Proof of security of the Secure Book Wishlist

by Hamza Maimoune

February 2023

The purpose of this project is to introduce myself to cybersecurity and cryptography. The setting is that of a book wishlist application with a user/password authentication. I have to make sure that only through high tech algorithms or the exploitation of preinstalled backdoors can a man have access to any information or modify database content or request/response payload without being authorized.

The project can be found at https://github.com/HamzaM3/book-wishlist

1 The setting

1.1 The actors

There are 4 *Actors*:

- U: the user (it's a person)
- M: the user's machine (it's a computer)
- S: the server (it's a computer)
- A: the attacker's machine (it's a computer)

1.2 The modelisation of a web application

A web application is made of 3 parts:

The database: Contains the data for the app. The user can request the creation, access, update or deletion of data, but only after verifying his authorizations.

The frontend: Display that depends on a state that can depend on both the user input and API responses (in particular).

The API: Interface allowing to manipulate the database. It is made of multiple paths which fill a specific function and takes payload as input to specify the desired action.

We will modelize the database as a function DB that takes a query as input and either outputs data or undertake a modification.

We will modelize the API as a function $\omega(p, b)$ where p is a path and b is the body of the request. The output is the data sent to the user and the queries done to DB. The body has to be of a certain type. If it is not, the API call fails.

We will modelize the frontend as a function $P(S)$ that takes a state as input and outputs a web page. This is the React framework paradigm. Basically, P is a program that we send to the user. It will be able to take his input, induce API calls and display the information.

Throughout this article, we will be relatively loose with respect to the types of the different concepts we introduce.

1.3 Secure API call

A secure API call is given by the following interaction:

- P performs an api call after the user manipulates the machine M. It will send a request at the path p with the payload b.
- S receives the call, calculates $\omega(p, b)$ and sends it to M (it might not if it's just modification)
- M receives the response and lets P use its data.

1.4 Unsecure API call

In real life, there is a possibility for an attacker to intervene in between the communication. He can gain access to what is sent and modify it. The goal is to prevent any unauthorized attacker to read any information or to do any damage.

An unsecure API call is given by the following interaction:

- P performs an api call after the user manipulates the machine M. It will send a request at the path p with the payload b.

- A receives the call's data (p, b) can do anything he want with it. He can read it in particular. The value after A's action is noted $(p, \bar{b})$. Then he can send it to S, or he can pretend to be S and send it back to M.

- S receives $(p, \bar{b})$, performs type verification, calculates $\omega(p, \bar{b})$ and send back the data (if there is).

- A receives $\omega(p, \bar{b})$, does anything he wants with it (like reading it) and sends it back to M. The result of A's operations is called $\bar{\omega}(p, b)$

- M receives $\bar{\omega}(p, b)$ and let's P use its data (after a potential type verification).

The purpose of the following operations is to make sure that $b = \bar{b}$ and $\omega(p, b) = \bar{\omega}(p, b)$ and prevent A from deducing any sensitive information from b or $\omega(p, b)$.

2 The algorithms

Here we list the available algorithms, the assumptions we make about them as well at their weak points, and how these weak points are accounted for.

2.1 Pseudo-random number generator

A pseudo-random number generator is a function that depends on a state and that makes the state evolve while outputing seemingly random numbers. This is a deterministic algorithm, in that, the same state will give the same output. But globally the function verifies some statistical properties that make it seem like it follows a uniform law (or any other probability law).

The issue with these is that if you have acces to a sufficiently long sequence, you can deduce the initial state of the generator (called the *seed*) and therefore all the following ones, thus making the generator completely predictable.

An example of such a weak generator is the linear congruential generator (used in the standard libraries of C for instance) [1].

But any generator will have this deterministic sequence weak point. So one has to make sure, in particular, that it is impossible to reverse the function, that the output value is a partial part of the state and that it hard to find the seed from any sequence of values.

[1] Linear congruential generator - Wikipedia

2.2 AES

I will not explain the algorithm here. We don't really care about the details. What we care about is the macroscopic property of it. If it's AES or anyone else, so long as it can be proven it verifies those properties we are good.

