Healthcrypt

Marcus A. Zimmermann
Marcus.Zimmermann1@Marist.edu

May 10, 2020

Abstract - As technology has progressed, it has become more reasonable to store patient information in electronic medical records, allowing healthcare providers to leverage the profound speed and storage capabilities of emerging technologies. However, the transition to electronic storage poses new challenges to the long-standing practice of strict doctor-patient confidentiality. To keep patient information as private and secure as possible, it is standard to encrypt electronic medical records before storing them. This paper documents the development of Healthcrypt, a system designed to meet that standard.

Introduction

Healthcare providers have always taken extreme measures to ensure doctor-patient confidentiality, using systems like on-premise, lock-and-key file repositories to prevent the disclosure of personal information. However, as technology has progressed, it has become more reasonable to store this information in the form of electronic medical records. In other words, instead of maintaining a vast, tangible file system, traditional patient files have been systematically digitized and stored in electronic mediums to take advantage of the profound speed and storage capabilities of emerging technologies. These systems are fast, efficient, and scalable well beyond traditional storage mechanisms. However, they pose new, significant challenges to security as well.

The inherent connectedness of electronic storage systems broadly extends the attack landscape, putting sensitive information at risk of sophisticated, novel forms of theft [6]. To strengthen the defense posture of electronic storage systems, it is standard to use state-of-the-art encryption methodologies as part of the filing and storage process, mitigating the potential harm of inappropriate and unwarranted access to sensitive information.

The remainder of this paper is organized as follows: The Background and Motivation section covers a brief history of healthcare security, as well as the motivation for the development of Healthcrypt; The Methodology section discusses the design and development of Healthcrypt, including the environment in which it was built and the most critical extensions and libraries that were used; the Experiment section describes Healthcrypt's performance on a simple demonstration of security; Lastly, the Discussion and Conclusions section describes observations made during the design, development, and analysis of Healthcrypt.

Background and Motivation

As early as the 1960s and 1970s, academic medical centers, government institutions, and private industry recognized the potential of electronic medical records and the "benefits to industry-wide standards," noting the need to create "organizations to tackle the broader issues that would facilitate the widespread use of electronic medical information" [1]. These broad-sweeping standards naturally included concerns related to the use of and access to electronic storage systems, as "increasing the connectivity capabilities also creates cybersecurity risks" like "unauthorized access to patient health information" and "changes to prescribed drug doses," not to mention many other causes for concern [6]. Unauthorized access to, and manipulation of, patient information is not something to be taken lightly.

The Security Rule under the Health Insurance Portability and Accountability Act (HIPAA) "establishes national standards to protect individuals' electronic personal health information that is created, received, used, or maintained by a covered entity. The Security Rule requires appropriate administrative, physical and technical safeguards to ensure the confidentiality, integrity, and security of electronic protected health information" [4]. That sounds comforting, but there is a minor detail pertaining to the implementation specifications of these technologies that may worry patients and healthcare practitioners alike. Each Rule contained within HIPAA is labeled as either 'Required' or 'Addressable.' What's the difference? Required Rules are self-explanatory. They are required. Addressable Rules, on the other hand, introduce opportunities to fall short of standards, including those related to security.

Addressable Rules, such as the practice of encrypting data at rest and data in transit, are necessary when "risk analysis shows such risk to be significant," the judgment for which is deferred to the covered entity [4]. In other words, implementing some form of encryption is left to the discretion of the health care provider. So, depending on the circumstances, your personal health information may not be encrypted at rest or in transit.

It is worth noting that the United States is among the leading countries when it comes to upholding encryption standards in healthcare [5]. This includes enterprise-level encryption strategies, which certainly offer some reassurance when it comes to the safety of patient records. However, we're not perfect. Not all data are encrypted, and data breaches are still well within the realm of possibility.

To better understand the systems storing and exchanging electronic medical records, the remainder of this paper documents the design, development, and analysis of Healthcrypt, an application that intends to offer a rudimentary exploration of encryption within the context of healthcare.

Methodology

Healthcrypt was built with Flask - a micro web framework - and a variety of extensions. Flask was chosen because it is lightweight and provides the ability to seamlessly integrate extensions as if they were built into Flask already. In other words, "Flask aims to keep the core simple but extensible ... Flask can be everything you need and nothing you don't"[8]. For sample, proof-of-concept applications, few frameworks are better designed than Flask, making it ideal for the development of Healthcrypt. In the paragraphs that follow, we will discuss the environmental conditions and configurations provided by Flask and implemented in Healthcrypt; the major extensions leveraged by Healthcrypt, including Flask-Login, Flask-WTF, and Flask-SQLAlchemy; and, lastly, Healthcrypt's use of the Fernet symmetric encryption and decryption system, which is used to keep physician and patient information secure.

