Build-it

Project specification	Build-it spec
On-time due date	Dec 7, 2020
Starter files	p4-starter-tcp.t...

Note: This is a **group project**; you can work in groups of at most 3.

This is part one; you have three weeks to work on the Build-It portion, and then will have one week to work on the Break-it.

There are multiple opportunities for extra credit, including at most 2 extra attacks to defend against, successfully attacking many other people's projects during the Break-it phase, and performing well in the competition overall.

Be sure to get your basic functionality tests to work.

1 Overview

This project will be a team project. Project 4 is in two parts: (a) Build-it and (b) Break-it. This document contains details of part(a) of the project4, where students in teams of up to 3 students will collaborate to write a protocol for communication among `atm` and a `bank`. There will be two programs. One program, called `atm`, will allow `bank` customers to withdraw and deposit money from their account. The other program, called `bank`, will run as a server that keeps track of customer balances.

2 Security Model

`atm` and `bank` must be implemented such that only a customer with a correct card file can learn or modify the balance of their account, and only in an appropriate way (e.g., they may not withdraw more money than they have). In addition, an `atm` may only communicate with a `bank` if it and the `bank` agree on an authentication (auth) file, which they use to mutually authenticate. The auth file will be shared between the `bank` and `atm` via a trusted channel unavailable to the attacker, and is used to set up secure communications.

Since the `atm` client is communicating with the `bank` server over the network, it is possible that there is a "man in the middle" that can observe and change the messages, or insert new messages. A "man in the middle" attacker can view all traffic transmitted between the two entities and may send messages to either the `atm` or the `bank`.

Note: For part b of this project (Break-It), the source code for `atm` and `bank` will be available to attackers, but not the auth file. The card file may be available in some cases, depending on the kind of attack. Make sure you design your protocol keeping in mind of attacks possible in this setup.

3 Basic Functionality

3.1 ATM

`atm` is a client program that simulates an `atm` by providing a mechanism for customers to interact with their bank accounts stored on the `bank` server. `atm` allows customers to create new accounts, deposit money, withdraw funds, and check their balances. In all cases, these functions are achieved via communications with the `bank` . `atm` cannot store any state or write to any files except the card-file. The card-file can be viewed as the "pin code" for one's account; there is one card file per account. Card files are created when `atm` is invoked with `-n` to create a new account; otherwise, card files are only read, and not modified. Any invocation of the `atm` which does not follow the four enumerated possibilities above should exit with return code 255 (printing nothing). Noncompliance includes a missing account or mode of operation and duplicated parameters. Note that parameters may be specified in any order.

3.1.1 Modes of Operation

The `atm` program can be used with multiple modes. Here are some examples:

```
atm -s <auth> -i <ip-addr> -p <port> -c <card> -a <account> -n
<bal> atm -s <auth> -i <ip-addr> -p <port> -c <card> -a
<account> -d <amount> atm -s <auth> -i <ip-addr> -p <port> -c
<card> -a <account> -w <amount> atm -s <auth> -i <ip-addr> -p
<port> -c <card> -a <account> -g
```

Each of the above 4 invocations uses one such mode; these are enumerated below.

- `-n <balance>` Create a new account with the given balance. The account must be unique (i.e., the account must not already exist). The balance must be greater than or equal to 10.00. The given card file must not already exist. If any of these conditions do not hold, `atm` exits with a return code of 255. On success, both `atm` and `bank` print the account and initial balance to standard output, encoded as JSON. The account name is a JSON string with key "account", and the initial balance is a JSON number with key "initial_balance" (Example: *"account":"55555","initial_balance":10.00*). In addition, `atm` creates the card file for the new account (think of this as like an auto-generated pin).
- `-d <amount>` Deposit the amount of money specified. The amount must be greater than 0.00. The specified account must exist, and the card file must be associated with the given account (i.e., it must be the same file produced by `atm` when the account was created). If any of these conditions do not hold, `atm` exits with a return code of 255. On success, both `atm` and `bank` print the account and deposit amount to standard output, encoded as JSON. The account name is a JSON string with key "account", and the deposit amount is a JSON number with key "deposit" (Example: *"account":"55555","deposit":20.00*).
- `-w <amount>` Withdraw the amount of money specified. The amount must be greater than 0.00, and the remaining balance must be nonnegative. The card file must be associated with the specified account (i.e., it must be the same file produced by `atm` when the account was created). The `atm` exits with a return code of 255 if any of these conditions are not true. On success, both `atm` and `bank` print the account and withdraw amount to standard output, encoded as JSON. The account name is a JSON string with key "account", and the withdraw amount is a JSON number with key "withdraw" (Example: *"account":"55555","withdraw":15.00*).

- `-g` Get the current balance of the account. The specified account must exist, and the card file
  must be associated with the account. Otherwise, `atm` exits with a return code of 255. On
  success, both `atm` and `bank` print the account and balance to stdout, encoded as JSON. The
  account name is a JSON string with key "account", and the balance is a JSON number with key
  "balance" (Example: *"account":"55555","balance":43.63*).

