Project 4: ATM Design and Implementation

Due dates
Apr. 25, 11:59 PM (Part 1)
May 2, 11:59 PM (Part 2)

In this project, you will design and implement a prototype ATM/Bank system. Then you will get a chance to (try to) attack another team’s design! I have tried to make these instructions as explicit as possible. Read them carefully; if anything is unclear, please ask for clarification well in advance.

Overview

1. You may work in teams of at most two people. Email your instructor who is in your team, and include both names in your writeups.

2. You will design and implement two programs: an ATM and a bank. (You may also find it useful to create various auxiliary files defining classes and functionalities that can be used by the ATM and Bank classes.)

3. You will be provided with stub code for the ATM, the bank, and a router that will route messages between the ATM and the bank. The stub code will allow the ATM and the router to communicate with each other, and the router and the bank to communicate with each other. The router will be configured to simply pass messages back-and-forth between the ATM and the bank. (Looking ahead, the router will provide a way to carry out passive or active attacks on the “communication channel” between the bank and the ATM.)

4. You will design a protocol allowing a user to withdraw money from the ATM. Requirements include:
   - The ATM card of user XXX will be represented by a file called XXX.card.
   - The user’s PIN must be a 4-digit number.
   - User balances will be maintained by the bank, not by the ATM.
   - You need not support multiple ATMs connecting to the bank simultaneously.

Of course, as part of the design process you will want to consider security…

5. You will then implement your protocol. Most of your work should involve the ATM and bank programs, with no (or absolutely minimal) modifications to the router.
Part 1—Basic Functionality

Your programs should support the following functionality:

- The bank should be set up with three user accounts, for Alice, Bob, and Carol. Alice’s and Bob’s balance should be initialized to $100, and Carol’s balance should be initialized to $0. In addition to the ATM, bank, and router programs, you will specify the files Alice.card, Bob.card, and Carol.card, and PINs defined for these users (see “Part 1—Deliverables” for how these PINs should be stored and submitted).

- Your programs must not take any command line arguments, and should be run in the following order: router, then bank, and finally atm.

- The stub code that we provide contains the networking operations you will need. The router, bank, and ATM all run on different ports on localhost (so you need not even be connected to the Internet for this to work).

- Bank commands: The bank should support the following commands:

  - deposit <user> <amt> will add $amt to the account of user. After successful completion of this command, this should print “$amt added to user’s account” to stdout.

  - balance <user> should print the current balance of user to stdout.

Withdrawals from the bank are not supported.

Here is an example transcript, with the bank balances initialized as stated above:

BANK: balance Alice
$100

BANK: balance Carol
$0

BANK: deposit Carol 2
$2 added to Carol’s account

BANK: balance Carol
$2

BANK: ...

- ATM commands: The ATM should support the following commands:

  - begin-session <user> is supposed to represent user walking up to the ATM and inserting his or her ATM card. It should then read from user.card, and prompt for a PIN. If the correct PIN is entered, print “authorized” and then allow the user to execute the following three commands. Otherwise, print “unauthorized” and continue listening for further begin-session commands.
– **withdraw <amt>** should print “$amt dispensed” if the currently logged-in user has sufficient funds in their account. Their account should be debited accordingly. Otherwise, print “insufficient funds”. (If no user is currently logged-in, print “no user logged in”.)

– **balance** should print the current balance of the currently logged-in user. (If no user is currently logged-in, print “no user logged in”.)

– **end-session** terminates the current session and prints “user logged out”. The ATM should then continue listening for further **begin-session** commands. (If no user is currently logged-in, print “no user logged in”.)

The ATM should support an unlimited number of **withdraw** and **balance** commands per session. Deposits at an ATM are not supported.

Here is an example transcript, assuming Alice’s balance is $100 (and this balance is not modified at the bank during this execution), that the file Alice.card is present, and that Alice’s PIN is 0000. Note the prompts, which change as a user logs in:

```
ATM: balance
no user logged in

ATM: begin-session Alice
PIN? 0000
authorized

ATM (Alice): balance
$100

ATM (Alice): withdraw 1
$1 dispensed

ATM (Alice): balance
$99

ATM (Alice): end-session
user logged out

ATM: ...
```

**Part 1—Deliverables**

Submit your implementation (all of the .c and .h files that collectively encompass your code, including any files we provided, even if you do not modify them), along with a Makefile. Building all of your executables should require only running “make”. We have provided an initial Makefile; modify this as necessary (or make your own). In addition to the implementation, you should write a design document that:
1. Describes your protocol in sufficient detail to understand the security mechanisms you put into place, without having to look at the source code, and

2. Discusses the threat model you assumed, along with a brief discussion of how your protocol counters those threats.

3. You can also mention threats that you chose to ignore because they were unimportant, as well as threats that were important but you were unable to address.

Your protocol should be secure against an adversary who is not in possession of a user’s ATM card, even if the adversary knows the user’s PIN.

However, we assume that the bank computer cannot be compromised, nor can the memory on the ATM be examined. In particular, you do not need to defend against the following: (1) Using code disassembly to recover secret keys, or (2) Attacks that require restarting the bank.

