Security Algorithms Final Writeup

Jarett Sutula

Due May 15th, 2022

Marist College

Spring 2022

MSCS 630L: Security Algorithms

Professor: Aaron Kippens

JSPM: AN AES ENCRYPTED PASSWORD MANAGER

JARETT SUTULA

Jarett.Sutula1@marist.edu

1 Abstract

While user password security has increased over time, users now maintain larger and larger amounts of online accounts than ever before. In order to ensure the compromise of one account does not include others, user password protocols typically advocate for the use of distinctly different passwords for different accounts. While this helps isolate credentials from each other, it also makes credentials harder for users to remember. Password managers exist to bridge this gap by storing a user's credentials for them in a secure and encrypted way where they can access them from any device while still maintaining strong, secure passwords without forgetting them.

2 Introduction

With an ever-growing presence of data on the internet, it is important for one to protect their identity and ensure the security of their information. With virtually every online service requiring an account, credentials have become the target of hackers and data vendors alike. Password managers were built to help users safely store and remember their important credentials without having to store them physically offline. This convenience puts our credentials in the hands of third-party companies and many have already been attacked and had their data leaked into the hands of others. Password manager leaks are not uncommon, and there are many instances where passwords are being saved in databases with outdated encryption methods or simply in plaintext by large companies.

This paper serves as a further look into creating a secure password manager with various levels of encryption that can prove to be trustworthy to users on the web or offline. This includes encryption of credentials before they are placed into a database as well as additional hashing and salting of the encrypted credentials after they are placed into the database to ensure the greatest security.

Jarett Sutula Password Manager (JSPM) is a python web application built to provide users with

a seamless, trustworthy experience with storing their website credentials online. It will also give them access to their passwords regardless of what device they are on as long as they can connect to the web server. While JSPM runs on a local web server for users to work, it could easily be lifted onto a domain and provide access for users without needing to download and run the web server themselves.

3 Background

A further look into password managers shows many different approaches for credential encryption. Previously, password managers used algorithms like MD5 and SHA1 for encryption but these algorithms have not withstood the test of time. MD5 is no longer safe, SHA1 has had collisions, and both were deprecated by the large tech giant Google and many companies followed suit. AES is still a government standard for encryption and therefore enjoys usage by password managers from both browser-based and standalone apps, although they typically are using at least 128-bit encryption. This could surely be expanded to 192+ or even a consistent 256-bit encryption to improve quality. Software companies also have taken Google's recommendations and many use SHA-256 and the newer SHA-3 hashing algorithms to further secure their credentials.

Using a combination of high-bit AES encryption with a combination of the popular python hashing and salting library bcrypt should prove a strong encryption process that guarantees the data's security not only when the user enters them, but also as they stored in the database with an extra layer of hashing/salting. This means, unlike some of the most infamous data breaches, credentials will not be sitting in low-protection databases without that extra layer of security. This app will also offer offline support for those who do not want their data being stored anywhere online but will come at the added cost of not being able to access their passwords from every device and will prove to be vulnerable to device destruction or loss of physical access to the offline app.

4 Methodology

Our lab code was in Java, but I wanted to build an online web server I could spin up and connect to a live database. JSPM uses python for AES encryption and decryption instead of Java. Once in python, I had access to a familiar web app server called Django. Django was used to spin up a server in which modules and views written in python would help a user create an account, add credentials to their favorite websites into a form, and have that form stored online in a database. I used the python module pymongo to connect Django's models and views to a MongoDB database.

Since a credential list in python would consist of a website, the user's username for that website, and the corresponding password for that website, it was essential that these credential lists were fully encrypted and not stored in plaintext anywhere. It was also imperative that the credential lists in hex strings (before AES encryption) were also not stored anywhere, as these could be reverse engineered if someone knows PKCS5 padding and could apply it to the plaintext in hex.

The safest way to do this was to use something hashed and salted as the secret key and entrust it to the user. In JSPM, the user's password they create for their JSPM account is hashed and salted by the python module bcrypt. Bcrypt's generation of the salt means that saving the same password "1234" with two different bcrypt salts means that they can have different hashed password values but still match each other in bcrypt.checkpw() (R. A. Karthika, 2015). This makes the secret key, which in AES needs to be the same for encryption and decryption, incredibly important to secure. Accessing the password value in MongoDB gives a binary number that is not in any way discernable, which means that as long as the user knows their password they are realistically the only person who can decrypt the data. There is no "decrypt" function of bcrypt which means even if someone had access to the code they wouldn't be able to just grab the hashed password from MongoDB and get the plaintext value or hex value from it.