First things first, Healthcrypt was built inside a virtual environment. Of course, this is standard procedure for many developers. Isolating development environments with venv - a virtual environment builder packaged with Python 3 - allows us to isolate our development environments and share projects with a reduced risk of compatibility issues. With the virtual environment generated and activated, we can begin building the application directory and installing packages fundamental to Healthcrypt, including Flask itself.

Flask provides default application structures to support large applications with many extensions. More specifically, large applications are built using a package pattern in which initialization files, or "__init__.py" files, are responsible for high-level imports, configurations, and initializations, and commands executed in the active virtual environment create environment variables that inform Flask of the appropriate entry points to run the applications. The package pattern facilitates the production of modular application structures, further encouraging the separation of files responsible for routes, models, forms, and dynamic templates. With this folder structure in place, and the "FLASK_APP" environment variable appropriately set, users can issue the command "flask run" in the active virtual environment and navigate to "localhost:5000" to interact with Healthcrypt. This simple, out-of-the-box support for large applications with many extensions allows developers to focus on the most important aspects of their applications.

The simple core of a Flask application facilitates the seamless integration of various extensions and libraries. Of all those used to power Healthcrypt, Flask-Login, Flask-WTF, Flask-SQLAlchemy, and the Fernet system for symmetric encryption and decryption are the most vital. In what remains of this section, a paragraph is devoted to each of these extensions, and the Fernet system as well.

Flask-Login provides session management, 'remembering' the current state of the user interacting with the application. In other words, if a user provides credentials matching that of an individual present in Healthcrypt's records, Flask-Login will mark the user as 'authenticated' and begin playing a larger role in his or her session. The most notable benefits come in the form of user conveniences. For example, if a user successfully logs in but navigates away from his or her homepage, Flask-Login functions can be used to automatically redirect the user back to his or her homepage. This user convenience turns out to be a convenience for the developer as well. Instead of writing complex conditionals to track the state of various users, Flask-Login provides simple validation checks that can be placed at the top of the execution of each routing function. As you might imagine, Flask-Login offers checks verifying the anonymity of a user as well. If an anonymous user attempts to access a page without logging in, their status, which is tracked by Flask-Login, informs the routing function that has been activated and navigates the user back to the login screen. The utility of Flask-Login's easy-to-integrate helper functions benefits developers in the same way the simple-by-design core of Flask itself does. It frees the developer to focus on the most critical aspects of an application.

Healthcrypt's interactive forms, which are used for registration, login, and creating and editing records, are powered by Flask-WTF, an extension that allows us to define various form fields, fine-tune their behavior according to validation rules, and render them inside HTML templates. These templates are displayed when certain routing functions are called. In other words, when a user accesses a page containing a form, it is a Flask-WTF form embedded in the page that they are interacting with. Like all the other extensions and libraries mentioned in this section, Flask-Login significantly alleviates the burden of development, allowing critical aspects of an application to remain at the center of attention.

Consistent with its bare-bones, essentials-only philosophy, Flask does not directly support any particular database. Instead, database integration is put in the hands of the developer, with many helpful integration extensions to choose from, and most of them offering simple, convenient constructs to get your application up and running. Healthcrypt uses Flask-SQLAlchemy, an extension that simplifies development "by providing useful defaults and extra helpers that make it easier to accomplish common tasks" [7]. Flask-SQLAlchemy is a wrapper to the SQLAlchemy package, an Object Relational Mapper (ORM) that allows us to take an object-oriented approach to designing and interacting with our database. In other words, database entities can be built and interacted with using the classes, objects, and methods familiar to object-oriented developers. In keeping with the package pattern's emphasis on modular design, database tables - which are written like traditional, object-oriented classes - are typically written in a separate "models.py" file and imported during the application's start-up. Routing functions are treated the same way, placed in a separate file and imported during start-up. Once the models are written and properly linked to the relevant files, the developer can use Flask-SQLAlchemy to execute database commands and inform routing decisions - for example, accessing information to drive conditional statements, or manipulating table records after a form has been validated and submitted. To conclude, Flask-SQLAlchemy significantly streamlines application development by providing simple constructs for database integration, thereby serving as another excellent example of how Flask's modular-by-nature design steers development towards the most critical components of an application.

