General Data Protection Regulation (GDPR) practical impact on software architecture

Giuseppe "Pino" De Francesco, *SMIEEE*

Abstract — During the two years grace period after the General Data Protection Regulation (GDPR) was published, most companies have not prepared to become compliant. On the 25th of May 2018 GDPR comes into full force, and the typical last-minute patchwork is what we see going on almost everywhere, especially in the software and data processing industry. The effort made by all last-minute Data Protection Officers is to ensure the less costly patchwork to be implemented, leaving *de facto* the company exposed to huge fines established by the GDPR. In this paper I identify the mandatory steps to make a software architecture GDPR compliant, looking at the legal instrument and keeping in mind that we are required to implement "*data protection by design and by default*".

1 Introduction

ON the 27th day of April 2016 the GDPR was adopted and was then published in the Official Journal of the European Union the 4th day of May 2016 [1]. The clock started on the 24th of May 2016, counting down to the 25th day of May 2018, when that regulation will take the full force of law in all Member States of the European Union (EU).

The first thing to clarify is that the GDPR is a European **regulation**, not a **directive**. The difference is as follows:

DIRECTIVE	REGULATION
Defines what the EU Member States shall implement in their legislation. A directive is received and implemented in many different flavours by each Member State as each government sees fit.	It is a legal instrument that replaces any conflicting legislation on the same matter in all EU Member States. It has the force of law without any government intervention of EU Member States.

This difference is substantial, and still, in March 2018 I was told by many Security Officers that the 25th of May was not that important, because "*…anyway our government has to legislate to implement it so that it can take effect*". The feeling is that most of the companies are in denial about the GDPR coming into force and bringing fines up to €20 millions for those found non-compliant.

Once they understand that it is happening and it's just in a month, then the denial moves to what is necessary to become compliant, trying to find any possible excuse not to intervene properly.

This new form of denial is by far more dangerous because it introduces false confidence in what the interpretation of the legal text must be.

In this paper, we are now starting a very practical and pragmatic journey in those parts of the legal instrument that affect software architecture. I will avoid all IT related issues, leaving them for a different study as they merit a separate analysis due to the complexity of making an IT infrastructure GDPR compliant.

While reading this paper, there is one thing that must be in your mind every time you say "*do I really have to do this?*": imagine to be in a courtroom, on the stand, being sued for having breached the GDPR, and the plaintiff's attorney asks you the big question, "*Did you or did you not implement the State of the Art in Data Protection? Could you or could you not, in your professional opinion, do more to prevent the data leakage?*". Remember that in a court of law you cannot lie unless you want to bear heavy consequences, and the GDPR is all about implementing and keeping the *State of the Art in the field of Data Protection* in our solutions. Also remember that the plaintiff will hire a consultant to prove that you did not implement such *State of the Art in the field of Data Protection* in your solution, so you better think twice every time you are tempted to dismiss a data protection standard.

2 Data Protection

When we talk about Data Protection, we often do not stop to think about the actual meaning of it, nor its relationship with Privacy. Moreover, there is a tendency to protect the data after a breach, and this approach is no longer acceptable under the GDPR.

2.1 Privacy Vs Data Protection

The first notion we must digest is the difference between *Privacy* and *Data Protection*. There is often some confusion about these concepts, so the best option I have is to start our journey by clarifying them both.

2.1.1 Privacy

We, Data Subjects, have *right* to privacy, and this is granted

• *Giuseppe "Pino" De Francesco is currently a Solutions Architect with DXC.technology in Galway, Ireland, and co-owner of DFT Games Limited.*

by various legal instruments in many different flavours depending on jurisdictions, but the overall take away for us is that *Privacy is a Right*. All the legal instruments establishing this fundamental right, from International Conventions down to local ByLaws, have tried to guarantee in some form the protection of our privacy.

2.1.2 Data Protection

There is only one way to guarantee our right to privacy: *protecting our data*. This is a simple concept, easy to digest: *Data Protection* is the mean by which *Privacy* is guaranteed.

Now, a mean is normally realized by using tools, and in our case, this applies. Data Protection is implemented in software architecture and design by selecting tools, pattern and standards that facilitate Data Protection, and where the tool is missing we create one.

