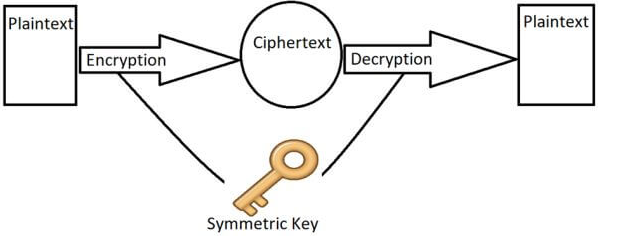
**PII\_Security methdology**

Although out databases themselves run on encrypted SSD drives, this does not solve our problem in relation to PII data stored there. The encrypted SSD drive prevents somebody from copying the database, but doesn’t prevent anyone with authorized access from reading PII data using something like SSMS.

In order to prevent random IT people and developers from reading PII, we must encrypt the data in each column that contains PII. The data in these columns should use symmetric encryption for the fastest possible encryption/decryption. When we encrypt data, it is converted into Ciphertext. This is a binary representation of the plain/clear text that is encrypted. The same symmetric key is use to both encrypt and decrypt the data. This binary representation is commonly Base64 encoded in order to copy the encrypted data into a text file or other location that doesn’t allow binary data.

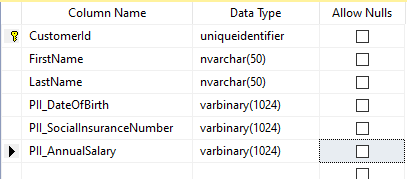


The problem we have through is that you can’t store Ciphertext is anything but a varbinary or varchar column. So if you have a decimal, date, or integer column that you need encrypted, you’re out of luck. The database will not let you put the wrong type of data (binary) into a column of the wrong type (like integer). The solution to this problem is to create a sister column that IS varbinary for every column that you need encrypted. When creating this sister column we use the convension “PII\_ExistingColumnName”. So, we simply prefix the new column name with “PII\_”.

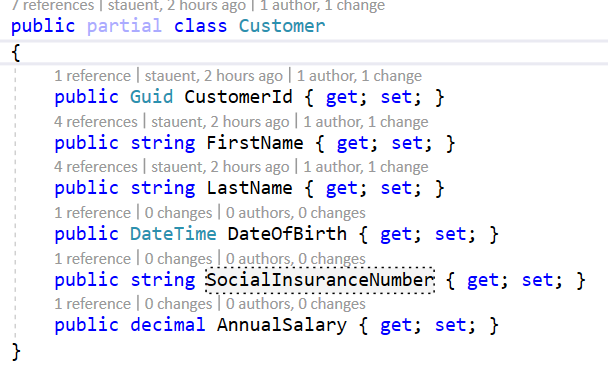
* In order to copy the plain text data into the encrypted sister column, we follow these steps:
* The column containing plain text must be read
* The plain text data is encrypted
* The encrypted data inserted into the sister “PII\_” column
* Once all PII columns are created, read both the plain text column along with the decrypted data from the sister column in every record to ensure both values are identical. This ensures you didn’t miss anything and there were no problems encrypting data.
* Delete the columns that contained the pain text. Now only the encrypted data remains in the “PII\_” columns.

During this process your DTOs that you use to read/write data, will have to be modified to include the new “PII\_” columns. Once the plain text columns are removed from the database tables, they should be removed from the DTOs as well. A new partial class should be created that extends the DTO. This extension partial class will be used to expose the encrypted as plain text, using the names of the properties that were deleted. In this way the original DTO reads/writes data to the database in encrypted format. Only the extension partial class decrypts the “PII\_” column data and exposes it as plain text using the same property names as before.

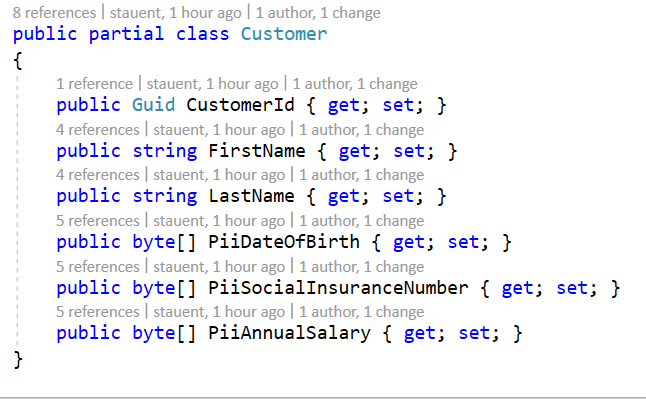
Here’s an example of a table that I converted and is capable of storing encrypted PII column data:



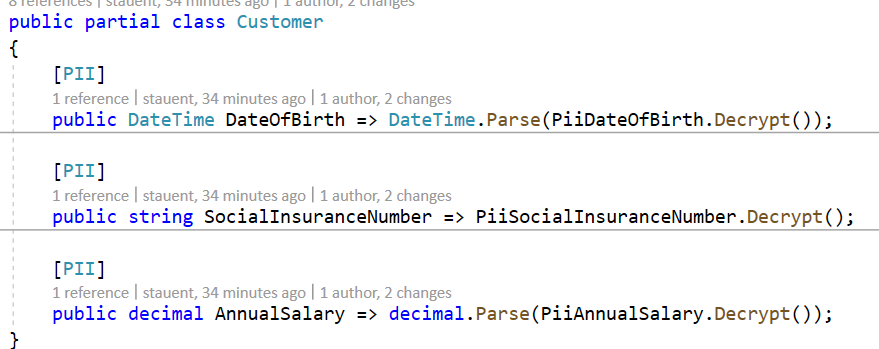
Before we removed the clear text columns from the database and replaced them with “PII\_” columns, here’s the DTO that was used to access the database. The DateOfBirth, SocialInsuranceNumber and AnnualSalary were available to see in plain text by the developer by using these properties.