AES is a symmetric-key cryptographic algorithm. This means that two people have access to a secret key K, and the algorithm encrypts any message with that key, and decrypts it with it. But it is impossible to decrypt a message except if you have access to the key. This is done by making the space of possible keys gigantic and the difference of encryption from one key to another very different. Also, two very close messages are going to be encrypted very differently. And the key is a random number, so it's hard to brute force it.

2.3 RSA

It is an asymmetric key algorithm. I have coded it in Python in the repository of the project. Again, I don't need to explain it detail here. The core properties of RSA is that each person has both a private and a public key. The public key is composed of two numbers (n, e) and the private (n, d). There are some mathematical properties that make it so:

- One cannot deduce the private key from knowing the public key
- (n, e) allows anyone to encrypt his message
- (n, d) allows the receiver to decrypt any message sent to him that was encrypted with the public key
- It is impossible to decrypt the message without (n, d)

I have to mention that the real life RSA (which we implemented) perfoms a random padding before encrypting. See the PKCS#1 v2.2 standard for more information. We use the recommended padding called OAES. This is important in order to guarantee that there enough *entropy* to the message.

Imagine coding either 0 or 1 straightforwardly, then RSA will give only two possible values which will give too much information to the attacker.

2.4 Hash functions

This is a class of function that allows one to transform any piece of data into a fixed length number that verifies the following properties:

- Two different values yield the same value with a very low probability
- Two very close values yield very different values
- It is impossible to reverse a hash function

These properties allow one to store data after hashing it. This prevents anybody reading the database to deduce the actual value. And we can verify whether a given value is the right one by recalculating the hash. (We cannot recover the data though)

There is a technique called salting which consists of concatenating a fixed random number before hashing. If secret, this prevents one from doing comparaisons of the results using some intelligent probability laws.

3 Possible actions

We are going to consider that actors have a definite list of functions they can apply and each actor can apply them at will. We will then list all the available functions each actor has access to. Afterwards, we can move towards presenting our method, proving how secure our application is from attackers and what are all the weak points to fix.

3.1 Atomic functions

An atomic function will be a basic function that can be used by an actor. You cannot decompose it. For instance, when someone writes id and $f^{-1} \circ f$, we consider that he means something different. The first one is the atomic identity, the second one presupposes that both f^{-1} and f are accessible to the actor and he applied them successively.

3.2 Multivariate composition

When an actor has access to n functions (f_i), he can calculate the list $x \mapsto [f_1(x), ..., f_n(x)]$.

When an actor has access to a function g of n variables and n functions (f_i), he can compose the list of f_i with g.

If an actor has access to a list-function, he can have access to only one of the components as an individual function.

So actors perform mutlivariate compositions of the atomic functions they have access to.

3.3 RSA couples notation

We are not going to bother ourselves with writing public key and private key variables during this article. So we say that there are two types of function, encryption functions (uses public key), and decryption functions (uses private key). These functions belong to an actor X. We note the couple (e_X, d_X). We have $e_X = d_X^{-1}$. e_X is accessible to other people (in principle) and d_X is accessible only to X.

RSA couples can be used for straightforward encryption but they can also be used for verification of identity. This is done by having a user X encrypt his message with d_X and publishing the result. People will decrypt it with e_X. If it respects a predefined syntax, it proves it comes from X. (This is hard to make a message that decrypts with e_X and respects any simple syntax without d_X)

Keys that are made for encryption are denoted with a superscript e. and keys for verification take a v superscript.

3.4 List of available atomic functions

There is a definite list of atomic functions, actors cannot do anything else during an API call:

- e_M^e, d_M^e: RSA couple of M for encryption
- e_S^e, d_S^e: RSA couple of S for encryption
- e_S^v, d_S^v: RSA couple of S for verification
- e_A^e, d_A^e: RSA couple of A for encryption
- e_A^v, d_A^v: RSA couple of A for verification
- a/a^{-1}: AES function and its inverse. Takes a message and a key.
- id: identity
- h: a raw hash function
- h_p: a hash function using a public salt (because it's in the program)
- h_r: a hash function that hashes data and uses temporary data as salt (not knowable afterwards)
- h_S: a hash function using a secret salt accessible only to S