2.2 Data Protection by Design and by Default

This mandate established by Article 25 of the GDPR is the most crucial piece of legislation for us to understand and fully digest. This is also where almost all business actors I talked with (product owners, account managers, CEOs and so on) tend to look for some escape route. Let us dig into the first phrase of Art. 25(1):

> *Taking into account the state of the art, the cost of implementation and the nature, scope, context and purposes of processing as well as the risks of varying likelihood and severity for rights and freedoms of natural persons posed by the processing, the controller shall…*

Do we all notice the same way that part that reads: "…the cost of implementation…"? Now, the mantra business people keep reciting sounds like this: "It costs too much to protect these data using the state of the art of technology! The GDPR itself says that!" Ok, that is why we have recitals in all complex legislation, to help us to understand those passages that could drive to misinterpretation of the law and consequent fines. Recital 26 clarifies this point about costs to consider, and it makes sense from a technical perspective as well:

> *To ascertain whether means are reasonably likely to be used to identify the natural person, account should be taken of all objective factors, such as the costs of and the amount of time required for identification, taking into consideration the available technology at the time of the processing and technological developments.*

As we can see, the costs to consider are those representing the effort for an attacker to "*identify the natural person*". We, technical people, use this approach all the time, and our goal is always to ensure that breaking our security measures is so costly and entails so much development that no person in his/her right mind will ever even start trying to break in. Recital 83 uses a simpler form, like Art. 25(1), Art. 17(2) and 32(1).

Naturally, as it is clear to the reader, I am playing devil's advocate here. Certainly, it makes sense to consider the actual cost of implementation of the necessary measures to protect the data, but this applies to balancing the choices, reducing the cost by implementing a level of security adequate but among those that can be implemented with the less possible cost.

Nowhere in the GDPR, we have been given a choice to opt for not protecting the data because it costs too much in our view: such an attitude would cost heavily to the company in the case gets sued for non-compliance.

So, now that we have clarified that pain-point, let us understand what the meaning of Data Protection "*by Design and by Default*" is.

2.2.1 By Design

The wording of Art. 25 is all about establishing by design all the necessary measures to protect personal data. This clarifies that before discussing any *implementation*, the *design of the solution* must preemptively consider how to protect the data, and specifically applying the Six Principle established in Article 5(1):

- Lawfulness, fairness and transparency
- Purpose Limitation
- Data minimisation
- Accuracy
- Storage limitation
- Integrity and confidentiality

The technical means to achieve such compliance often mentioned in the GDPR are:

- encryption
- pseudonymization
- security of processing

We, the Architects, are tasked to architect and design our solutions implementing those principles, not moving forward if we are not satisfied that we did all we can given the context and the state of the art of the technology.

2.2.2 By Default

Stating "by default" seems redundant, but it is another very important concept that the legislator has established.

In our design, all behaviours must default to the strictest rules of Data Protection. Quite often in the past, the default behaviour in software was represented by the minimum set, the basics. Now we are called to ensure that if we don't know or don't have enough information at any point in our solution and its processes, then we apply the strictest rules, this to guarantee that no data will ever be processed or leaked by mistake.

> *The Privacy by Design approach is characterized by proactive rather than reactive measures. It anticipates and prevents privacy invasive events before they happen. PbD does not wait for privacy*

> *risks to materialize, nor does it offer remedies for resolving privacy infractions once they have occurred – it aims to prevent them from occurring. In short, Privacy by Design comes before-the-fact, not after.* [2]

3 Data Processing

A special mention is necessary for processing the data. Let us first make clear what *processing* means, especially when law interpretation comes into play.

If I take note on a piece of paper of a person name and address, I am processing that person's data. If I take that note and I put it in my wallet, I am again processing those data. When it comes to computing, then the notion is easy, taking the definition from the Oxford dictionary:

> *Operate on data by means of a program.*

Any *operation* involving the data in any way represents Data Processing, not just the use of data in a computational algorithm relates to those data. This is stressed in the GDPR in multiple passages: we must limit data processing to the *strict minimum* that is necessary to carry out *the operations that have been authorised* by the data subject.

In principle, if for a given operation I need just first name and last name, I am supposed to retrieve only those from the database, not a full record including email and address: I am supposed to create a data view that satisfies that minimalistic data access.

This because the more data we move around in memory the more data we put at risk in case of an attacker watching our memory operations, and this brings us to data encryption, but first it is worth to clarify the meaning of State of the Art.

