CSCI4230 Blackhat Analysis of Ian Conrad and Frank Johnson's Project

Aidan McHugh, Kai Orita, and Bradley Presier

April 24th 2023

Contents

1 Alterations

For various reasons, we had to make a few minor alterations to the provided code. For the most part, these did not change the behavior of the project. We did make major alternations to the prime number-finder, which we elaborate on later. It will be explicit where we change behavior.

1.1 Multiprocessing Issues

The program did not immediately function on my machine due to interaction between Python multiprocessing's forking and TCP sockets. However, this issue was quickly resolved by adding

```
if __name__ == "__main__":
```

to the main files of both the server (`SERVER_main.py`) and client (`CLIENT_main.py`).

1.2 Address/Port Changes

In order to simplify intercepting encrypted messages, we changed the address that the sockets would connect on. The same could also be achieved with packet sniffing tools, but for ease of use we created a relay script through which packets pass. This does not provide any view of the internal state except what is already sent out into the internet.

1.3 Prime Number Finding Optimization

Copied prime number-finding code from our project, resulting in a speedup 700x. This provides no alteration to the behavior, but allowed us much faster load times. This additionally allowed us to the `multiprocessing` module, making the Multiprocessing Issues changes redundant.

2 Message Organization

By intercepting packets and looking at the source code, we analyzed the format of the transmitted messages. First, we found that all messages are in a request-response pattern, initiated by the client.

2.1 HELLO messages

Presumably based on TLS 1.3, the first request and response are of type `HELLO` (id 0). They do not appear to be used after this. They lack a MAC or Signature and are of the following format:

2.1.1 Request

A string: *version* | *clientNonce* | *sessionID* | *cipherSuite* | *compression*

- `version`: Always `1.3` (as in TLS 1.3)
- `clientNonce`: A random 64-bit prime number in decimal format
- `sessionID`: A random 64-bit prime number in decimal format
- `cipherSuite`: Always `RC5:SHA-1:stream:F:20`
- `compression`: Always `SHA-1`

2.1.2 Response

A string: *version* | *clientNonce* | *sessionID* | *cipherSuite* | *compression* | *certificate*

- `version`: Always `1.3` (as in TLS 1.3 and same as request)
- `serverNonce`: A random 64-bit prime number in decimal format
- `sessionID`: Same as request
- `cipherSuite`: Always `RC5:SHA-1:stream:F:20` (Same as request)
- `compression`: Always `SHA-1` (Same as request)
- `certificate`: A string `Certificate:`N`:`e where N, e are the server's public key, in decimal format (same for all users).

2.1.3 Potential vulnerabilities

Prime Nonces
Since both parties only use prime nonces, the number of possible nonces is significantly decreased. This may make guessing or precomputing nonces easier, and also makes the possibility of nonce reuse much higher.

Prime sessionIDs
This is much less significant than the prime nonces, but may still provide opportunity if the smaller id space causes reuse.

Lack of Certificate Authority
No verification information is included here to allow the client to verify that the keys actually belong to the server. As ATMs owned by or affiliated with the bank they are connecting to, information could be hard-coded onto the ATM. A simple option would be to just hold the server's permanent public keys. As this server regenerates its RSA keys every time it starts (which may be more realistic than a single set of RSA keys for all bank servers), it could still have those public keys signed by a mock CA that is trusted by the client. We use this for our Man-in-the-Middle attack.

2.2 KEYGEN messages

These messages seem to be used to communicate for Diffie-Hellman key generation. They come directly after the HELLO messages and do not seem to be used more than once. They allow both parties to generate a session key based on the `premasterSecret`, `serverNonce`, and `clientNonce`. They lack a MAC or Signature and are of the following format:

2.2.1 Request

A string: *`rsaPow`*|*`rsaCipher`*
These values are generated using the server's public key e, N and the client's `premasterSecret`, a 128-bit integer.

- `rsaPow`: A random integer less than N, encrypted using the server's public key. Represented as a decimal.
- `rsaCipher`: The 128-bit `premasterSecret` XORed with the 160-bit hash of the random integer from `rsaPow`. Represented as a decimal.

2.2.2 Response

A string: *`paillier_n`*|*`paillier_g`*

- `paillier_n`: The generated value n for the Paillier cryptosystem.
- `paillier_g`: The random value g for the Paillier cryptosystem.