	M	A	S		M	A	S
e^e_M	X			d^e_M	X		
e^e_S	X	X	X	d^e_S			X
e^v_S	X	X	X	d^v_S			X
e^e_A		X		d^e_A		X	
e^v_A		X		d^v_A		X	
a	X	X	X	a^{-1}	X	X	X
id	X	X	X	h	X	X	X
h_p	X	X	X	h_S			X
ω			X				

Table 1: Accessibility table

- ω: the API. Takes a path and a body as input

There are some special functions:

- r: a random function (we already explained the problems of it)
- ρ: a replace function usable by the attacker (with whatever accessible data he has)
- P: the program itself

3.5 Accessibility table

An accessibility table has been furnished.

Now, understand that if the attacker is clever enough with his compositions on the data he sees, he can get access to functions he shouldn't be accessing. Also, yes, e^e_M is accessible at start only to M.

3.6 Random guessing

The attacker has the ability to evaluate the function he has with whatever value.

For instance, he could try to reverse e^e_S by evaluating multiple input until reaching the value he tries to decrypt (this is called *brute force*). Using all the information he has on the API call (from the program and observing previous interactions) as well as exploiting *a priori* data (like a table of the most frequent password), he can improve his chance of finding the decrypted message.

One important question would be, if the attacker observes for suffiently long the back and forth between the machines and the server and with enough computing power and mathematical knowledge, can he become better at guessing the value ? This is possible, and one has to verify it is not the case or rather that it is intractable for A to increase his knowledge in a non-neglectable manner.

4 Our cryptosytem

Let's now present the cryptosystem we propose.

4.1 Authentication information

We have to find a way to guarantee that the server sends data and performs modification only for authorized users. So we have, first of all, to identify him. This is done using a couple username/password which will be supposed to be known only by the user.

After receiving the username and password, the server will send an *authentication key* (or *authkey*) which will be a special value that represents the user it communicates with.

For each subsequent API call, the user has to send that authkey to guarantee his identity. This is done to minimize the exchange of the couple username/password (which might be used on other platforms and is very sensitive) and also to avoid saving it anywhere on the user's machine. The couple should appear only once during the sequence of interactions and it is on the screen of the user.

4.2 Protocol

It is now time to describe the actual protocol:

4.2.1 Program and key reception

1. M requests S for the program P
2. S sends the program
3. M receives the program and launches it in his browser
4. P at start generate the RSA couple of M (randomly), stores it in the localstorage of the browser and asks for the public keys of S through the path */keys*

5. S has sent his public keys (encryption and verification)

6. The API calls can now be secure

This is the weak point of the interaction. At this moment, two things can be hijacked, the program P and the server's keys. It is out of my reach to prevent this from happening because this requires that the user actually identifies my app concretely. This is called the Man In The Middle attack and the difficulty of resolving it is very known. It can be solved for example if we do a trustworthy public reception (through an ad or a visit card you could imagine) or if we use a trusted third party...

So we are vulnerable to this and we will assume for the rest of the protocol that nothing happens during this part.

4.2.2 Encryption wrapper

When the machine wants to send a request with payload b to path p, it does this:

1. It picks randomly a long key K for a.

2. it returns the value:

$$F_M(b) = [a([b, e^e_M], K), e^e_S(K)]$$

(When we put e^e_M, we mean the public key of the machine.)

When the server receives the message (m', K'), it does this:

$$F^{-1}_M(m', K') = a^{-1}(m', d^e_S(K'))$$

And so it receives b and e^e_M.

It performs $\omega(p, b)$ and sends it using this function, with K_2 picked randomly:

$$F_S(b) = [a(\omega(p, b), K_2), e^e_M(d^v_S(K_2)]$$

.

Then the machine get (m'', K''). We perform:

$$F^{-1}_S(m'', K'') = a^{-1}(m'', e^v_S(d^e_M(K'')))$$

.

This is provably impossible to read, and impossible to modify usefully.