3.1 State of The Art

We have seen that the GDPR mandates to use the state of the art of current technology. This is mentioned in four places: Recitals 78 and 83, Articles 25(1) and 32(1).

The definition of State of the Art is a moving target: it changes over time. The GDPR established that Data Protection must be guaranteed by implementing the *State of the Art of technology at the time of processing*. Therefore we have now a legal obligation to keep our software solutions current, fully up to date. Implementing the State of the Art of technology for Data Protection today does not guarantee that in three months time the solution is not obsoleted by new findings.

3.2 Database encryption

The fact that we must architect, design, implement and maintain using the State of the Art of technology has a huge impact on the database we use.

There are two main levels of DB encryptions: *at rest* and *in memory*. The encryption at rest was the state of the art until live DB encryption was introduced, obsoleting *de facto* the simple at rest option because that encompasses both, at rest and in-memory data encryption.

While I write this paper, there are only two databases that I know of featuring both encryptions, Microsoft SQL Server starting at version 2016 and Oracle Database Server with the Transparent Data Encryption module, so at this point in time, any other database seems to be non-GDPR-compliant. This represents a big problem for us because many solutions are based on non-compliant databases, and there is no known roadmap for those to become compliant. In truth, some free DB will not have enough funding ever to be compliant, so they are necessarily destined to be dropped from any solution dealing with European citizens' personal data.

The situation is different for NoSQL databases: none of them, in my knowledge, currently offers both at rest and in memory encryption. Therefore encryption at rest is the state of the art feature as for now, encompassing many of the most commonly used NoSQL DBs, like Mongo DB (only Enterprise with WiredTiger Engine) and Cosmos DB.

3.3 Database structure and pseudonymization

Once we have selected the correct database, we have to think about its structure. When we talk security, we all know the old say about not keeping all your eggs in one basket, and that applies quite well in this case.

Designing databases using the 3rd Normal Form (3NF) or Boyce-Codd Normal Form (BCNF) seems to be no longer enough. The principle of *pseudonymization* established in the GDPR cannot be easily implemented within our usual DB design practices.

The principle of pseudonymization is known to us with the more familiar term of *steganography*. What we must do is to hide information by replacing it with a pseudonym, thus hiding the data in plain sight in a way that the attacker sees different and meaningful information in its place.

Let us take this number, imagining it is a valid social security number:

1228475

Storing this as a steganographic information can be represented, for instance, as a list of US cities:

Rome, Denver, Washington, Arlington, Lebanon, Madison, Greenville

This results by applying the following translation table:

Original	1st Occurrence	2nd Occurrence
0	Clayton	Auburn
1	Rome	Hudson
2	Denver	Washington
3	Springfield	Franklin
4	Lebanon	Clinton
5	Greenville	Bristol
6	Fairview	Salem
7	Madison	Georgetown
8	Arlington	Ashland
9	Dover	Oxford

Obviously, we need more columns for the digit occurrences, but this gives you a practical example of using steganography to apply pseudonyms to data, to the end of making them useless to an intruder.

This implies that the column where we store the social security number should be defined in the DB something like *VisitedPlaces*, so we close the circle about fooling the intruder.

We also must tackle the issue of where we store the translation table. Storing that in the same DB along with the data we apply the table to would be an error. Therefore we should put the translation tables in separate storage, possibly an encrypted NoSQL DB.

One more thing needs to be told to secure the steganographic system: remember to replace the translation table regularly. To make this operation efficient you need a column to signify which translation table has been used to encode the column, and this can be any conventional value. For instance, if you replace the table every week, then you can have the week number stored. This is necessary because replacing the translation table entails parsing all the records in the table, and it could be a long operation executed at low priority. Therefore if the table is significant, at some point we will have some value still encoded using last week table, and we have to know this unless we lock the whole table while we replace the encoder, but that would be bad practice, especially for big tables parsed at low priority.

3.4 Issues with in-memory data

We know how often an intruder is nothing but a memory observer, a program that monitors the RAM looking for data it can recognise and snatch, often referred to as *memory sniffers* or *memory scrapers*. Against these attacks, we have the DB in memory encryption, but at some point in time we do have to retrieve the data to use them computationally, and that is the moment we become vulnerable again. This happens even sooner when using a NoSQL DB because, as we have determined, they only offer encryption at rest.

