Csc380 Computer Security SPR 2025 Secure(?) chat swap analysis
Christopher Barbosa

Introduction/Project description:
The goal of the chat project was to put what we had learned so far on cryptographic protocols such as:

1. Diffie-Hellman
2. Digital Signatures
3. Message Authentication
4. SSH, etc.

in order to develop a working chat program in c which provides security to both parties utilizing the protocols aforementioned. We were tasked with implementing authentication and integrity between parties(MACs/HMAC), message secrecy through encryption and decryption(using AES, etc.), in particular the following was explicitly asked of us:

```
I've given you a skeleton which does very basic chat stuff: Depending on the invocation,
it will listen for connections, or make one with another host. Beyond that, it just sends
and receives text,displaying each message in a log window. It will be up to you to:

- Write some sort of handshake protocol to setup ephemeral keys
(your protocol should have perfect forward secrecy!).
- Mutual authentication, using public key cryptography.
- After authentication, each message should be encrypted and tagged with a message
authentication code. You may also want to take measures to prevent replay attacks.


I think SSH will be a good model on which to base your protocol. In particular,
don't use PKI (public-key infrastructure,with certificates and such), and instead assume
that communicating parties have already exchanged public keys. However, implementing
deniable authentication would be a nice touch (and is something SSH does not provide).
If you want to use 3DH, you can find an example in dh-example.c.
```

The first part of the project was implementing the above on our own. This report outlines my analysis from the second part of the project, which was the scrutinize the project implementation of another group. Therefore, the remaining of this project will aim to achieve the following:

a. analyze logic of the cryptographic protocols implemented
b. test groups claims on security of the system
c. look for and identify any glaring software security issues present

1 (protocols implemented).

Solution.
(Since the main importance of this analysis is to analyze the security claims the group makes, I will keep this section fairly short and concise) The bulk of the cryptography used for this project can be broken down into 3 main features:

1. **handshake/key exchange and authentication**

2. **message encryption and decryption**
3. **MAC and message integrity parameters**

In the report they explicitly outlined how each of these features were implemented. Specifically, they wrote the following:

> Server and client generate public and private RSA keys before establishing a connection. The keys are already stored in the repository's keys directory. If the keys are not present, use generate_rsa_keys in util.c to create them. Server and client generate a Diffie-Hellman key pair, using the functions in dh.c. Server and client each sign their public Diffie-Hellman keys with their private RSA keys, and send both the DH public key and this signature over the channel. Both parties verify the received signature using the other party's public RSA key, ensuring that it matches the Diffie-Hellman public key. The Diffie-Hellman shared secret, generated by each party from their own secret keys and the received public keys is used as the key for HMAC. Every message is encrypted with AES and then sent along with its corresponding HMAC. If the HMAC is verified to match the AES encrypted text, the text is decrypted and the message displayed to the recipient.

To implement the key exchange they used privacy-enhanced mail(PEM) formatted files. PEM-file were specifically created in order to send keys, certificates, and other cryptographic data as securely as possible. The protocol has built in security features that provides reassurance to both the receiver and the sender, guaranteeing the data was not tampered with through transportation; doing so by comparing the data before sending it out and afterwards, properly mitigating man in the middle attacks. An aside is that the only way for the chat program to run correctly is for me to generate the RSA keys beforehand and have the stored in the PEM files already. This is assumed to be done as securely as possible. Example of usage(server-side):

```c
// read in appropiate RSA keys
    FILE *fp = fopen("keys/server/private.pem", "rb");
    EVP_PKEY *rsa_sk_server = PEM_read_PrivateKey(fp, NULL, NULL, NULL);
    fclose(fp);

    printf("\nServer successfully read server private RSA key.\n");

    fp = fopen("keys/client/public.pem", "rb");
    EVP_PKEY *rsa_pk_client = PEM_read_PUBKEY(fp, NULL, NULL, NULL);
    fclose(fp);

    printf("Server successfully read client public RSA key.\n");
```

Looking into it, it seems like you could add a password onto the PEM files for further security, which they didn't do, I'm not too sure how important this is though. Most of the protocols were implemented very well with proper security parameters and considerations in place, however I was able to identify some problems with their implementation. These issues will be addressed in the upcoming sections. □

2 (**Groups security claims**).

Solution.
The first security claim they made was:

> **Confidentiality**: Messages are encrypted using AES-256-CBC with keys derived from a DH shared secret, ensuring secrecy against passive eavesdroppers.

From what I was able to see in the code, encryption and decryption were implemented correctly, they used a random IV for every message and followed the paradigm of encrypt then MAC.

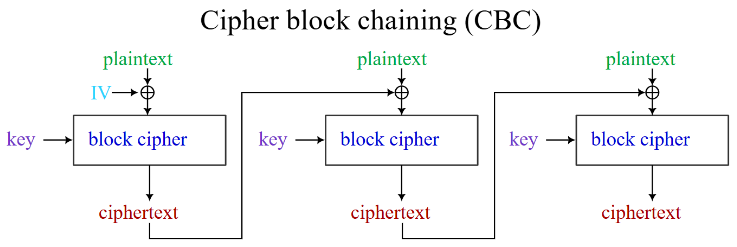

Figure 1: CBC mode enc

The reason for:

```
MAC(ENC(m))
```

is based on how CBC mode works with the block cipher. It breaks down the plaintext into multiple of 16 byte blocks and if it can't do so evenly then it attaches some uniform padding to the end of each block; meaning if they add 3 bytes then each block will end in 3 bytes of the number 3, etc. if we MAC first then an attacker has a chance to figure out the MAC utilizing a paddle oracle attack that is possible for CBC mode encryption. Based on the timing they can tell whether it's a padding error or they wrote into the MAC.