The Fernet encryption and decryption system is part of Cryptography, a Python package that provides "both high level recipes and low level interfaces to common cryptographic algorithms such as symmetric ciphers, message digests, and key derivation functions" [3]. Healthcrypt uses Fernet to generate keys, encrypt physician and patient information for storage, and decrypt physician and patient information for display. More specifically, when a physician creates an account and registers with Healthcrypt, Fernet is used to generate a unique key for the physician, which is stored along with the rest of his or her information, and the key and any other relevant information - either encrypted or not - are accessed when necessary and passed as parameters to Fernet's built-in encryption and decryption functions. Before we continue discussing the details of Healthcrypt's encryption and decryption strategy, it is worth mentioning the underlying cryptographic primitives used by Fernet. According to its documentation, Fernet uses AES in CBC mode with a 128-bit key for encryption, with PKCS7 padding; HMAC with SHA256 for authentication; and os.urandom(n), a method used to generate a string of n random bytes, for generating initialization vectors [2]. Continuing with Healthcrypt's encryption and decryption strategy, it is important to note the data types used by Fernet in comparison to the data types used in Healthcrypt's models and forms. The key is stored as a "LargeBinary" data type, one of many generic types supported by SQLAlchemy, and the Physician's first name and last name are encrypted with this same key and stored as "LargeBinary" data types as well. The "LargeBinary" data type was used because Fernet accepts and returns bytes - that is, byte literals with the prefix 'b' - for its encryption and decryption functions. As you may have guessed, all the details of a patient's record - first name, last name, date of birth, and diagnosis - are encrypted and stored as "LargeBinary" data types as well. This storage strategy is certainly more secure than unencrypted plaintext storage; however, it demands a specific approach to retrieving and display the data, and a specific approach to storing and restoring the data. Any time data are displayed in a form, or in the welcome message of a physician's homepage, it needs to be not only decrypted but also decoded, leaving nothing but the original plaintext string. For the patient's date of birth, an additional step is required. The plaintext date of birth string must be converted into a "datetime" object with the same format specified in the "PatientRecord" form of the "forms.py" file. Of course, when the patient's date of birth, updated or not, is resubmitted into Healthcrypt's database, it must be converted into a string, encoded into bytes, and encrypted with the physician's key. No other piece of information requires a datetime conversion, but they all require some combination of encoding and decoding prior to encryption, decryption, and display. Constant conversions may seem tedious, but they allow Healthcrypt to function as intended, providing a clean, intuitive display, and securely storing patient and physician information in the background. Like the extensions previously mentioned in this section, the Fernet encryption and decryption system greatly alleviates the burden of development, steering development towards the most critical components of an application and providing developers greater peace of mind when it comes to meeting functionality and security requirements.

Experiment

After launching Healthcrypt locally and creating an account with one or more patient records, a simple test can be conducted through the command line to ensure patient information is encrypted in storage. With Healthcrypt still up and running, activate the virtual environment in a second command line utility window and execute the command "flask shell." This will start an interactive Python shell with knowledge of Healthcrypt and its database, which are imported and defined in the "Healthcrypt.py" file. Once the Python shell is running, execute the command "print(Record.query.all())" to view all encrypted records present in the database. There is a definition in "models.py" that puts the records in a nice, digestible format when queried and displayed; however, as you will notice, the information is encrypted, preventing anyone with this level of access from actually determining the contents of the patient records. This simple test is useful for developing and debugging, as it provides a means to ensure patient information is encrypted in storage.

Discussion and Conclusion

Healthcrypt provides a rudimentary exploration of encryption within the context of healthcare, and as you may have guessed, throughout the development of Healthcrypt it became apparent that there is much more to these systems than meets the eye. Encrypting information prior to storage is difficult to manage on its own, but most legitimate storage systems for electronic medical records incorporate many more features than Healthcrypt, often requiring additional security measurements for each one as well. In other words, Healthcrypt's use of encryption prior to storage and decryption prior to display is the tip of the iceberg when it comes to security in the context of healthcare. For example, consider the possibility of adding a feature that allows physicians to share patient records. This would complicate the system by requiring some mechanism to manage the exchange of and/or access to the keys used to encrypt and decrypt patient information. Furthermore, assuming the physicians work in different locations, and the system is accessible remotely, securing the information in transit becomes a necessity as well. Features like these are arguably essential to maintain an efficient flow of information and effectively treat patients. The difficulties present in developing a system as simple as Healthcrypt and the reality of what's required to effectively circulate patient information in order to treat individuals properly showcases the necessity of understanding critical, cryptographic mechanisms in the context of healthcare. There is much to know and much to double-check when building these systems.

References

[1] Jim Atherton. *Development of the Electronic Health Record*, 2011.

[2] The Cryptography Developers. Fernet (symmetric encryption). `https://cryptography.io/en/latest/fernet/`, 2017.

[3] The Cryptography Developers. Cryptography. `https://pypi.org/project/cryptography/`, 2020.

[4] Office for Civil Rights. Hipaa for professionals. `https://www.hhs.gov/hipaa/for-professionals/index.html`, 2017.

[5] Ponemon Institue. 2020 global encryption trends study. `https://www.ncipher.com/2020/global-encryption-trends-study`, 2020.

[6] NIST Information Technology Laboratory. *2017 ANNUAL REPORT*, 2017.

[7] The Pallets Projects. Flask-sqlalchemy. `https://flask-sqlalchemy.palletsprojects.com/en/2.x/`, 2010.

[8] The Pallets Projects. Foreword: What does "micro" mean? `https://flask.palletsprojects.com/en/1.1.x/foreword/`, 2010.