It is evident that we cannot use the data if we do not have them in clear text, so at some point, we will be anyway vulnerable if a *sniffer* is watching our process, therefore the trick is to keep the data in clear text for the shortest possible amount of time. In case we need those data for multiple processes then we must keep it in an encrypted memory area and decrypt them only for the microseconds necessary to our computational need.

Nowadays technology allows us to encrypt and decrypt on the fly at a non-significant time cost from a human user perspective. Therefore this is the best approach we can apply using today's cryptographic state of the art libraries.

3.5 Sensitive data

Sensitive data are listed in Article 9(1):

> *"racial or ethnic origin, political opinions, religious or philosophical beliefs, or trade union membership, and the processing of genetic data, biometric data for the purpose of uniquely identifying a natural person, data concerning health or data concerning a natural person's sex life or sexual orientation"*

When in need to process this type of data we must ensure a better and stronger cryptography, so the RSA key pair must be no less than 2048 bits, and 512 bits for block cyphers.

The most important thing about data, and especially about sensitive data, is to ask ourselves: do we need to process them? Are these data necessary to carry out the service? Do we have an explicit consent to process the data subject's sensitive data?

Do remember that we can no longer collect data with generic or implicit consent: we must ensure that the data subject is fully aware of the fact that we are processing these data, (s)he must explicitly consent to the processing using a form that also clarifies the scope and timeframe of the data processing. Moreover, we must securely store the consent, and in case we must process the data for a longer period a new consent is necessary, especially for sensitive data.

3.6 True random numbers generators

Every time we mention cryptography, we are implying the generation of random numbers, and those shall be *cryptographically secure*.

State of the art for cryptographically secure random numbers generation is hardware cards using shot noise, photoelectric effects and other electron or photon-based systems.

These cards should be compliant with NIST SP800-90 B [3] and C (still in draft) [4], like the Comscire PQ4000KS [5] or SwiftRNG Pro [6]. Given the low cost and high availability reached by these true random number generators, there is no longer any justification to avoid implementing the technology: we can safely state that the era of pseudo-random numbers based on algorithms instead of dedicated hardware is over, hence we must adopt this technology to be compliant with the GDPR.

3.7 Integrity and security of processing

Four are the principal elements that must be considered to guarantee data integrity and security of its processing:

- Always implement referential integrity constraints at the database level
- Never delete physically any data unless they have reached the limit of their storage period: use logical delete only (a boolean column *Is_Deleted*)
- Define a clear backup policy and comply with it
- Use Test Driven Development and have a QA team working in isolation from the development team

None of the four points above is more important than another one: all have a considerable impact on being able to guarantee that the system we architect, design and implement is as robust as the GDPR requires it.

If in a court of law will be proven that the product has weak data integrity and that there was no standard QA during all phases of the software lifecycle, your company

will most likely be found in breach of the GDPR.

3.7 Securely storing passwords

Passwords have been a serious problem and always underestimated when it comes to storing them in a safe mode. Most users have the habit of using the same passwords in rotation on all their password-protected resources, and this is an understandable behaviour because otherwise, the average user would end up with an average of 20