3.1.2 Parameters

The `atm` should support the following commands. Required parameters are labeled as such, the rest are optional.

- `-a <account>` The customer's account name. **(Required parameter)**
- `-s <auth-file>` The authentication file that `bank` creates for the `atm` . If -s is not specified, the default filename is "bank.auth" (in the current working directory). If the specified file cannot be opened or is invalid, the `atm` exits with a return code of 255.
- `-i <ip-address>` The IP address that `bank` is running on. The default value is "127.0.0.1".
- `-p <port>` The TCP port that `bank` is listening on. The default is 3000.
- `-c <card-file>` The customer's `atm` card file. The default value is the account name prepended to ".card" ("<account>.card"). For example, if the account name was 55555, the default card file is "55555.card".

3.2 Bank

`bank` is a server that simulates a `bank`, whose job is to keep track of the balance of its customers. It will receive communications from `atm` clients on the specified TCP port. Example interactions with `bank` and the `atm` are discussed in the Examples section. On startup, `bank` will generate a auth file with the specified name. Existing auth files are not valid for new runs of `bank` – if the specified file already exists, `bank` should exit with return code 255. Once the auth file is written completely, `bank` prints "created" (followed by a newline) to stdout. `bank` will not change the auth file once "created" has been printed. If an invalid command-line option is provided, the `bank` program should exit with return value 255. After startup, `bank` will wait to receive transaction requests from clients; these transactions and how the `bank` should respond are described in the `atm` specification. After every transaction, `bank` prints a JSON-encoded summary of the transaction to stdout, followed by a newline (this summary is also described in the `atm` spec). `bank` should bind to any host. The `bank` program will run and serve requests until it receives a SIGTERM signal, at which point it should exit cleanly. `bank` will continue running no matter what data its connected clients might send; i.e., invalid data from a client should not cause the server to exit and thereby deny access to other clients. The `bank` program will not write to any private files to keep state between multiple runs of the program.

3.2.1 Parameters

```
bank -p <port> -s <auth-file>
```

There are two optional parameters. They can appear in any order. Any invocation of the `bank` that does not follow the command-line specification outlined above should result only with the return code of 255 from the `bank` (invocations with duplicated or non-specified parameters are considered an error).

- **`-p <port>`** The port that `bank` should listen on. The default is 3000.
- **`-s <auth-file>`** The name of the auth file. If not supplied, defaults to "bank.auth".

4 Setup Details

This section described details about valid inputs and how to use the submission server. This project is setup on a separate server, https://bibifi.cs.umd.edu. To check the functionality of the components, you can use the Oracle setup on the server, which can use your query on your given build to make sure the your code functions as per requirements.

4.1 Oracle

Teams may query the oracle during the build-it round to clarify the expected behavior of the `atm` and `bank` programs. Queries are specified as a sequence of `atm` commands, submitted via the "Oracle submissions" link on the team participation page of the contest site. Here is an example query (this will make more sense if you read the specifications for the `atm` and `bank` programs first):

```
[
        {"input":["-p", "%PORT%", "-i", "%IP%", "-a", "ted", "-n", "10.30"],"base64":false},
        {"input":["-p", "%PORT%", "-i", "%IP%", "-a", "ted", "-d", "5.00"]},
        {"input":["-p", "%PORT%", "-i", "%IP%", "-a", "ted", "-g"]},
        {"input":["LXA=", "JVBPUlQl", "LWk=", "JUlQJQ==", "LWE=", "dGVk", "LWc="],"base64":true}
]
```

The oracle will provide outputs from the `atm` and `bank` . Here is an example for the previous query:

```
[
        {
                "bank": {
                        "output": {"initial_balance": 10.3,"account": "ted"}
                },
                "atm": {
                        "exit": 0,
                        "output": {"initial_balance": 10.3,"account": "ted"}
                }
        },
        {
                "bank": {
                        "output": {"deposit": 5,"account": "ted"
                        }
                },
                "atm": {
                        "exit": 0,
                        "output": {"deposit": 5,"account": "ted"}
                }
        },
        {
                "bank": {
                        "output": {"balance": 15.3,"account": "ted"}
                },
                "atm": {
                        "exit": 0,
                        "output": {"balance": 15.3,"account": "ted"}
                }
        },
        {
                "bank": {
                        "output": {"balance": 15.3,"account": "ted"}
                },
                "atm": {
                        "exit": 0,
                        "output": {"balance": 15.3,"account": "ted"}
                }
        }
]
```


In general, a test must begin with account creation, so that the `atm` program creates a card file, which can be used on subsequent interactions. Though we have not shown it, you can also pass the cardfile explicitly during a test, i.e., using the -c option. The `bank` will run at a dynamically chosen IP address, listening on a dynamically chosen port. On the local system, the default port is 3000 and default IP address is 127.0.0.1. You can test your implementation with the default IP address and port, although on the server this information cannot be known when writing the query. Make sure you write your implementation to sanitize inputs for IP address and port as well. For queries, you can specify these values using the following variables:

- IP - IP address of the `bank` server
- PORT - TCP port of the `bank` server

Each "input" is the argument list passed to `atm`. Queries may specify inputs using base64-encoded
values by by providing the optional boolean field "base64".

