University of Maryland
CMSC414 — Computer and Network Security (Fall 2017)
Instructor: Mike Marsh (project originally created by Jonathan Katz)

Course Project: ATM Design and Implementation

Due dates
December 4, 2:00:00 PM (Part 1)
December 11, 2:00:00 PM (Part 2)
Note: There is no late acceptance for either part.

In this project, you will design and implement a prototype ATM/Bank system. Then you will get a chance to (try to) attack other teams' designs! 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 (use the subject "CMSC 414: Project team"), CC the other teammate, and include both names in your writeups.

2. You will design and implement three programs: an *ATM*, a *bank*, and an *init* program that initializes state for them. (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.
   - You also do **not** need to maintain state between restarting the bank (e.g., all user balances can be maintained in memory).

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, bank, and init programs, with no (or absolutely minimal) modifications to the router.

1 Part 1—Basic Functionality

This section describes the various programs you will need to write for this project. We have provided stub code for much of it, along with some helper utility code (a hash table and a list). You are welcome to use these, or to write your own, but the way that the code communicates (the ATM and bank route their messages through the router) must remain the same.

Also, in this section, we describe outputs for the program: make sure that these match *exactly* (the outputs do not have periods at the end, for instance) so that we can use automatic graders.

Building the programs: what and how

From your submission directory, there should be a Makefile such that when we run `make` it creates the following programs:

- `bin/atm`
- `bin/bank`
- `bin/init`
- `bin/router`

The `atm`, `bank`, and `router` programs must be implemented in C. `init` can be a script or C program. 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).

Your submissions must build with stack guard turned off (`-fno-stack-protector` and executable stacks (`-z execstack`) — as already specified in the Makefile provided. As you all know, this is bad practice, and you should not do this outside of this class! But it will require you to be all the more on your toes—and should make the break-it phase of the project even funner.

Invoking the programs

The `init` and `router` programs can be run in either order, but both must be run before `bank`, which in turn must be run before `atm`. The order of these will be clear from what they create.

1.1 The `init` program

The `init` program takes one command line argument, as follows:

```
% <path1>/init <path2>/<init-fname>
```

This program creates two files: `<path2>/<init-fname>.bank` and `<path2>/<init-fname>.atm` — the point of the paths here being that the program should not be creating the files in a hard-coded directory, but rather the user should be able to specify where they go.

File contents: The contents of these files can be *whatever you want*. In fact, their contents are an important part of your protocol's design and security. When you design them, keep in mind:

- When the `bank` program is started, we pass it the `<init-fname>.bank` file, and when the `atm` program is started, we pass it the `<init-fname>.atm` file.
- *Only* the bank can access the `.bank` file, and only the ATM can access the `.atm` file—they cannot access one another's.
- Attackers are *not* allowed to access either of these files.

Behavior of the `init` program:

- If the user fails to provide precisely one argument, then print "`Usage: init <filename>`" and return value 62.
- Otherwise, if either `<path2>/<init-fname>.atm` or `<path2>/<init-fname>.bank` already exist, print "`Error: one of the files already exists`" and return value 63 without writing over or creating *either* file.
- Otherwise, if for any other reason the program fails, print "`Error creating initialization files`" and return value 64 (you do not need to delete or revert any files you may have created).
- Otherwise, if successful, print "`Successfully initialized bank state`" and return value 0.

1.2 The `bank` program

The `bank` program takes one command line argument, as follows:

```
% <path1>/bank <path2>/<init-fname>.bank
```

That is, it is called with a `<init-fname>.bank` that was created by the `init` program. You can assume that it will NOT be called with any other file than one that was created with your `init` program.

Behavior of the `bank` program:

- If `<init-fname>.bank` cannot be opened, print "`Error opening bank initialization file`" and return value 64.
- Otherwise, present the bank prompt "`BANK: `" and process bank commands, as described below.

Bank commands:
The bank should support the following commands:

- `create-user <user-name> <pin> <balance>`
  - Inputs:
    - `<user-name>`: a name that can consist only of upper and lower case characters ([a-zA-Z]+). (Valid user names are at most 250 characters.)
    - `<pin>`: a four-digit number [0-9][0-9][0-9][0-9]
    - `<balance>`: a non-negative integer [0-9]+ at most what can be represented with an `int`.
  - Behavior:
    - If the inputs to the command are invalid, then print "`Usage: create-user <user-name> <pin> <balance>`" there should be no side effects in this case.
    - Otherwise, if there is already a user with that (case-sensitive) name, then print "`Error: user <user-name> already exists`" there should be no side effects in this case.
    - Otherwise, it should create a user named `<user-name>` in the bank with an initial balance of `$<balance>` It should also create a file called `<user-name>.card` in the current directory (if you run `./<path>/bank` then it should create the file in `.`, not in `<path>`). Like with the `<init-fname>` files, the contents can be whatever you want: it's part of your design.
      - If it is unable to create the file, then it should print "`Error creating card file for user <user-name>`" and roll back any changes the bank may have made.
      - Otherwise, if it successfully creates the file, then it should print "`Created user <user-name>`"
    - In any of the above cases, it should return the user back to the bank prompt for more commands.
    - **Note:** Your protocol cannot modify card files after creating them.
    - **Note:** The bank should not have any user accounts pre-established: these will be created at run-time with a `create-user` command.