4.2.3 Signing up

To create an account, the user can send his username and password through the */signUp* path. The password has to be hashed once with h_p before it is sent (to make it harder to figure out if the message is decrypted). Then within S, a permanent authkey will be calculated using h_r a and temporary data sent with the credentials (timestamp, some random value from M).

So the authkey correspond to:

$$authkey = h_r(username, h_p(password), temp\_data))$$

Since the salt is temporary, it is impossible for anyone, even S, to brute force it. Unless one knows when the sign up was made and also what was the state of the random generator in M at that moment. Which is a priori very improbable.

Also, the hashed password is again salt-hashed with h_S which uses a secret salt that is not stored in the filesystem of the server but in the RAM. The way we do that is by asking the developer launching the server to enter it by hand in a prompt and save it for himself on some paper. This is done to avoid situations where the attacker successfully gains access to the filesystem. We consider the RAM to be safe (we'll discuss the case when it is not afterwards).

So we save the following value as the *hashed password* in DB:

$$double\_hashed\_password = h_S(h_p(password))$$

And we save the username. So $\omega(/signUp, [username, password, temp\_data])$ saves the following row in the database:

$$[username, authkey, double\_hashed\_password]$$

4.2.4 Signing in

Once the account is created, the user can connect by sending his credentials to the */signIn* path. The password is again hashed with h_p. At reception, S hashes it with h_S and performs a look up in the database for a row corresponding to the $[username, double\_hashed\_password]$ couple. It then returns its corresponding *authkey*.

4.2.5 Authenticated calls

For any other interaction, M is required to put the authkey of the user in the payload so that his authorizations can be evaluated. The reason we use an authkey is to avoid having the username/password information floatting too much in the network. This is a very sensitive piece of information as it can be used to access other platform authenticated as that user.

4.2.6 Recap

The steps of the interaction are the following:

1. M requests P
2. S sends P
3. P generate e_M^e and d_M^e in M
4. P requests e_S^e and e_S^v for M
5. S sends them
6. M signs up and sends his credentials as well as temporary data using F_M and h_p for the password
7. S receives the data, decrypts it with F_M^{-1}, calculate the authkey using h_r, hashes the password with h_S, and saves the data
8. S sends the authkey with F_S
9. M receives the data and decrypts it with F_S^{-1}
10. M interacts with S using other paths but always adding the authkey in the data and encrypting/decrypting with F_M and F_S^{-1}.
11. S receives the authenticated requests and encrypts/decrypts with F_M^{-1} and F_S
12. M logs out (forgets the authkey)
13. M signs in by sending his encrypted username/hashed password.
14. S receives the data, decrypts it, hashes the hashed password, finds the authkey and sends it back after encryption.
15. Repeat step 9

4.3 Analysis of the protocol

It is now time to analyze the protocol with rigour and figure out what are the absolutely guanranteed property of the method and what are the weaknesses of it. We will make a lot of assumptions. We will explain what will the consequences of one of them failing afterwards.

4.3.1 Assumptions

We are going to make reasonable assumptions. If any of them fail, there will be consequences on the quality of the system.

1. M and S have chests unaccessible to A (localstorage and RAM). They can put anything in it secretly.

2. Decryption can only be done through the decryption function or a probabilistic brute force.

3. The attacker A has access to the set of information I made of *a priori* knowledge (about human users e.g.), mathematical knowledge, all API calls, the program P and the server's filesystem. Even possessing all this information, he cannot build in a conputationally tractable manner a probability law $\pi(\cdot|I)$ such that the probability of figuring out the antecedent of encrypted messages is non-neglectable (i.e. the expectation of finding it is not greater that a universe constant bounding tractable probabilities)

4. With respect to the pseudo-random generators, it is impossible to deduce non-neglectable information from a sequence of randomly generated values by S and M, even if the algorithm is public. You cannot find the seed from the values. You cannot have a computationally tractable good probability law for the following values or the preceding ones from the published values.

5. Here are the property about the size of input and output of the encryption/decryption algorithms:

   a/a^{-1}: Input: arbitrary size / Output: approximatively same size as input

   e_X: Input is smaller than output. Input size is fixed (by the key size).

   d_X: Input is bigger than output. Input size is fixed (by the key size).

