recovery attacks against SSH. 2009.
[12]. R. Sharma, S. Bollavarapu. Data Security using Compression and Cryptography Techniques. 2015.
[13]. N. Mark, J. Gailly. The Data Compression Book (2nd Edition). 1995. ISBN: 978-1558514348.
[14]. M. S. E. Mohamed, S. Bulygin, M. Zohner, A. Heuser, M. Walter. Improved Algebraic Side-Channel Attack on AES. 2012.
[15]. W. Stallings. Computer Security: Principles and Practice (2nd Edition). 2011. ISBN: 978-0132775069.
[16]. B. Sullivan, V. Liu. Web Application Security, A Beginner's Guide. 2011. ISBN: 978-0071776165.
[17]. M. Howard, D. LeBlanc. Writing Secure Code: Practical Strategies and Proven Techniques for Building Secure Applications in a Networked World. 2004. ISBN: 978-0735617223.