I did not see an explicit check to make sure that the two shared keys generated are the same, which could lead to problems(small subgroup confinement, serialization mismatches, etc.). I think that them not doing this is actually mostly fine because of the utilization of digital signatures. Each side(server/client), sign there public keys before sending them over, and both sides check to make sure that these signatures are correct before generated the shared secret, which leaves an attacker virtually no wiggle room to modify the public keys in transport(unless they corrupt the memory directly). The implementation is as follows:

```
    // sign DH public key
    size_t sig_len = EVP_PKEY_size(rsa_sk_server);
    unsigned char *signature = OPENSSL_malloc(sig_len);

    generate_signature(rsa_sk_server, dh_pk_server,
        &signature, &sig_len, fds);

    printf("Server successfully generated signature of DH public key.\n");
    size_t dh_pk_server_len = serialize_mpz(fds[1], dh_pk_server);
```

The above signs and send the public key, and the next bit of code verifies the received signature and key:

```
    unsigned char recv_buf[2048];
    if (recv(sockfd, recv_buf, 2048, 0) == -1) {
        error("ERROR receiving signature from client.\n");
    }

    printf("\nServer successfully received signature!\n");

    mpz_t dh_pk_client;
    mpz_init(dh_pk_client);

```

```c
    unsigned char* signature_client = NULL;
    size_t sig_len_client;

    extract_signature(recv_buf, dh_pk_client, &signature_client, &
       sig_len_client, fds);
    int verify_ok = verify_signature(rsa_pk_client, dh_pk_client,
       signature_client, sig_len_client, fds);
    if (verify_ok == 1) {
        printf("Server successfully verified client signature!\n");
    }
```

Lastly, an argument can be made for the following:

```c
    dhFinal(dh_sk_server, dh_pk_server, dh_pk_client, dh_shared_key, 256);
```

instead of directly computing the final key for the encryption using the keys, they could have used a KDF which would have increased the security. Also, since the decrypt was implemented to simply return -1 with any error, an attacker could conduct a padding oracle attack. For example, an attacker could capture a ciphertext and modify the last bits, if the decrypt returns immediately its a padding error, else the padding is valid so we just read into the text bits, which could cause issues.
The second security claim they made was:

> **Integrity**: Each message includes a SHA-256-based HMAC for tamper detection.

The usage of HMAC was correct, however there is an issue with the following in the verify_hmac(...) function in the util.c file:

```c
        unsigned int actual_length;
        unsigned char* actual_hmac = generate_hmac(key, key_length, (const
           unsigned char*)msg, msg_length, &actual_length);

        if (!actual_hmac) {
            printf("generate_hmac() returned NULL!\n");
            return -1;
        }
        // length check
        if (actual_length != hmac_length) {
            return 0; // invalid because of length mismatch
        }
        // compare HMACs using constant time comparison
        if (memcmp(actual_hmac, expected_hmac, hmac_length) == 0) {
            return 1; // match
        } else {
            return 0; // mismatch
        }
```

they say explicitly that they compare the HMAC values in constant time but this is not true for the standard c memcmp in general. Therefore, the provided code is susceptible to timing attacks. For example, typically memcmp works by testing byte by byte to see if they match and returns once it finds a difference, thus an attacker could use this to check how close to the actual HMAC there guess is, and gradually learned the HMAC.
The third security claim they made was:

> **Mutual Authentication**: Both client and server verify each other's DH keys via RSA signatures to prevent man-in-the-middle attacks.

The implementation of the signatures seemed fine to me, I couldn't find anything glaringly wrong with the code, and most of my thought on the signature usage was talked about under the first claim. □

3 (**Additional security vulnerabilities and possible attack vectors**).

Solution.
They acknowledged themselves in the report that they did not take any precautions against replay attacks, since no type of nonce or timestamp is used. Additionally, the keys for encryption and decryption are generated per session therefore there might be issues for long sessions where a lot of messages are sent for an attacker to play around with. It such cases it is recommended to rekey every x amount of messages so that an attacker has less time to try and figure the keys out. Also since no bounds checking is done on the length of the messages we could do a possible stack smashing attack(assuming all system level mitigation's are turned off). For example, the following function:

```c
int extract_hmac(size_t* len, unsigned char* message, size_t* hmac_len,
   unsigned char* hmac, unsigned char *iv, unsigned char* hmac_buf) {
    memcpy(len, hmac_buf, sizeof(size_t));
    memcpy(message, hmac_buf + sizeof(size_t), *len);
    memcpy(hmac_len, hmac_buf + sizeof(size_t) + *len, sizeof(size_t));
    memcpy(hmac, hmac_buf + sizeof(size_t) + *len + sizeof(size_t), *hmac_len)
       ;
    memcpy(iv, hmac_buf + 2 * sizeof(size_t) + *len + *hmac_len, 16);

    return 0;
}
```

The value of *len is not regulated so when it is called in chat.c in the recv_message() function:

```c
        size_t ciphertext_len;
        size_t hmac_len;
        unsigned char hmac[256];
        unsigned char iv[16];
        extract_hmac(&ciphertext_len, ciphertext_buf, &hmac_len, hmac, iv,
           hmac_buf);
```

We don't check to make sure that nothing malicious is in the ciphertext which could cause issues. So an attacker could overflow the buffer, spawn a shell, and since at this time dh_shared_key is in memory they could dump that and learn the shared key. Or they could alter the return address to execute malicious code which turns off certain security parameters making the messages vulnerable, etc. □