2.2.3 Potential Vulnerabilities

Partial Hash Exposure with SHA-1
In the request's `rsaCipher`, the 32 most significant bits of the random integer are directly exposed. While this is not normally an issue, the hash function is SHA-1 which is considered broken. Additionally, while we assume this would not be the case in practice, the random integer is smaller than the hash length (We cannot assume that the `premasterSecret` would be larger in practice, as then the hash would not be able to mask it). While we do not have the computational power to do so, it may be possible to narrow down the set of possible values of the random number between the hash and its RSA encryption.

Malleable premasterSecret
Due to being XORed, any of the 128 least significant bits of the request's `rsaCipher` may be flipped to flip the corresponding bit of the `premasterSecret` seen by the server.

2.3 ENCRYPTED message

This has message type (id 2) contains all the Protected Data Models as defined by their documentation. They always include a MAC, and requests (never responses) will include a Signature after the user provides RSA keys (should be all but `VERIFY` and sometimes `FREEZE`).

2.4 ENCRYPTED message/VERIFY

This has the protection operation id 3 and is used to send the username and password for authentication.

2.4.1 Potential Vulnerabilities

Unbounded Content Length
The client takes the username and password as-is, conatenates with a `/` character between, then encrypts and sends. However, the maximum supported length of a packet here is 4096 bytes, and the storage of all encrypted data as a string of `1`s and `0`s effectively causes an expansion factor of at least 8 (testing shows it may be slightly more). The user data is encrypted again within the main encrypted data, resulting in this part having a net expansion factor of at least 64.

Effectively, this provides (by testing) about 47 characters to be split between the username and password before the message becomes too long, causing json decoding to fail. Unfortunately, we were unable to use this to extract or alter any information due to it causing a decoding error upon receipt.

2.5 ENCRYPTED message/FREEZE

A string: *`username`*
Freezes the account of whatever `username` is provided. We use this for an attack.

3 Cryptographic Primitives Analysis

Here we will analyze the implementation of cryptography and compare them to their general form.

3.1 Hash Function

They chose SHA-1 to implement, which is a vulnerable hashing algorithm.

3.2 HMAC

They implemented HMAC. While they use the aforementioned SHA-1, we will not repeat the mentioned vulnerabilities here.

3.3 Symmetric Cipher

They chose RC5 to implement. The implementation seems to generalize the parameters, but they specify that they use 20 rounds and 16-bit words.

3.4 Asymmetric Cipher

They chose RSA to implement. They use 64-bit primes for p, q. While we might typically assume that larger values would be used in practice and this choice was merely for the sake of speed in this sample system, larger primes would still leave an issue: each user needs to retain their own RSA parameters (N, p, q, e, d), and the server must retain (N, e) for each user. While perhaps more secure (or perhaps not), full-sized personal RSA parameters would cause issues when attempting to log into an ATM using a debit card, as they would either have to be stored in the card or entered by the user (both non-standard behavior that limits compatibility).

Nevertheless, we will work under this system and assume a brute force attack to find secret keys would be intractable.

They expanded the encryption algorithm to be nondeterministic, by XORing the message with a hash of a random number, then encrypt that random number with RSA and attaching it. However, this makes the message malleable via XOR and limits message length to the hash length (160 bits, though their implimentation further limits it to 128 bits due to RSA key length).

3.5 Asymmetric Digital Signature

They implemented an RSA-based digital signature scheme. As detailed below$_{\downarrow}$, we can easily forge signatures under this scheme. We specify two methods for forging signatures, which require either

- Having the public key and seeing a single valid signature (without message), or
- Seeing a single valid signature with its corresponding message (without public key)

Since we typically get the message using the public key, the first option is the one we implement in our attacks. Both can be done in constant time, so do not require any brute force.

4 Negative Money Vulnerability

A simple but major vulnerability was immediately found: the system allows a negative amount of money to be specified.
These attacks require no control of the server or client, nor the communication channel between them.