With this in hand, it means that JSPM can now keep encryption safe and only needs to store the fully-encrypted website credentials in MongoDB. From here, I built AES decryption in python which would require the user to log in to verify their password. By verifying their password, we know the secret key will work on the decryption key and therefore can send out the decrypted website credentials only on the page and not stored anywhere else.

JSPM was built using Django's hierarchy, meaning that the folder "aes_sutula/polls" contains the majority of the web server's functionality. CSS and static images are stored in the "static" folder, the "templates" folder. The file "urls.py" routes the html pages between each other and each URL pulls a view from "views.py" that populates the page with html and forms. Each form is a model from "models.py". Actual AES encryption and decryption work is from "aes.py" and the use of "constants.py" and called when encrypting or decrypting.

5 Experiments

The biggest struggle originally was the padding of plaintext for AES encryption and decryption. The lab code we worked on gave us a set 16-byte plaintext string and a 16-byte key string, meaning that they required no padding to meet AES's strict size requirements. In my use case for JSPM, however, users may be saving a website between "apple.com" and "thebananarevolution.com", which means that JSPM would need to pad whatever leftover bytes were left. The code available in /prj/writeup/code/aes.py has a function called plaintext_to_hex which, when given a plaintext string like "apple.com", figures out how many 16-byte strings it will need to generate. For anything 16 characters and under, 1 string works - but long credentials like thebananarevolution.com would take 2 strings since it is between 16 characters and 32 characters.

There were a few different padding possibilities for my project. I decided upon using the PKCS5 padding scheme and tweak it for AES's 16 byte encryption. PKCS5 is one of the more popular AES encryption padding schemes and allows plaintext to be ASCII or even binary (Adla Sanober, 2022). While binary does not apply itself to the scope of this project, it is a relatively simple scheme that would potentially offer functionality down the road if JSPM ever grows to be something larger.

6 Discussion

While playing around with AES and coding decryption, it did become obvious why AES is not the most popular password manager encryption method. The secret key both encrypts and decrypts the plaintext, making AES fundamentally symmetrical and prone to vulnerabilities. If someone gains access to the key and has encrypted data, they can decrypt it. AES is also strict with its input, as it requires exactly a 16-byte key and plaintext to encrypt correctly. While this can be solved with thing such as the PKCS5 padding JSPM integrates, it still means that AES may require more effort into taking in input and pushing out output than another algorithm of choice.

While the most common algorithms for password managers typically used a hashing encryption algorithm, JSPM takes a non-hashing encryption algorithm in AES and uses bcrypt to hash the secret key. We can effectively mimic hashing capabilities in AES, which proved useful as I already understood how AES worked and how to manipulate keys and plaintext to be used by it. After playing with tons of test pushes, MongoDB only ever stored the user's password and their set of website credentials. The user's password was hashed and is no longer recognizable from the plaintext the user enters. The user's set of website credentials are encrypted using AES and the key (when it is not hashed), meaning that the encrypted data and the secret key in plaintext are never stored at the same time in JSPM - in fact, the secret key is never stored in anything but its hash, meaning there isn't a way for someone with access to the database to get the secret key.

7 Conclusion

Despite my original findings of password managers, AES was not too difficult to work with to get a secure, working credential encryption. While it may not be optimal as it doesn't come with a hash, I did find there are plenty of ways to implement hashing and salting into AES that proved useful. We know that AES-128 is still secure and used by the US government, so we can be sure of the reliability of AES encryption - the added hashing just makes it more secure. Together, I'm confident that even if the MongoDB database got compromised, the culprit wouldn't even be able to figure out what website one of the credentials is for. It also put into light how unjustifiable it was for large companies like Yahoo! and Facebook to be storing passwords in plaintext within the last decade, and I am confident JSPM shows just how simple and effective encryption can be to the security of our data.

References

Adla Sanober, Shamama Anwar. 2022. “Crytographical primitive for blockchain: a secure random DNA encoded key generation technique.” *Multimedia Tools and Applications* .

URL: *https://doi.org/10.1007/s11042-022-13063-z*

R. A. Karthika, P. Sriramya. 2015. “Providing password security by salted passwords hashing using bcrypt algorithm.” *ARPN Journal of Engineering and Applied Sciences* 10(13).