4.2 Valid Inputs

Any command-line input that is not valid according to the rules below should result with a return value of 255 from the invoked program and nothing should be output to stdout.

- Command line arguments must be POSIX compliant and each argument cannot exceed 4096
  characters (with additional restrictions below). In particular, this allows command arguments specified as "-i 4000" to be provided without the space as "-i4000" or with extra spaces as in "-i 4000". Arguments may appear in any order. You should not implement `--`, which is optional for
  POSIX compliance. You should implement guideline 5 (ex. `atm -garay` is valid).
- Numeric inputs are positive and provided in decimal without any leading 0's (should match /(0|[1-9][0-9]*)/). Thus "42" is a valid input number but the octal "052" or hexadecimal "0x2a" are not. Any reference to "number" below refers to this input specification.
- Balances and currency amounts are specified as a number indicating a whole amount and a fractional input separated by a period. The fractional input is in decimal and is always two digits and thus can include a leading 0 (should match /[0-9]2/). The interpretation of the fractional amount *v* is that of having value equal to *v*/100 of a whole amount (akin to cents and dollars in US currency). Command line input amounts are bounded from 0.00 to 4294967295.99 inclusively but an account may accrue any non-negative balance over multiple transactions.
- File names are restricted to underscores, hyphens, dots, digits, and lowercase alphabetical characters. File names are to be between 1 and 127 characters long. The special file names "." and ".." are not allowed.

- Account names are restricted to same characters as file names but they are inclusively between 1 and 122 characters of length, and "." and ".." are valid account names.
- IP addresses are restricted to IPv4 32-bit addresses and are provided on the command line in dotted decimal notation, i.e., four numbers between 0 and 255 separated by periods.
- Ports are specified as numbers between 1024 and 65535, inclusive.

4.3 Examples

Here is an example of how to use `atm` and `bank`. First, do some setup and run `bank`.

```
$ ./bank -s bank.auth
```

Now, create an account 'bob' with balance $1000.00.

```
$ ./atm -s bank.auth -c bob.card -a bob -n 1000.00
```

Here are some use cases for depositing money, withdrawing money, and creating a new account:

```
$ ./atm -c bob.card -a bob -d 100.00 $ ./atm -c bob.card -a bob
-w 100.00 $ ./atm -a alice -n 1500.00
```

5 Project Details

5.1 Code and Submission Requirements

This project must be coded in C. This project will not be submitted on the submit server used for previous projects. This will be submitted on the server: https://bibifi.cs.umd.edu. This project is hosted on the Build-it-Break-it-Fix-It server, which is a competition run every year by UMD CS.

5.2 Team Requirements

Form teams of **up to 3 members**. You can have teams of 1 or 2 members as well, but not more than 3 members.

- To run your build on this server, every student has to individually enroll on the server: https://bibifi.cs.umd.edu. On the home page, go to "Register" on the top-right corner. Remember your username and password.
- The team leader must create a team for the Fall 2020 contest. Go to the "Announcements" tab, and "Sign-up" (button on the right). Make your team by adding Team Name and Member's emails. (Note: The server might not mail each team member to join your team, hence make sure to create a Piazza post described in the next step).
- After enrolling, make a private post on Piazza with your Team name, participant user names and **gitlab** emails. This is required for making sure that everyone has registered on the submit server and providing access to gitlab repo to submit your code. TAs will create a private repo for your team on gitlab. After this is set up, your build will be tested on the server whenever you commit/push in your gitlab group repository. Starter code for build is provided. Note that you can modify/add files, but the code submitted must be in the build folder. If you intend to make sub-directories for `atm` and `bank`, do required changes in Makefile as well. Your submission will be scored after every push to the team gitlab repository. You can submit more queries on the website under oracle submissions, build submissions can be seen under build submissions.

5.3 Starter Code

Feel free to make any changes in the starter files as needed.

- `atm.c` has stub code available code for setting up communication between the `atm` and `bank`. With *atm_process_command*, you can define your security protocol function, by performing encryption-decryption as required, and use *atm_send* and *atm_recv* functions respectively.
- `atm.h` and `bank.h` are the header files including basic function definitions for `atm` and `bank`, respectively.
- `atm-main.c` currently has default input for port and IP address, you might not need to change that, as on the local system, you can test with default values. Make sure to sanitize inputs in this code so that user given inputs are validated before passing to `atm.c`. This is the