Also the size for RSA is very small. So we will prefer to use AES for the body of the message and hide the key with RSA (because it fits).

6. A Man In The Middle attack didn't happen during the key and program reception. The user should make sure to have a trustworthy and non-compromised DNS server and a way to confirm its identity. Our job is also to make sure there is no way to pretend to be us (interesting subject to explore).

4.3.2 Some explanation about the assumptions

For assumption 1, this assumption is necessary because we need a place to store the private keys and the secret seed of S. If there are no ways to store anything secretly (which would be catastrophic if I am not going to lie) then an option is to *obfuscate* the program using some extreme tool. For example, there is something called PyArmor that does that for Python programs.

Let's say you do $a = b + 2$, an obfuscation would be doing $for(let\ i = b;\ i < 0x2; ..)$. You make simple calculations more complicated and hide the private keys in it or the seed. So even if you read the RAM, it's not possible to decrypt anything. And if you have an extreme secret, you delete the clear code source afterwards.

For assumption 2, we say that the only way for an attacker without the decryption algorithm to guess the message is through evaluate and compare. And with enough information, you can make it more probable for you to succeed. For example, if we take the hash of a password with no secret seed. If you have a table of most frequent passwords, you can evaluate their hash one by one and compare. You'll have a high probability of finding it out.

Assumption 3 explains that we solved the problem of assumption 2. Of course we have to show it, but basically we expect that brute force is impossible. Which implies that the attacker will search for the private key. Which we'll make unaccessible thus rendering the communication perfectly secure.

Assumption 4 talks about the issue that pseudo random generators have (we explained it section 2.1). We assume we have proven that it cannot be hacked using sequences.

Assumption 5 is important for the verification of identity. We ask the verification keys to be smaller that the encryption keys, basically.

Assumption 6 is the weak point, it's unsolvable except through identification for example through the means we have talked about.

4.3.3 Access and interpretation

Actors do two things, they access information and then they interpret it. We assume that all information (except for the chests) are accessible. And so the only defense we have is to make sure that even if some unauthorized user has access to it, he cannot understand it. So this is not about authorized access, this is about authorized *interpretation*. This is because there are a lot of vulnerabilities in software etc. that make it probable that any part of the communication or of the computers can be listened to/read or tampered with.

4.3.4 What should the attacker not be able to do

In general, actors can do 3 things (in this setting):

- Read data
- Modify data
- Destroy data

Of course, our focus here is uninterpretability, so destruction is supposed impossible (or just useless for any attacker). But with respect to reading data, the API calls and the database can be read or be modified by A. But the attacker should not be able to modify an API call in flight or understand it.

With respect to database reading and writing, what we are focusing on is the case where the attacker accesses it through API calls (using information he gathered). A little focus is given to the situation where the attacker gains access to the computer's filesystem (by hashing the passwords and securing the sensitive data in the RAM). Maybe we can assume that the attacker needs a secret password to access the database (which we would also input using a prompt and then store it (or not) in the RAM).

4.3.5 Can the attacker acquire enough non-neglectable information ?

Assumption 3 explained the worry I have with respect to finding good probability laws on the message or key using all available information. To illustrate how this can happen with a too weak method, I will again take the example of the Linear Congruential Generator.

Suppose we generate a sequence (X_n) using the LCG (they will both verify good statistical properties that make them look uniform) So we have a, b and m, three integers such that:

$$X_{n+1} = aX_n + b \ [m]$$

We assume that a and m are coprimes (since this is the case for the C generator).

We only publish the sequence Y_n calculated by doing:

$$Y_n = X_n \ [m']$$

with $m' < m$.

The attackers will have access to a contiguous sequence of (Y_n). Let's say $m = 10^9$ and $m' = 50$. By having access to Y_0, I have reduced the domain of research of X_0 by 50. Now if I have access to multiple Y_n, then I will greatly reduce the possible domain for all the smaller X_n. Maybe, if the equation is solved I'll find one X_i and unlock the whole sequence.