What to submit: Submit the following via the submit server:

1. Your bank, ATM, and router code (and any supporting code), as well as compiled versions of the bank and ATM within the VM we have provided. The code should not reveal any secret/private keys that might be present in the compiled programs. (E.g., if your bank shares a secret key 12345 with the ATM, then you can compile the Bank/ATM with this key written into the code, but make sure to change the key to 00000 in the source code afterwards.) The secret/private keys are the only part of your code that you can zero-out.

2. The files user-info/Alice.card, user-info/Bob.card, and user-info/Carol.card. (Recall that the ATM reads .card files as a result of a begin-session command.)

3. Files user-info/Alice.pin, user-info/Bob.pin, and user-info/Carol.pin containing the PINs for the users. (Note: these files are for testing purposes; you are not required to use them in your system in any way.)

4. Your design document, typeset in a sane format (pdf or ps).

Part 2 — Attacking Another Team’s Implementation

After submission, each team will be given the chance to attack another team’s implementation. Specifically, each team will be given the following information submitted by some other team:

- The code submitted in Part 1, as well as the compiled bank and ATM programs.
- The files user-info/Alice.card and user-info/Carol.card. (The Bob.card file will not be given.)
- The PINs for Bob and Carol. (The PIN for Alice will not be given.)
• The design document.

In your attack, you may arbitrarily modify the router code and the *.card files. A successful attack will be any attack that results in a net outflow of money to the attacker. By way of illustration, examples of successful attacks would be (these are not exhaustive):

• Withdrawing any money from Alice’s or Bob’s account.

• Withdrawing $1 from Carol’s account without making any prior deposit at the bank.
  (Note that although the attacker has Carol’s ATM card and PIN, the bank starts out with Carol’s account initialized to $0.)

• Depositing $1 to Carol’s account and then withdrawing $2.

**Deliverable:** Submit a vulnerability analysis of the assigned implementation. This analysis should describe your attack(s), if any, at both a high level (so someone reading it can understand what you did) as well as in sufficient detail to enable someone to replicate your attack. You can also describe any vulnerabilities you were able to find but not exploit (due to limitations of the project); e.g., an attack that would require multiple ATMs to connect to the bank at once. If you were unable to find any attack, simply explain what types of exploits you looked for. *Your vulnerability analysis should begin with the name(s) of the students whose protocol you are attacking, and a 1-paragraph summary of what attacks (if any) you were able to find.*

Submit your vulnerability analysis via the submit server. This, too, must be typeset in a sane format (pdf or ps). *In your analysis, please include the names of the students whose protocol you are attacking.*

You should also submit (via the submit server) any code you wrote to implement your attack. This will likely include the modified router code, but could include any other utilities you wrote as well. Make sure to provide details on how to use your program(s) as part of your vulnerability analysis.

**Grading**

Part 1 will be worth 100 points and will graded as follows: 30% of the grade will be based on automated tests by the TAs that your submission achieves the basic functionality. 50% of the grade will be based on my evaluation of the security of your protocol, based on my reading of your design document. (If your design document does not correspond to your implementation, you will be given no credit—if you are not able to implement some feature that you think should be present, be honest about it.) 20% of your grade will be based on the security of your implementation, as evidenced by whether the other team is able to find a successful attack.

Part 2 is worth 20 points. A successful attack (that is also described clearly in the vulnerability analysis) will automatically be awarded 20 points. Even if you are not able to find a successful attack, you can still get points by (1) pointing out potential vulnerabilities that you were not able to successfully exploit, and/or (2) writing a good vulnerability analysis.
analysis that outlines the exploits you looked for and argues why they are not present in
the implementation you were given to attack.

I reserve the right to award extra points for multiple attacks, or particularly clever
attacks, so be creative!

Extra credit—Measurement study of certificate revocation lists

The extra credit is not team-based.

In this extra credit assignment, you are encouraged to perform an analysis of certificate
revocation lists (CRLs). Recall that certificate authorities (CAs) append revocation infor-
mation to the end of a file (the CRL). Clients periodically obtain CRLs from the CAs whom
they trust, so that they may know what certificates to avoid in the future. As we discussed
in class, CRLs are generally not downloaded by clients in a timely fashion, leaving many
clients with out-of-date revocation state, thereby potentially leaving them (unnecessarily)
vulnerable.

What we did not discuss, however, is the nature of the revocations themselves. For this
extra credit assignment, obtain a set of CRLs, analyze them, and report on your findings.
Be creative! Below are some questions to get you started. Feel free not to answer them but
note that the amount of extra credit you get will be a function of how deep you dig, how
much data you collect, and how creative you are in analyzing the data.

• Where/how did you obtain your data? How representative is it?
• How frequently do CAs issue revocations? Has this changed significantly over time?
  Why or why not?
• What are the reasons that certificates get revoked?
• Has the latest OpenSSL bug resulted in any revocations? Can we determine that the
  bug is the root cause? How?

You may ask me for data and perform your own data analysis, but this will result in
fewer (extra) points than if you were to perform and report on your own data collection.

What/when to submit: No later than May 8, submit a typeset document (pdf or ps)
via the submit server. This document should describe where you obtained your data, how
you went about analyzing it, and it should detail your findings. If you include plots and
data analysis (and I hope you do), they should be easily legible, with well-labeled axes and
clear, concise descriptions.