Cryptography and Network Security Final Project - Blackhat

Ryan Prashas Leith Reardon Dylan Ross

August 18, 2020

Contents

1 Password Management **1**

2 ATM Authentication **1**

3 Connection Sequence Weaknesses **2**
3.1 Connection Struct 2
3.2 Sequence Vunerabilities 2
3.3 Using the Vunerability 2
3.4 Code to take advantage 3
3.5 Limitations 3

4 Secrets **3**

Introduction

The project that we were given is written in C, and uses various libraries in order to implement a large number of options for both symmetric and public key cypto systems in order to implement a pseudo-ssh algorithm in order to create a secure connection between a bank and an atm. Below are our attacks on the implementation.

1 Password Management

Done by Dylan Ross

In the project we were given to attack, the first vulnerability that we found became apparent immediately upon reading the documentation. The code does not make use of passwords of any kind, and instead allows the ATM to check the balance of, deposit into, and withdraw from arbitrary accounts without authentication. The could trivially be leveraged by an attacker to steal money from other people's accounts by selecting a different account during a withdrawal.

It could be said that this attack is outside of the scope of attacking the cryptographic communication between an ATM and a bank as the ATM could be handling user authentication locally, but that would likely not be feasible. In order for that scheme to work, every ATM would need to locally store the credentials of every user in the bank, which would require a significant amount of memory and as such would drastically increase the price of the ATM machines themselves. Additionally, this scheme would require updating the local storage of every ATM owned by the bank whenever a new user account is created, which would cost a lot of time and money.

2 ATM Authentication

Done by Dylan Ross

Another vulnerability that we found in our assigned project is that the bank never authenticates that the client is an actual ATM once a connection is established. As such, there is nothing to distinguish an attacker connecting to the bank from their laptop from an ATM that can ensure that various rules are being followed. For example, we assumed that the ATM would realistically check that the user actually inserted money before sending a message to the bank saying to deposit money into their account. However, an attacker being able to connect using a different device can send arbitrary deposit messages to the bank in order to add money into their account and withdraw it from an actual ATM.

This attack, when combined with the lack of passwords mentioned in section 1, could also allow an attacker to bypass any local credential checking done by the ATM in order to send withdraw messages with a different account number, which would empty the target account's balance without the money going to anyone else. However, even if passwords were implemented on the bank side to prevent such an attack, an attacker would still be able to arbitrarily deposit money into their own account as they would know their own credentials and be able to send them to the bank.

We were able to generate a working proof of concept for this attack by creating a copy of the atm.c file and modifying it slightly for ease of use and to remove parts that were unneccesary for the attack. After compiling this new file with the same flags used for atm.c (done by copying from the make file), we could send arbitrary deposit messages and received the correct responses from the bank. However, this attack could not be shown very well using the current implementation of the bank because the bank does not store account balances at any point. Instead, it initializes every account to a pre-determined balance at runtime. As such, our proof of concept cannot be shown to work between sessions on this implementation, but neither do any legitimate transactions done by the ATM itself. Due to the theory behind the attack and the proper response codes received by the proof of concept, there is no reason to believe this attack would not work on an actual bank implementation that saves user balances.

3 Connection Sequence Weaknesses

Done by Ryan Prashad

In the provided blackhat code, the other team uses an attribute of every connection labeled the sequence in order to prevent replay attacks. This method works by incrementing a counter for each message sent to verify that messages are being sent in order and are not being taken out of context to randomly send to the bank in a future session to force unexpected behavior.

3.1 Connection Struct

The connection struct in this code basically carries all the applicable encryption schemes locally for the bank and atm. The agreed upon suite in the handshake is imported into this struct and then used for all the send/receive calls. The sequence attribute of the `ssh_t` struct is the counter that gets incremented from message to message in this code between the bank and atm. Each time the recv or send commands from the ssh call are used, the connection sequence variable is correctly incremented by the pad/unpad function.

3.2 Sequence Vunerabilities

The way that this code is written, connection.sequence is there but does not guard against injecting the same message twice to the bank server within a single session. If there was a man in the middle between the bank and atm and they intercepted the ciphertext, the ciphertext could be injected back into the bank receive call to do the same action again. An example of this could be draining someones bank account by sending the same false withdraw statement over and over again, or depositing money into your own bank account by interception your own message and sending it to the bank to infinitely deposit money there.

3.3 Using the Vunerability

Because the ssh class never asserts that connection.sequence has any order, sending the same message twice could technically work because the encryption is still valid in this session, and sequence is never checked against the previous sequence to see if the message is in the right order (of increasing sequence).

3.4 Code to take advantage

Modified atm code found in seqexploit.c assumes we've intercepted a message going to the bank from the atm accross localhost already. Since this message is already encrypted, we represent the input to this code as a string of 2 digit hex numbers. This code is then injected by just sending it to the bank (even though we are not an atm at this point). If it works, the bank should handle this message as normal and proceed and we've succeeded with injection.

3.5 Limitations

Due to the way the bank file was written, we cannot actually connect an atm and our exploit code to the bank server. If the bank.c file were setup to take a variable amount of connections, we can spoof being an atm or just send the intercepted code to the bank and trigger the recieve call on its port. The exploit code as is just interprets the hex encoded representation to an unsigned character array and sends the message as if it were an atm.

4 Secrets

Done by Dylan Ross

In our assigned project, all variables used for public key cryptography are stored in the `sshConstants.h` file. This allows both the ATM and the bank to have access to the public variables required for the signatures and verifications to work properly without much hassle. However, due to the open-source nature of the project, this also allows attackers to read the private keys as well. This issue could have been avoided by having the private keys be in a file that represented local storage on the bank and/or the ATM as oppossed to hard-coded.

As a result of this vulnerability, an attacker can sign messages as the bank and therefore setup an authenticated ssh session with an atm. Furthermore, this false bank can approve all withdraw requests sent by the atm, allowing the attacker to withdraw as much money as they would like regardless of their balance.