By the way:

$$X_n = a^{-1}(X_{n+1} - b)[m]$$

So this is the kind of risk I imagine. I do not know if when someone has enough information about the syntax of the message or the probable content of it or the random generator he can reduce the scope of research, or find out what values he should start with using the result of the encryption function we have. Is there a mathematical theorem that says that when a message is in JSON, the last bit of AES encryption is 1 ? Or if the encryption of the key ends with 110 twice then the state is even ? I do not know whether or not the attacker has access or not to non-neglectable information allowing him to have a good enough probability to unlock the message in time. So RSA and AES are supposed to not allow that (by me), but I have no idea, in particular with the use of modern function approximation techniques (also misleadingly called A.I.). A comprehensive study of the available literature is necessary to build an enlightened opinion.

4.3.6 A cannot read or modify the payload of M

Remember that A receives:

$$F_M(b) = [a([b, e^e_M], K), e^e_S(K)]$$

He also has access to e^e_S and P.

A can do two things: invert the decryptions or perform a brute force attack.

We know that it is as hard to find d_S from e_S alone as solving large number factorization (which is hard). So unless factorization is easy, this is not possible.

The entropy from the $OAES$ padding supposedly prevents any statistical analysis. K is drawn randomly from an untracktable interval but using a public deterministic algorithm. The attacker knows the syntax for m but is it enough to draw any non-neglectable information ? Also, the attacker can have an idea of the probable values of b, but is the entropy of K enough to prevent the attacker from deciding which one it is.

There has to be some work on proving that the API call doesn't improve at all the probablity law the attacker have on b. I didn't analyze this part.

I believe the best way to protect ourselves from a brute force attack on the encrypted message is to make the encryption algorithm itself secret (sorry Shannon, I don't accept it to be known). How about we have a probability law on the encryption algorithm then ? This is nearly unsolvable. I think that so long as a solution exists, there is always a chance to access it. But how about the problem of finding an unsolvable encryption system ? Is there a chance to access it ?

With these concerns expressed (except the last comment), we will assume for the rest of the analysis that a probabilistic brute force is impossible. (I should try myself to attack this in an ideal and simplified setting in order to reach some conclusion (and clever people in general (and please don't keep it to you (the CIA has enough backdoors!))))).

This implies that the attacker doesn't have access to K, e_M^e and b for the rest of the interaction.

So he cannot just take b and modify some fields (did I mention it's a JSON ?), he has to do one from scratch, same for e_M^e and K. Which is equivalent to throwing out the request of M and running the request from the program P.

4.3.7 A cannot read or modify the payload of S

Now, S receives the request from M. S doesn't care about who M is, it cares about two things:

- The payload contains the credential (authkey or username/password) of the user to confirm that U sent the request.

- It sends the response to the same machine that made the request and only it can interpret the response

If we assume that these two properties are verified, then we will be sure that the user is the only one to read the response. The user/password combination is supposed to be known only to U (so U has a secure and secret password (not 123456)). The username/password and e_M^e are sent encrypted and A cannot have access to it. He cannot replace e_M^e while keeping the username/password. So when a message is sent using e_M^e, he will not be able to decrypt it (as we'll explain).

So if the authkey is sent to the machine that sent the username/password, it will be the only one to have access to it. And so only machines where the user entered his username/password have access to the authkey. This makes sending an authkey or a user/password possible only on machines used by a user. Which makes the user/password couple or the authkey *authenticating* information, which then can be used to evaluate authorization and so on.

Here is the function used to encrypt the information:

$$F_S(b) = [a(\omega(p, b), K), e_M^e(d_S^v(K)]$$

Now, here's something to think about. A can always send the requests he wants to S (because he has access to P). Plus, he can replace the response of S to M. This implies that A could pretend to be S. A knows what M wants (because p is public (which it should not)) and he can send false responses to M.

For instance, he can send his own authkey to M so that M modifies the data of A's account and A can read it by connecting to it. Which makes our previous reasoning wrong, because a machine can always send his authkey to another one. (This shows that one has to really be sure that he is 100% rigorous and thinks about ALL the possibilities without dismissing any).