After signing in, one can do the following (excludes repeated lines):

```
Welcome kai
What would you like to do today?
1.  Deposit Money
2.  Check Balance
3.  Withdraw Money
4.  Save and Log out of my Account
How are we helping you today?2
Your current balance is 0.0
[...]
How are we helping you today?1
How much money would you like to deposit?
USD (truncated to 2 decimals):  100
[...]
How are we helping you today?1
How much money would you like to deposit?
USD (truncated to 2 decimals):  -10
[...]
How are we helping you today?2
Your current balance is 90.0
```

Additionally, this produces an even more significant vulnerability when in combination with the communication mode with the server. Namely, the amount sent is encrypted with Paillier, and is thus operated under modulo. Due to a lack of checks, depositing negative money wraps around to be the additive inverse of the number (so $-2 \bmod 20 = 18$).
While one cannot withdraw more money than exists in the account, they can still deposit the corresponding negative amount of money (mod n).

Continuing from the above block,

```
[...]
How are we helping you today?1
How much money would you like to deposit?
USD (truncated to 2 decimals):  -100
[...]
How are we helping you today?2
Your current balance is 1996616118896.61
```

And thus, we have nearly 2 trillion dollars.

Interestingly, this also allows us to discover the value n used in Paillier encryption, though this is not so much a vulnerability due to it being part of the public key. Using the raw values in cents, $n = 10000 + 199661611889661 = 199661611899661$

5 ATM Authentication Vulnerabilities

These rely upon the fact that while the user is authenticated with username, password, and a full set of RSA keys, the ATM client itself is never authenticated. This means that any user who can communicate with the server endpoint can act with all the powers provided to an ATM and its user.

This operates under the natural view of this project in which the server, client (ATM), and user are three separate entities, none of which is initially known to the others as authentic. With a valid client certificate, the server could authenticate that the client is controlled by the bank and is presumably running its software (though it would still be best to make relevant checks server-side as well). However, since client authentication is not included here, the server cannot verify that the client is not controlled by the user, which can be exploited. Note that under this model, even if both the server and client are controlled by the bank, the channel between them may not be.

5.1 Fraudulent Deposits

A simple example of this is that, with no authenticated ATM, there is not necessarily a trusted machine to accept physical cash for deposit. As such, an attacker can send unlimited DEPOSIT messages to the bank to add as much money to their account as they would like, without actually supplying the corresponding amount of cash.

5.2 Unlimited Withdrawal

While less useful than a fake deposit since a fraudulent ATM can't give the user physical cash, we can still use this to WITHDRAW unlimited money from an account. As mentioned above, one cannot WITHDRAW more money than exists in the account. However, this is only checked by the ATM, not the server. Since RSA keys authenticate the user, not the ATM, a user could use an altered version of the ATM that does not make this check. This would produce a similar effect to the negative deposit discussed above.

5.3 Circumventing Account Freeze

The login attempt limit is managed entirely by the client ATM. Thus, a fraudulent ATM could brute force (or implement other attacks that require several login attempts) without requesting an account freeze. As mentioned, due to the machine itself not being authenticated, the fraudulent ATM could simply choose not to implement the account freeze.

5.4 Hostile Account Freezing

In addition to the lack of ATM authentication, FREEZE requests include the username of the account to freeze, and the server will freeze this account without

verifying that the specified account is the same one as this current session. As such, the ATM client has the power to freeze any account, which includes unauthenticated fraudulent clients.

Even more trivially, one can indefinitely (presumably until bank staff review it) freeze anybody's account by having one failed login attempt with the target's username on a valid client. While this may counteract brute force attacks, it allow malicious actors to easily target individuals wherever they are.

6 Found Attacks

At the end of the day, we found and wrote scripts for 4 unique attacks, in addition to the negative money vulnerability. They are, in no particuluar order:

1. `attack_FRAUDCLIENT`
2. `attack_MITM`
3. `attack_FREEZE`
4. `attack_FIRE`

6.1 FRAUDCLIENT

The logic for freezing of user accounts is all done client-side; simply by commenting out the code that allows for only 3 sign-in attempts, the user of the ATM has unlimited attempts to guess the password. This leads to further attacks, such as a brute-force user password attempt attack. However, as explained later, there are more efficient means of accomplishing the same result. This pattern of the client being responsible for things that should be checked server-side has other implications. `FRAUDCLIENT` abuses the lack of authentication on the client done by the server. While the client cannot pretend to be another user without further attacks, the client can claim just about anything about the balance of the account. For example, the user can deposit as much as they want ot their account without the need to prove anything about their account. Additionally, clients can freeze whoever they want, as shown by our new 5th option to the client.