After the clear text columns were removed and replaced with “PII\_” columns, this is what the same class looks like. Notice that all PII properties are of type byte[] so that they can read/write the varbinary data in the database. We could have used varchar instead, but then we’d have to convert the byte[] to Base64 in order to store it. Base64 is about 33% larger than the original binary data, so to save time and space, we store the data in the database in varbinary columns.

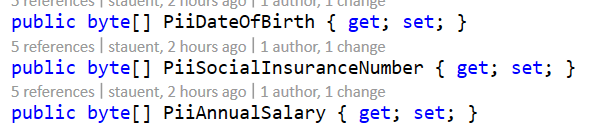


I now create a second partial class with the names of the original Customer class that we deleted. In my sample code I use entity framework to reverse engineer the database. That’s what produces the Customer class above. It’s ok, if you hand code your DTOs and use dapper, so long as you follow the naming convention. This methodology allows your primary DTO to be generated by some tool (like entity framework), and not have it overwrite the code you’ve written in the second partial class.



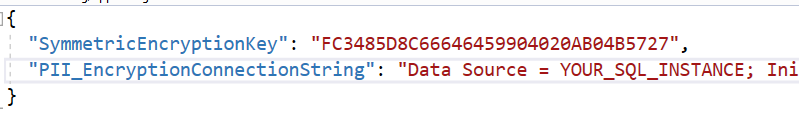
You’ll notice that each property is adorned with the [PII] attribute. This is NOT a requirement. I simply add this attribute to allow me to easily find the properties in my DTOs that will contain decrypted data.

As you can see, what I do in the second partial class is I decrypt the value stored in the sister “PII” column that contains the encrypted data. All of your existing code should continue to work because as far as it’s concerned, then only thing that’s changed is that you’ve introduced a couple of new properties:

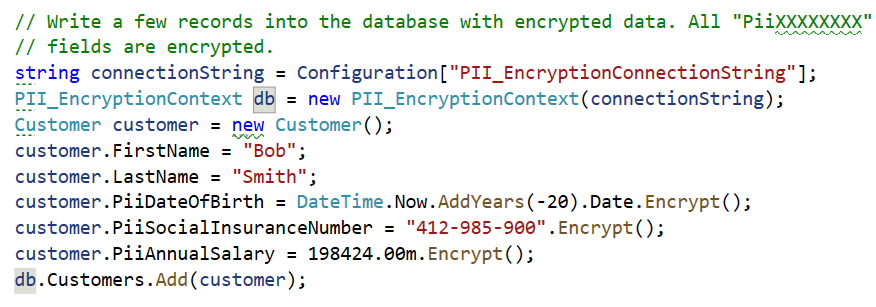


At this point, as far as your original code is concerned, the “Customer” class that it was using a month ago is the same one it’s using now. All of the properties still exist, they’re just generating data differently under the covers.

In order to run the reference application, you’ll need to restore the “PII\_Encryption.bak” database that is in the “Database” folder of the repo. Next, you’ll need to modify the “PII\_EncryptionConnectionString” property in the appsettings.json file to use your sql server instance name, user name and password.



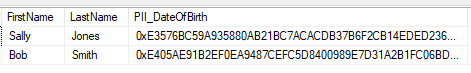
Here’s an example of how we write a record into the database that contains encrypted columns. The “PIISecure” class provides extension methods for encrypting and decrypting data for properties.



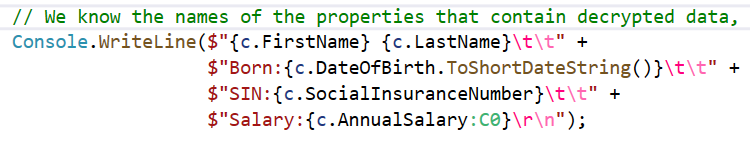
When the data is read from the database, the PII columns of the Customer class will contain the encrypted data.



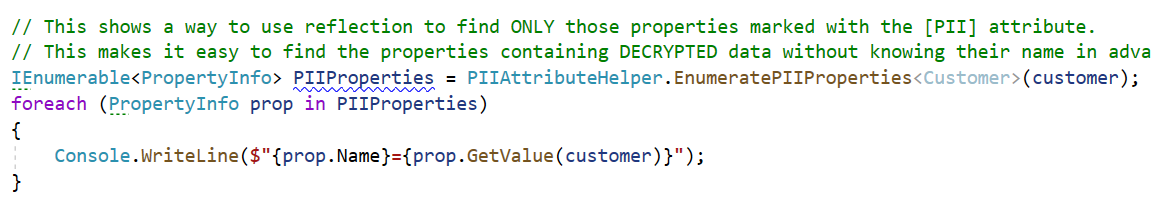
If you take a look in the database at this point, you’ll see only encrypted PII data:



The sister columns without the “PII” prefix use the properties in the second partial class to display the decrypted values:



And as a final example, I use the [PII] attribute on the partial class properties (along with reflection) to enumerate all properties that contain decrypted data, without having to know the property name.



As you can see the database only contains encrypted data and the primary “Customer” DTO also only contains encrypted data. Only the second partial Customer class provides access to the decrypted property values.

Assume we create a “Claim” called “PII Authority” and assign it to a couple of users. It is now simply to write a method in a controller (or an entirely separate controller) that can only be accessed by users with the “PII Authority” claim. These controllers/methods are the only ones that would use the properties exposing the decrypted data. This makes it very simple to lock down who can and can’t see PII.