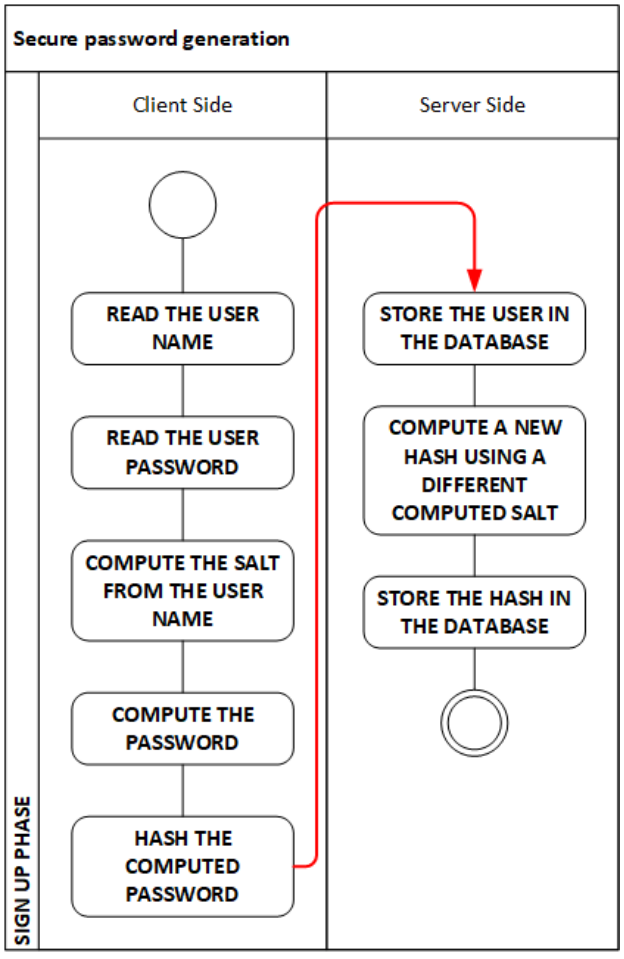

Figure 1 – Sample algorithm for secure password generation

passwords to manage, and there is still little confidence in the public in the password managers on the market.

Due to this habit, if one password is hacked then the user must deal with a breach in many services, not simply the one that suffered the hacking. It is our responsibility to make it hard to hack the password, therefore we cannot just hash it as it is common habit in most architecture: there are too many excellent brute force hash solvers available, and because the computing power available to any hacker is growing exponentially we cannot leave the passwords exposed by simply hashing them with a standard hash algorithm, not counting online brute-force with dictionary helpers resources like https://hashkiller.co.uk/, https://crackstation.net/ and hundreds more, almost all available online for free.

So, what if we must play the role of the identity provider, having to store users' credentials? I am offering here (Figure 1) a sample solution, an algorithm I published a few years back. The first password hashing always happen on the client side, so that the original password chosen by the user is never sent over the wire in clear text. In a context of a signup process, we have both a username and a password; therefore in the algorithm, I propose to derive the salt from the username in a creative way, like, for instance, the following (taken from my previous article).

Imagine a Mr John Doe signing up and the email being the username as well (for simplicity):

john.doe@somedomain.com

moreover, he decides to use as a password

MyPassword

The first step on the client where our new user is signing up is to compute the *salt*. To do so first, we note the ASCII value of the first letter in the username:

$$j = 152$$

Being 152 an *even* number we arbitrarily decide to pick from the username all the characters placed in an *odd* position, so we get our raw *salt*:

jh.o@oeoancm

So, let us now get the *MD5 hash* of this salt to get a better form of it to use:

c4211ead299f2bd80a3465ab9be18c05

Now we add the salt to the password, getting the following

MyPassword * c4211ead299f2bd80a3465ab9be18c05

I am adding " * " in between just as an added complexity and better presentation in this sample. We now have something complex enough: let us translate this into a *Base 64* encoded string of the *SHA512 hash* of it:

723b6f133818b87215e9f476350d06e55815c8de00fc
13af58aa94dee4c66398b441b2a1060dd1e2f7f82dd3a
b2a420ee4245b943fc7721ab59b765f84eefa26

This is a typical Base 64 string, with all alphabetic characters in *lowercase*. At this point let us make it more interesting, so, again because *j* is an even number, we arbitrarily decide that all characters in *odd position* shall be *uppercase*, getting the following password:

*723b6f133818**B**87215**E**9**F**476350d06**E**55815**C**8**D**e00**F**c13A*
*f58A**a**94**D**e**E**4**C**66398**B**441**B**2**A**1060d**D**1**E**2**F**7**F**82d**D**3A**b**2*
*a420e**E**4245b943f**C**7721a**B**59b765f84**E**e**F**a26*

This represents the actual password the client software will send to the server.

As you can see from Figure 1, on the server we do something similar, *but not identical*, just any other creative way, like picking the 11th letter of the hashed password and do something based on that being even or odd in ASCII, or any other rule you come up with.

The best approach is to create an algorithm unique to your company so that even gaining access to the database it will be impossible to try to reverse-compute the original value that the user picked as a password.

3.8 About software libraries and access

I did refer to algorithms for steganography, in-memory encryption, and password protection, and you will end up with many more before your solution is complete, and this will bring up another problem: the security of the implemented algorithms.

It comes without saying that the libraries offering the Data Protection functionalities must be *heavily obfuscated* so that the cost to disassemble them would be too high for any hacker or organisation to afford. But what about the most dangerous person we can deal with? One might ask "Who is that person?"; only one is the worst regarding dangerousness: *the disgruntled employee*.