6.2 MITM

`MITM` (Man-in-the-Middle) showcases how, by inserting ourselves between the client and the server, we can fake the server's RSA key-pair to learn the `premasterSecret`. With knowledge of the `premasterSecret`, we can recover the session key, decrypt and alter all traffic in either direction between the client and server. Additionally, as the user credentials are sent in this encrypted channel, we learn the user's login as well. This means we can perfectly impersonate the user, due to the digital signature scheme being easy to forge. This `MITM` attack is very powerful and constitutes a complete break of the system's security. Undetected,

we are able to impersonate a user and perform whatever operations we want on the account, even when the user is not online. Additionally, we are able to intercept, read, and alter messages sent to and from the server at will. This attack involves no brute-forcing or complex cryptanalysis, just a simple abuse of the written protocols. This attack is impressive and is explained in detail later in the paper.

6.3 FREEZE

Alluded to in `FRAUDCLIENT`, users are able to freeze any user account they want. We can take this a step further; freezing user accounts does not require an authenticated user. All freezing requires is a secure channel, which the server will do with any client that contacts it. While the server does close the connection after a freeze operation is made, nothing stops or rate-limits connections made to the server. As a result, `FREEZE` is able to quickly shut down user accounts. It starts by freezing the accounts of the us, the authors, and then moves on to freezing random accounts. An account freeze in this system is a permanent shutdown of that account; there is no recovery and this mechanism is available to anyone with a connection.

6.4 FIRE

Earlier in this paper we spoke about the original prime number finding algorithm used by both the client and server. This algorithm is very slow and is accessed through a singleton created in the python package file. The server only uses this one singleton to run the algorithm, and if the algorithm is already running, it will not start another invocation. This leads to a simple yet effective denial of service attack on the server. By opening a connection with the server, the server kicks off a prime-number finding. This takes anywhere from 1 to 10 seconds, and during that time any other connection requests will fail due to the singleton object being in use. This is the attack; no one else can connect to the server. As soon as the connection finishes, we have a second process ready to begin a connection immediately. In practice, this makes the server unusable.

7 Man-In-The-Middle Attack

This Man-in-the-Middle attack allows us to recover the entire session key. Due to the lack of authentication of the server, our script can pose to the client as the server, using a generated public key, then relay the content to the server. While the server does send a 'certificate,' this only provides an RSA public key with no means for it to be authenticated.

In detail:

7.1 Session Handshake Phase

1. Generate a new RSA keypair. Call this the fraudulent RSA keypair. Also, generate a fraudulent Paillier keypair.

2. Gain control of the channel between the server and client. This means the attacker can read and alter packets in either direction.

3. Intercept client `HELLO` and save the plaintext `client_nonce` and `sessionID`. Pass along unaltered.

4. Intercept server `HELLO` and save the plaintext `server_nonce`, along with the server's RSA public key.
   Pass along the message, but replace the server's public key with the attacker's fraudulent public key.

5. Intercept client `KEYGEN`. Using the fraudulent RSA private key, extract the `premastersecret` and calculate the `session_key` from `premastersecret`, `server_nonce`, and `client_nonce`.
   Pass along the message, but re-encrypt `premastersecret` using the real public key instead of the fraudulent key.

6. Intercept server `KEYGEN` and save the server's Paillier public key. Pass along the fraudulent Paillier public key.

Now that we have the `session_key`, we can use this to encrypt and decrypt RC5, as well as to produce MACs.

7.2 User Authentication Phase

From this point on, all messages will be of type `ENCRYPTED`. For each of these, we can intercept each message and use the `session_key` with RC5 to decrypt the `data` field. Call this decrypted data `protected_op`.
The following is the initial sequence of user login:

1. Intercept protected `VERIFY` request. Decrypt its `data` field using RC5 and the `session_key`. We now know the username and password.

2. Ignore protected `VERIFY` response.

3. Intercept protected `CHALLENGE` request. Save the `keys` field as the client's public RSA keys.

4. Ignore protected `CHALLENGE` response.

After this, only `DEPOSIT`, `WITHDRAW`, `CHECK` messages will be sent by an unaltered client. We can read and alter any of these messages.