- `deposit <user-name> <amt>`
  - Inputs:
    - `<user-name>`: a name that can consist only of upper and lower case characters ([a-zA-Z]+). (Valid user names are at most 250 characters.)
    - `<amt>`: a non-negative integer [0-9]+ at most what can be represented with a `int`.
  - Behavior:
    - If the inputs to the command are invalid, then print "`Usage: deposit <user-name> <amt>`": there should be no side effects in this case.
    - Otherwise, if there is no such user, then print "`No such user`": there should be no side effects in this case.
    - Otherwise, if the `<amt>` would cause integer overflow, then print "`Too rich for this program`": there should be no side effects in this case.
    - Otherwise, the command will add `$amt` to the account of `user-name`. After successful completion of this command, this should print "`$amt added to <user-name>'s account`"
- `balance <user-name>`
  - Inputs:
    - `<user-name>`: a name that can consist only of upper and lower case characters ([a-zA-Z]+). (Valid user names are at most 250 characters.)
  - Behavior:
    - If the inputs to the command are invalid, then print "`Usage: balance <user-name>`"
    - Otherwise, if there is no such user, then print "`No such user`"
    - Otherwise, if `<user-name>`'s current balance is `<balance>`, then the command will print "`$<balance>`"
    - No matter the input, this command should have no side effects (beyond printing).
- Any other commands are invalid (note that withdrawals are not supported at the bank), and should result in printing "`Invalid command`"

Example transcript

Here is an example transcript of using the bank: suppose that no users exist yet.

```
BANK: balance Alice
No such user

BANK: create-user Alice 1234 100
Created user Alice
```

```
BANK: deposit Alice 2
$2 added to Alice's account

BANK: balance Alice
$102

BANK: ...
```

1.3 The `atm` program

The `atm` program takes one command line argument, as follows:

```
% <path1>/atm <path2>/<init-fname>.atm
```

That is, it is called with a `<init-fname>.atm` that was created by the `init` program. You can assume that it will NOT be called with any other file than one that was created with your `init` program.

Behavior of the `atm` program:

- Similar to the bank, if `<init-fname>.atm` cannot be opened, print "`Error opening ATM initialization file`" and return value 64.
- Otherwise, present the bank prompt "`ATM: `" and process ATM commands, as described below.

ATM commands:
The ATM should support the following commands:

- `begin-session <user-name>`

  This command is supposed to represent `<user-name>` walking up to the ATM and inserting his or her ATM card. It should then read from `<user-name>.card`, and prompt for a PIN. If the correct PIN is entered, print "`Authorized`" and then allow the user to execute the other three ATM commands (`balance`, `withdraw`, and `end-session`). Otherwise, print "`Not authorized`" and continue listening for further `begin-session` commands. More details here:

  - Inputs:
    - `<user-name>`: a name that can consist only of upper and lower case characters ([a-zA-Z]+). (Valid user names are at most 250 characters.)
  - Behavior:
    - If there is a user already logged in, then print "`A user is already logged in`"

    * Otherwise, if the inputs to the command are invalid, then print "`Usage: begin-session <user-name>`": there should be no side effects in this case.
    * Otherwise, if there is no such user registered with the bank, then print "`No such user`"
    * Otherwise, if your protocol uses the card files, then it should look for `<user-name>.card` in the current directory (e.g., if you run `./<path>/atm` then it should look for `<user-name>.card` in `.`, not in `<path>`). If it is unable to open the file, then it should print "`Unable to access <user-name>'s card`"
    * Otherwise (if there is no error accessing the card file), then the program should prompt the user for his or her pin by printing "`PIN?` "
    * The user should then enter a `<pin>` consisting of a four-digit number [0-9][0-9][0-9][0-9].
    * If the pin entered is invalid (not formatted correctly) or if it is unable to authenticate the user, then the program should print "`Not authorized`" and return to the ATM prompt ("`ATM:` ").
    * Otherwise, it should print "`Authorized`" and enter the authorized atm prompt ("`ATM (<user-name>):` ").