Let say that Mr Doodle is a chief analyst in our data analysis department and he got fired due to workforce reduction. He is angry and has access to the data, so he dumps all the databases we manage on Torrent. Well, because we implemented the measures mentioned before, from a Data Protection perspective we do not care much because we will incur in no consequences in respect of the GDPR: all data are steganographically protected and no clear password is stored anywhere. Thus nobody can be even identified. Well, this is true if our Mr Doodle has no access to the algorithms we use and no access to the source code of their implementation. Otherwise, he can push those as well on a Torrent and then we will be in some serious trouble.

It is very important that we take all relevant precautions to ensure that no single person has full knowledge of the Data Protection algorithms nor access to the full implementation source code. This implies that the development of the implementations must happen atomically, having different teams developing different parts of them, leveraging functionalities offered by other *obfuscated libraries*, so that it would be impossible to figure out what the Data Protection code is doing. This means that in case of leaking of the algorithms only the architect that designed them or the custodian of the files would be the source of the leak, making simple to identify the culprit in such case.

Here as well I am sure you are thinking "*Isn't this too much?*" and again I invite you to figure yourself on the stand in a court of law being asked "*Did you or did you not implement the State of the Art in Data Protection? Could you or could you not, in your professional opinion, do more to prevent the data leakage?*".

Setting up the correct process and access levels is a simple thing; therefore, it is not justifiable not to put those security measures in place in any company from medium size and up. A small company should take it seriously, but adapt this to a reasonable infrastructure considering the available means.

3.9 Data erasure (Right to be Forgotten)

Another technical trouble comes from the right to be forgotten established in Article 17 of the GDPR. This is a problem that has no simple solution because we must have a data backup plan implemented to be compliant with Article 32 of the GDPR, so when a data subject requests to be forgotten (full erasure of all his personal data) we have a problem.

The solution, in this case, cannot be technical, because the cost of safely erasing data from backups is too high. The GDPR allows exceptions as long as the user consented to those in the first place so, in my opinion, when we collect the data subject's consent we shall have a clear clause, explicitly accepted, where we clarify that in case of the consent to be revoked exercising the Right to be Forgotten, the data on backup media will stay and will be destroyed along with the backup media when the retention time limit is reached.

A clear process must be in place in case the data are restored to ensure that none of those data will be restored along. It is reasonable to fully restore a backup and have a batch process immediately after the restore operation executed to delete the data that must be erased.

I know that this is not a clean solution, but the alternative would be to process all the backups we have in a sandboxed environment, restore them, then erase the data in question and make a clean backup again and destroy the original one: the cost of this operation would be unreasonable in most cases.

3.10 Right to Data Portability

The GDPR establishes for data subjects the right to have their data exported in a portable standard (Article 20). It goes even farter, by establishing that, where technically possible, if the data subject needs the data to be ported to another controller, that process shall be executed (Article 20, paragraph 2).

In simple terms, this means that if I have a Google Plus account and I want all to be moved to Facebook, where Facebook has an endpoint to allow such a data transfer, Google will be compelled by the GDPR to execute the transfer of all the data to Facebook.

This doesn't present any technical problem, but, because all the data are scrambled and encrypted, we must put in place a process to allow the exercise of the Right to Data Portability, and it has to be in place before any user exercises this right, otherwise the time to execute it risks to go beyond any reasonable delay and the data subject will be entitled to a compensation while it'll be likely for our company to be fined for not having established a proper mean to export the data in a timely fashion in conformity with the GDPR.

4 Client and Server

Most, if not all, data processing involves at least two tiers: a client, usually running on the operator device, enabling the operator to query, add, amend, or delete data, and a server, where the data are stored and where most of the computation is performed.

Sometimes there are multiple servers, where usually the server interfacing the human client is, in turn, a client of

