University of Maryland
CMSC414 — Computer and Network Security
Professor Jonathan Katz

Project 2

Due dates

Mar. 11, 11:59 PM (Part 1)
Apr. 1, 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! The instructions are long, but this is because I have tried to make them as explicit as possible. Read the instructions carefully; *if anything is unclear, please ask for clarification well in advance.*

Overview

1. 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.)

2. 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.)

3. 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...

4. You will then implement your protocol. Most of your work should involve the ATM and bank programs, with no (or 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 have the files Alice.card, Bob.card, and Carol.card, and PINs defined for these users.
- Your programs should be run as follows (in this order):
  1. java Router `bank-port ATM-port`
  2. java Bank `bank-port`
  3. java ATM `ATM-port`

  The port numbers must be distinct, positive integers greater than 1024.
- The bank should support the following commands:
  - `deposit user amt` will add `$amt` to the account of `user`. After successful completion of this command, return "`$amt` added to `user`'s account".
  - `balance user` should return the current balance of `user`.

  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:  ...
```

- The ATM should support the following commands:
  - `begin-session user` is supposed to represent `user` walking up to the ATM and inserting his/her ATM card. Read from `user`.card, and prompt for a PIN. If the correct PIN is entered, return "authorized" and then allow the user to execute the following three commands. Otherwise, return "unauthorized" and continue listening for further `begin-session` commands.

- `withdraw amt` should return "$`amt` dispensed" if the currently logged-in user has sufficient funds in their account. Their account should be debited accordingly. Otherwise, return "insufficient funds". (If no user is currently logged-in, return "no user logged in".)
- `balance` should return the current balance of the currently logged-in user. (If no user is currently logged-in, return "no user logged in".)
- `end-session` terminates the current session and returns "user logged out". The ATM should then continue listening for further `begin-session` commands. (If no user is currently logged-in, return "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

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. (As part of this discussion, 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 only needs to defend against attacks that would be feasible in a "real-world" deployment where, for example, 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.
2. Attacks that require restarting the bank.

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.

Hand in a hardcopy of your design document in class on March 13. *Please write your design document using a word processor such as latex or Word. Do not submit a handwritten document, or a printout of a text file.*

Submit the following to the TA:

1. Your bank, ATM, and router code, as well as compiled versions of the bank and ATM. The code should *not* reveal any secret/private keys that might be present in the compiled programs. (I.e., 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.)
2. The files Alice.card, Bob.card, and Carol.card.
3. Files AlicePIN.txt, BobPIN.txt, and CarolPIN.txt containing the PINs for the users. (Note: your programs should not read these files; they are for the TA's testing purposes only.)
4. Your design document.

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:

1. The bank, ATM, and router code, as well as the compiled bank and ATM programs.
2. The files Alice.card and Carol.card. (The Bob.card file will *not* be given.)
3. The PIN for Bob and Carol. (The PIN for Alice will *not* be given.)
4. The design document.

The router code and the .card files can be arbitrarily modified. 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):

1. Withdrawing any money from Alice's or Bob's account.

2. 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.)

3. 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 a 1-paragraph summary of what attacks (if any) you were able to find.* **Hand in a hardcopy of your vulnerability analysis in class on April 3**. *Please write your vulnerability analysis using a word processor such as latex or Word. Do not submit a handwritten document, or a printout of a text file. In your analysis, please include the names of the students whose protocol you are attacking.*

You should also submit (using the submission script) 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 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 will be worth up to 50 bonus points, though I reserve the right to award more points for multiple attacks, or particularly clever attacks. A successful attack (that is also described clearly in the vulnerability analysis) will automatically be awarded 50 points. Even if you are not able to find a successful attack, you can get these bonus points by (1) pointing out potential vulnerabilities that you were not able to successfully exploit, and/or (2) writing a good vulnerability analysis that outlines the exploits you looked for and argues why they are not present in the implementation you were given to attack.