7.3 Encrypted Requests

For requests (e.g. `DEPOSIT` and `WITHDRAW`), we can do the following:

1. Intercept request. This will contain the following fields:
   - `id`: Will always be `2`, representing `ENCRYPTED`.
   - `data`: This is the encrypted `protected_op`.
   - `mac`: This is a MAC of `data`.
   - `signature`: This is a digital signature of `data` using the client's RSA keys.
2. Decrypt `protected_op`. This will contain the following fields:
   - `id`: The Protected Operation ID (e.g. `DEPOSIT` or `CHECK`).
   - `nonce`: A random number.
   - `data`: The raw data transmitted by the request.
   - `value`: Another store of raw data transmitted by the request used by some request types.
3. Read and/or alter `protected_op.data` and `protected_op.value` at will. If altering Paillier-encrypted data (e.g. `DEPOSIT` or `WITHDRAW` amounts), Decrypt with the fraudulent Paillier private key, read/alter as necessary, then re-encrypt with the real Paillier public key.
4. Re-encrypt the modified `protected_op` and put it back in the request.
5. Using the `session_key`, calculate and add the new MAC (replacing the old one).
6. Calculate the `new_signature` from `new_encrypted_data`, old signature (y_1, y_2), and the client's public key e, N
   (a) `new_hash` $= sha1($`new_encrypted_data`$) >> 32$
   (b) Recover $r = y_1^e \bmod N$
   (c) Calculate $H_r = sha1(r)$
   (d) Then `new_signature` $= (y_1, H_r \oplus$ `new_hash`$)$

   And replace the request's `signature` with the `new_signature`.

Note the strategy we use above for altering the digital signature. This works because the implemented signature scheme is malleable. Note that this scheme is generally used with a hash as the message, since it is limited to the size of H_r (which is also a hash).
The following is how the digital signature scheme works:
Signing using private key d, N and message m:

1. Generate a random $0 \leq r < N$
2. Let $H_r = sha1(r)$
3. Return $(r^d \bmod N, m \oplus H_r)$

Verifying with public key e, N, message m, and signature (y_1, y_2):

1. Recover $r = y_1^e \bmod N$
2. Find $H_r = sha1(r)$
3. Check that $m == H_r \oplus y_2$

However, as noted above, this scheme is flawed since we can change the message then alter the signature by recovering r and H_r from y_1 using the client's public key in the same manner as during verification, then calculating the new

$$y_2 = m_{new} \oplus H_r$$

Alternately, if we don't have the signer's public key (which we should), we can do the following:

$$y_{2_{new}} = m_{new} \oplus H_r = m_{new} \oplus m_{old} \oplus m_{old} \oplus H_r = m_{new} \oplus m_{old} \oplus y_{2_{old}}$$

We can also create new messages rather than altering them by reusing previous values of (y_1, y_2) or r and H_r, as reuse is not checked for these.

7.4 Encrypted Responses

Most encrypted responses after the user is authenticated (i.e. `DEPOSIT`, `WITHDRAW`, and potentially `FREEZE`) do not have a response. Nevertheless, we can alter the responses that do come through (e.g. `CHECK`).
This operation is identical to the process for reading and altering requests, except that the signature should be left as `null`.

8 Key Generation

Session keys are generated using the following parameters:

- `premastersecret`: Generated by the client, encrypted using the server's RSA public key, then decrypted by the server.
- `client_nonce`: A random 64-bit prime number generated by the client and sent in the cleartext.
- `server_nonce`: A random 64-bit prime number generated by the client and sent in the cleartext.

As discussed earlier, the usage of only prime nonces may pose a vulnerability due to it significantly decreasing the number of possible nonces. However, due to them being sent in cleartext, this may not be so useful.
Also note that using a man-in-the-middle attack, the `premastersecret` can be retrieved.

TODO: I BET THERE'S A LOT OF PROBABILITY AND MATH CRYPTANALYSIS HERE

notes: key both for rc5 and hmac (same key) is produced by hashing the concatenated binary strings (so "01010101...") of the premaster secret, server nonce, and client nonce then it is left shifted until it aligns with 8 bits. I think this means that the session key will be a multiple of 8 bits (so a full number of bytes) but the leftmost bit will always be 1, and any shifts will result in the rightmost bits being 0.