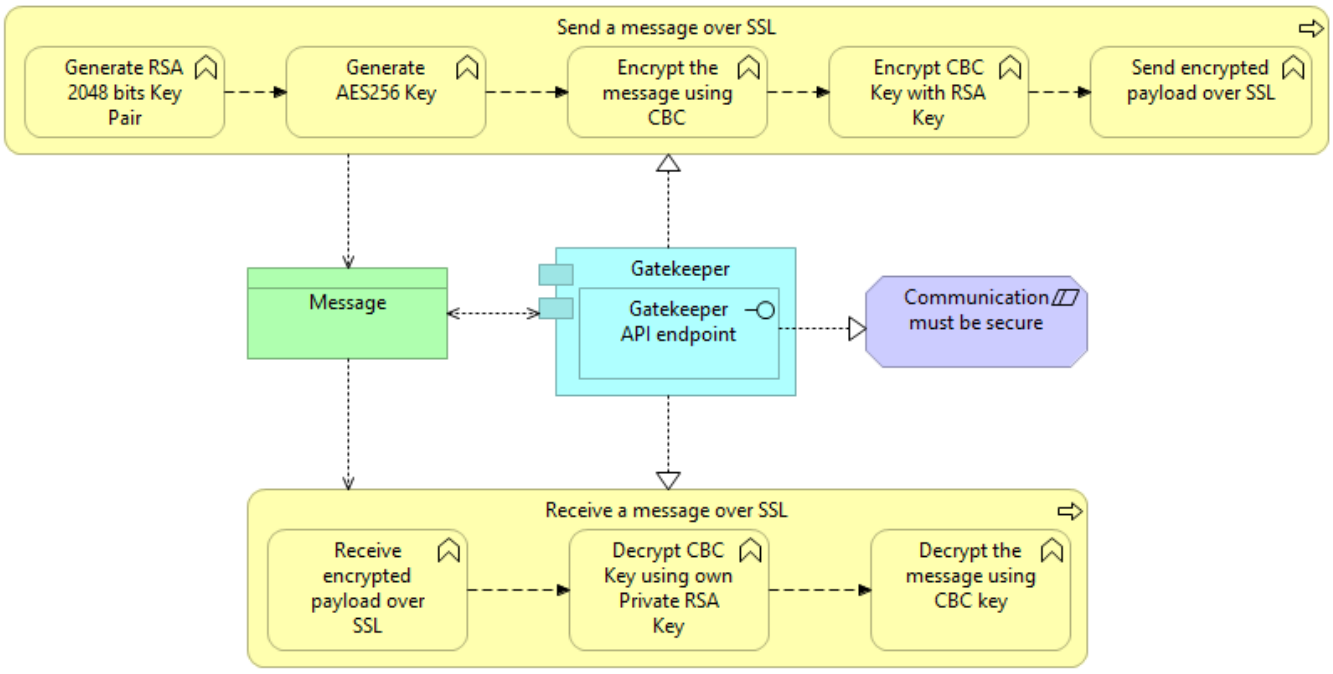

Figure 2 – Enhanced Gatekeeper implementation

other servers. All this talking between machines has one major implication: data are exchanged over a network, which, by its very nature, is almost never secure.

4.1 An enhanced Gatekeeper

As I mentioned in the introduction, in this paper I will not touch the matter of securing the IT infrastructure, so let us go up a few logical levels, up to the Application Layer, I will illustrate how to secure the data we exchange.

The obvious first step is to design the application to leverage the Transport Layer Security (TLS, ex SSL, Secure Socket Layer), so we are assuming that our Client-Server architecture is based on a RESTful approach, not on a transport layer connection, otherwise we have to re-invent the wheel, and that is not something we like to do, especially due to the costs involved in establishing and then maintain a custom security protocol based on raw TCP/IP packet exchange.

In Figure 2 we can see the process diagram of my own implementation (which I put in the public domain a few years ago) of the known Gatekeeper Design Pattern. In my vision the Gatekeeper has one single REST endpoint, expecting a POST message structured as follows:

Field	Value
Sender Public Key	The sender public key that the receiver will then use to encrypt the CBC key used to encrypt the answer
Encrypted CBC Key	This is a SHA256 key used to encrypt the Body. This key is encrypted using the other party's RSA Public Key
Body	This contains the actual payload (API call and body) encrypted with the CBC key

The SHA265 CBC (Cipher Block Chaining) key is generated anew every time we send a message: never recycling this key is paramount to securing the data transferred between Client and Server.

Two are the assumptions I make in this algorithm:

- The Server's RSA Public Key can be retrieved from a location known to the client: this allows to regenerate this key every day.
- The client will generate a new RSA Key Pair every time it is executed

The body is the actual API call that the Gatekeeper will relay, acting as a proxy. The usual structure of the encrypted body is a JSON structure carrying four fields, representing the API call to be forwarded by the Gatekeeper, as shown in the following table:

Field	Value
API Query	The legal query string, something like */api/user&param=xyz*
Body	The eventual payload to pass to the API endpoint
Verb	The HTTP Verb to use (GET, POST, PUT and so on)
Headers	The headers value for the endpoint, especially necessary to pass on a JWT token for authenticated services

The Gatekeeper will add the base URL, prepare a new message using the proper header and body information provided and place the call on behalf of the client on the internal API server.