- `withdraw <amt>`
  - Inputs:
    * `<amt>`: a non-negative integer [0-9]+ at most what can be represented with an `int`.
  - Behavior:
    * If no user is logged in, then print "`No user logged in`"
    * Otherwise, if the inputs to the command are invalid, then print "`Usage: withdraw <amt>`"
    * Otherwise, if the user has insufficient funds, then print "`Insufficient funds`"
    * Otherwise, if the user has sufficient funds, then print "`$<amt> dispensed`" and those funds should be reduced from the logged-in user's balance.
    * If a user was logged in, then that user should stay logged in after this command.

- `balance`
  - Behavior:
    * If no user is logged in, then print "`No user logged in`"
    * Otherwise, if the inputs to the command are invalid, then print "`Usage: balance`"
    * Otherwise, print if the user's current balance is `<balance>` then print "`$<balance>`"
    * If a user was logged in, then that user should stay logged in after this command.

- `end-session`

  This command represents the user terminating his or her session and logging out of the ATM.

  - Behavior:
    - If no user is logged in, then print "`No user logged in`"
    - Otherwise, terminate the current session and print "`User logged out`". The ATM should then continue listening for further `begin-session` commands.

- Any other commands are invalid (note that deposits are not supported at the ATM), and should result in printing "`Invalid command`"

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

Example transcript

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 1234 (continuing the example from the `bank` program). Note the prompts, which change as a user logs in:

```
ATM: begin-session Aliceeee
No such user

ATM: begin-session Alice
PIN? 1234
Authorized

ATM (Alice):  balance
$100

ATM (Alice):  withdraw 1
$1 dispensed

ATM (Alice):  balance
$99

ATM (Alice):  end-session
User logged out

ATM: balance
No user logged in

ATM: ...
```

Threat model

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, and vice versa. The attacker is also in control of a router on the path between the ATM and the bank, and can inspect, modify, drop, and duplicate packets—as well as create wholly new packets—to both ATM and bank.

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, or (3) Attacks that involve inspecting the contents of the files created by the `init` program: think of these like secret information that a bank operator inputs at the bank server and when installing the ATM (though, again, the specific contents of these files are purely up to you).

Part 1—Deliverables

Submissions must be your **team** repository (even if you're working solo), with the hash provided through ELMS as usual. *Both members of the team must submit hash values independently.*

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 must write a *design document* in which you:

1. Describe your overall protocol in sufficient detail for a reader to understand the security mechanisms you put into place, without having to look at the source code. This must include the format of the files you create (with the `init` and `bank` programs) and the messages you send between ATM and bank.

2. List, one by one, the specific attacks that you have considered and describe how your protocol counters these threats. This is critical for how we will be grading this part of the project (see Grading below).

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.

This must be submitted in a sane format: plain text (including markdown), ps, or pdf.

2 Part 2 — Attacking Other Teams' Implementations

After submission, each student will be given the chance to attack *another* team's implementation. Specifically, each student will be given the full submission of one team, including all

of the code and the design document, via a new git repository. You will obtain the name of this repository by running `ssh git@gizmonic.cs.umd.edu info break-it`.

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 a user's account without access to his or her card file and/or PIN.
- Withdrawing more money from a user's account than the balance permits.
- Remotely depositing money to a user's account (i.e., without accessing the bank's command line interface).
- Learning a user's balance without having access to his or her card file and/or PIN.

Deliverable: In your `break-it/<uname>` repository, create an `analysis` directory. *All of your files for this part must be in this directory.*

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 git, with the usual hash submission on ELMS. This, too, must be typeset in a sane format (plain text or markdown, pdf, or ps). *In your analysis, please include the team name of whose project you are attacking.*

You should also submit (in your git repository) 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.

3 Grading

Part 1 will be worth 100 points and will be graded as follows: 25% of the grade will be based on automated tests that your submission achieves the basic functionality (e.g., proper account balances, not permitting withdrawals with insufficient funds, etc.). The remaining is based on your protocol's ability to protect against attacks.

Each distinct attack and countermeasure you present in your design document is an additional 15%. ("Distinct" here is pretty loose: it doesn't mean that, e.g., all confidentiality attacks are equivalent, but rather that the attacks are morally different. For example, if there is a vulnerability such that modifying a part of a packet to some number x causes the user to withdraw $\$x$ instead of their intended amount, then exploiting that vulnerability

with $x = \$200$ and $x = \$300$ are fundamentally the same attack.) You *must* list the attacks and countermeasures one-by-one in your document to make it clear for grading. You may include additional attacks beyond the 5 required for full credit. While this will not increase your score, the more attacks you prevent, the greater your chances of surviving the break-it phase of the project.

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, feel free to describe it so long as you state clearly that it is not implemented; for well-described attacks and designs without implementation, partial credit will be available.

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 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!