Basically, what this example tells us it that another property has to be verified, which is that using the program properly, one cannot receive responses from anyone but the server. And yes, machines can still exchange information but through another interface. But the chain:

$$U - M - P - S - P - M - U$$

should not breakable. Something like:

$$U - M - P - A - P - S - A - P - M - U$$

should be impossible. So the attacker should not be able to compromise the program by sending another API response than the one expected. This is done by guaranteeing that only the server's answers to the machine's response can be accepted by the program. It has to be proven that any response received was made to the specific request made by the program, and was not crafted in another context.

The guarantee comes from three facts:

- A cannot have access to e_M^e
- A cannot have access to d_S^v
- The attacker does not have the ability to craft reponses that are syntactically correct after decryption without these two functions.

and also that brute force doesn't work (as explained above)

This means that only S can send syntactically correct responses and so any response that the program accepted without crashing is a server response. An attacker's replacement of the response is impossible.

4.3.8 *A* cannot exploit the filesystem

Suppose the attackers inflitrates the filesystem. The database is protected by a powerful password. The password is typed through a prompt at the start of the server. The private keys are stored in the RAM (and generated randomly at every start). The seed is typed though a prompt also and stored in the RAM. The source code is available though. But there is no dangerous information in it (the algorithm is public).

With respect to the specific application we have, there is one big problem and it's the images. We do not verify if they are malicious or not. And we leave them unprotected in some folder. This easy to solve.

- Verify the syntax of what is downloaded
- Encrypt the files before storing them using a prompt-entered RAM-stored secret key.

maybe just store the base64 format of each images in the database.

Other than that, the data is perfectly secure. But again, the */keys* path is the absolute vulnerability. So all the energy should be focused on that, whether from the attacker's perspective or the defender's.

4.3.9 Superfluous measures

We believe that the cost of attack should always be increased to the max. So any measure that make it harder to decode (despite the fact that the system is already secure) should be taken. This is why we hash the password before sending it, even with a public seed, why we use authkeys, why we use temporary data to generate them, and so on. And also we are somewhat future-proof with respect to the algorithms. (But in fact not)

4.3.10 Weak points

So we have delimited the only weak points of the system (supposing perfect implementation):

- The */keys* path
- The RAM or localstorage
- The algorithms expected properties
- A combination of mathematical knowledge and big data traitments yield very poor guessing strategies.
- THE PSEUDO RANDOM-GENERATOR (the most important (everything relies on it)
- The fact that the algorithms are public

The strategy for the first problem would be to have some sort of booklet distributed with all the public of different important websites.

The strategy for the second would be total obfuscation of the code with random operations performed constantly and the RAM memory constantly changing in an uninterpretable manner (with destruction of the clear source code potentially).

The strategy for the third is to verify the proofs of the algorithms and constantly changing them (twice a year or more).

The strategy for the fourth, is gathering mathematical knowledge and big data capabilities faster than the criminals.

The strategy for the fifth would be to have better generators. I am thinking about using chaotic movements like a double pendulum and truncating the state of it so that the next states are unguessable. Maybe a chaotic electronic phenomenon could be discovered and added to future computers to help with this. With respect to uniformity, I don't think it's necessary.

What we want is that no tractable range of possible values have a high enough probability of containg the value we seek. This could be a good future project.

The strategy for the sixth is to have a large arsenal of encryption algorithms that are used unpredictably.

These are not precisely the only strategies.

And of course, implementations will never be perfect so maybe the use of automatic logical verification of the validity of the program would be good. But then what properties should it verify. ERRORS ARE UNAVOIDABLE. And this is good because if the defender can make some, the criminal will as well.

5 Conclusion

We see that it easy to build a theoretically secure cryptosystem, and that the only limit is the property of the different cryptographic primitives used. This is going to be the main focus of any capable attacker in such a situation. So either one should solve the mathematical problem of decryption, or use information about the context, combined with good statistical information to have efficient guesses of the encrypted message.

But I think that one of the most important issue in practice is the error in implementation. There are a lot of exploitable vulnerabilites and it is the biggest risk because it's always highly probable. So my future task in terms of studying cybersecurity would be to read CVEs, as it basically constitute a database on the history of computer program bugs. Such a knowledge would make me much more aware of the possible attacks on my programs, thus making me a much better programmer that the average one.