Introduction to IT Security

Homework 4 – Authentication and Authorisation

Abraham Murciano

September 13, 2020

1 Hashing

Firstly, we are to build a function which receives a string and returns a hex encoded SHA256 hash. Here is said function.

sha256_hash.py

```
import hashlib

def sha256_hash(plaintext: str):
        # convert the plaintext into bytes for the hash function
        bytes_plaintext = plaintext.encode()

        # convert the plaintext bytes into hashed bytes
        hashed_bytes = hashlib.sha256(bytes_plaintext)

        # convert hashed bytes into base 16
        return hashed_bytes.hexdigest()

if __name__ == "__main__":
        print(sha256_hash(input("Enter a string to hash:\n")))
```

Next, we are tasked with building a function which will receive login credentials and store them securely in a database. Below is some Python code which will achieve this.

add_user.py

```
import sha256_hash
import sqlite3
import random
import string
```

```
def add_user(username: str, password: str, db_conn):
        salt = rand_string(255)
        hashed_password = sha256_hash.sha256_hash(password + salt)
        db_conn.cursor().execute("""
                insert into users
                        (username, hash, salt)
                values
                        (?, ?, ?);
        """, (username, hashed_password, salt)).close()
        db_conn.commit()


def rand_string(length: int, chars: str = string.ascii_letters + string.digits):
        return "".join(random.choice(chars) for i in range(length))


if __name__ == "__main__":
        conn = sqlite3.connect("users.db")
        conn.cursor().execute("""
                create table if not exists users (
                        username varchar(255) primary key,
                        hash char(64),
                        salt char(255)
                );
        """).close()
        conn.commit()
        username = input("Username: ")
        password = input("Password: ")
        add_user(username, password, conn)
        conn.close()
```

If an attacker would try to crack the hash using brute force, it will take an extremely long time, but calculating an average would be impossible without knowing what kind of processing power the attacker has at their disposal or the length and character set of the password whose hash they are attempting to find.

Other attack vectors one may consider would be to have a database with a list of all possible hashes along with strings that generate each hash. Creating such a resource would take up a ridiculous amount of space and time, but once created it would take a matter of seconds to find a string which hashes to a particular hash. The amount of space such a table would occupy would be 64 bytes (32 for the hash, and about 32 for the string) multiplied by the number of different SHA256 hashes, which is 2^{256}. This equals approximately seven unvigintillion four hundred ten vigintillion terabytes, which is a completely infeasible amount of storage.

Rainbow tables are a method which aids in greatly reducing the amount of storage that a database would take up, at the cost of computation time. However, neither of these methods would work to crack the password whose

hash we stored in the database, since the hash is stored along with a salt which is a randomly generated 255-character long string, which is different for each user. This means that if a rainbow table or database would be used to try to reverse lookup the hash, said rainbow table or database would have to be specially crafted for each individual user, which would prevent attackers from using pre-generated rainbow tables or databases.

2 Public Key Protocols

We are tasked with describing a protocol we can use to communicate securely between a client and a server over a public connection without using CA certificates and without keeping any of the client's source code private.

The protocol we will describe is the Diffie-Hellman key exchange. We start off by choosing two numbers, which are sent publicly between the client and the server. These we will call g and p. Both of these are prime, and p is very large (on the order of four thousand bits).

Next, the client chooses a secret number c_s at random between 1 and p, which it keeps private, and the server chooses a similar number which we shall label s_s. The client then calculates $g^{c_s} \bmod p$ and the server calculates $g^{s_s} \bmod p$. We will label these resulting numbers c_p and s_p respectively, since these are shared publicly.

Once c_p and s_p are exchanged between the client and the server, the client now calculates the secret key $k = {s_p}^{c_s} \bmod p$, and the server calculates the exact same secret key $k = {c_p}^{s_s} \bmod p$. The mathematical reason why the key they each generate is beyond the scope of this paper, but the important part is that both c_s and s_s are kept private and are randomly generated, and without these it is extremely difficult for an attacker to guess the secret key k without these.

Once we have a shared secret key, the client and server can use a symmetric encryption algorithm such as AES to encrypt their messages.

However, the problem with Diffie-Hellman is that it is vulnerable to a man in the middle attack. One solution we can use to avoid this is by using some asymmetric cryptography protocol. The server can have a public and private key. The client does not need a private key, since if it had one we are given that any installation files may be stolen. The client does however store the public key of the server in its installation files.

This way, the client can be sure that any $g^{s_s} \bmod p$ which can be decrypted with the server's public key was in fact sent from the server. Even though the client does not have a pair of keys, and messages from the client can be intercepted and changed by an attacker, the attacker would not be able to cause the client to think that they are actually the server. So once they establish the Diffie-Hellman exchange, the attacker would be left harmless.