Naturally, the Gatekeeper will rely on the result back to

the client using the same approach. Therefore the raw answer from the internal API Server will be encrypted using a new CBC key, and the key will be encrypted using the client RSA Public Key.

This approach will guarantee state of the art in data protection, at least while I am writing this paper.

Naturally, the Gatekeeper can be extended to also play the role of an API Gateway simply by implementing a set of Access Control List (ACL) to the internal APIs available so that the Gatekeeper will allow or deny access based on the rules set in said ACL.

4.2 Never trust the client

We all have been trained to repeat the mantra "Never, ever trust the Client", still it is good to touch this subject, especially in the context of the liabilities established by the GDPR.

We must remember in our design that any client can easily be hacked, a client device can be lost or stolen, so we must design our solutions assuming that our clients will fall in the hands of an attacker.

The consequence of this is that no business logic will be executed on the client, and especially no personal data will ever be stored on the client device. We also have to pay attention to always encrypt in-memory strings and also design the user interface in a way that on each screen there is the least possible identifiable data and the way it is displayed should be easy to understand only to a human, this in anticipation of the device being contaminated by a *screen-scraper*, a malicious software capturing screenshots and trying to extract meaningful information out of that.

The screen-scraper protection can be implemented by designing the user interface using uncommon fonts, variable for each data type, so it will be easier and meaningful to the user but will make little sense to a program trying to decode the content of the screen.

Another measure, optional but very effective, worth implementing in case the client consumes sensitive data, is to ensure the use of screens with a low viewing angle and to leverage the LCD colour distortion as suggested by Harrison and Hudson in their paper [7], so to scramble the screen content when viewed by anyone that is not in front of the monitor.

5. Conclusion

In this paper, I have explored the extent of the impact of the GDPR on software architecture and proposed practical solutions to ensure full compliance with the regulation. The proposed solutions have been verified in real-world applications. Thus they represent a pragmatic approach to implement compliance. Because the legislator in writing the GDPR used a form that has considered the pace technology evolves, we shall establish a protocol for a periodical revision of our solutions to ensure that they are still compliant. Companies shall reflect these protocols in their ISO certifications processes and in their software maintenance budgets.

References

[1] European Union (2016). *Regulation (EU) 2016/679*. Brussels: European Union, pp.1-88. (URL link http://bit.ly/GDPR_Pdf).

[2] A. Cavoukian, "Privacy by Design - The 7 Foundational Principles," Internet Architecture Board, 02-Nov-2010. [Online]. Available: http://bit.ly/AnnCavoukianPhDPbDD. [Accessed: 05-Feb-2018].

[3] M. S. Turan, E. Barker, J. Kelsey, K. McKay, M. Baish, M. Boyle, NIST, and NSA, "SP 800-90B, Recommendation for the Entropy Sources Used for Random Bit Generation," SP 800-90B, Entropy Sources Used for Random Bit Generation | CSRC. [Online]. Available: http://bit.ly/SP800-90B. [Accessed: 17-Feb-2018].

[4] E. Barker, J. Kelsey, and NIST, "SP 800-90C (DRAFT), Recommendation for Random Bit Generator (RBG) Constructions," SP 800-90C (DRAFT), Recommendation for RBG Constructions | CSRC. [Online]. Available: http://bit.ly/SP800-90C-DRAFT. [Accessed: 02-Apr-2018].

[5] "PureQuantum® Model PQ4000KS," ComScire. [Online]. Available: https://comscire.com/product/pq4000ks/. [Accessed: 17-Jan-2018].

[6] "SwiftRNG Pro - TectroLabs," - TectroLabs. [Online]. Available: https://tectrolabs.com/swiftrng-pro/. [Accessed: 17-Feb-2018].

[7] C. Harrison and S. E. Hudson, "A New Angle on Cheap LCDs: Making Positive Use of Optical Distortion," Chris Harrison | A New Angle on Cheap LCDs. [Online]. Available: http://chrisharrison.net/index.php/Research/ObliqueLCD. [Accessed: 14-Apr-2018].