Ciências ULisboa

Segurança Aplicada

(Tecnologias de Segurança)

2024/2025

Project – ATM Communication

1. Goal

This project is inspired by the BIBIFI security programming competition (see https://builditbreakit.org/). The aim of the project is twofold. First, it allows students to set up a secure application scenario, namely a secure ATM protocol, while considering everything learned in previous security-related courses. Second, it lets students try to find real vulnerabilities in an application scenario that was produced by other groups.

2. Technical Details

2.1 Overview

In this project, you will design and implement an **ATM communication protocol**. There will be two programs. One program, called **`atm`**, will allow bank customers to withdraw and deposit money from their accounts. The other program, called **`bank`**, will run as a server that keeps track of customer balances. Specifications for these two programs and the security model are described in more detail below.

You will **build the most secure implementation you can in phase *Build it***; then, you will **have the opportunity to attack other groups' implementations in the *Break it* part**.

As the other groups can only effectively start to attack your code after they can run it, **you are responsible** for providing a copy of your code and instructions on how to compile it. **Failure to do so will penalize you in your grade!**

You will have to work in groups of **three people**. You can write your implementation in C, Java, or another language. You may also use any open-source libraries available on the Internet.

2.2 General Security Considerations

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

Since the ATM client is communicating with the bank server over the open network, a Man-In-The-Middle (MITM) can observe and change the messages or insert new messages. An *MITM* attacker can view all traffic transmitted between the **atm** and the **bank**. MITM may send messages to either the **atm** or the **bank**.

The source code for **atm** and **bank** will be available to attackers, but the ***auth file*** and the ***card file*** are usually kept secret (the ***auth file*** may be available in a specific kind of integrity attack).

2.3 Programs

The specifications for each program are listed below.

Bank server

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

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 at the specified TCP port. **bank** should bind to any host. Example interactions with **bank** and the **atm** are given later.

On startup, **bank** will generate an ***auth file*** with the specified name. Existing ***auth files*** are not valid for new **bank** runs -- if the specified file already exists, **bank** should exit with return code 255. Once the ***auth file*** is written entirely, **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 a return value 255.

After the startup, the **bank** will wait to receive client transaction requests; these transactions and how the bank should respond are described in the **atm** specification. After every transaction, the **bank** prints a JSON-encoded transaction summary to stdout, followed by a newline (this summary is also described in the **atm** spec).

The **bank** program will run and serve requests until it receives a SIGTERM signal, at which point it should exit cleanly. The **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 any private files to keep the state between multiple program runs**.

Command Line Options:

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

`-p <port>`: The port that the **bank** should listen to. The default is 3000.

`-s <auth-file>`: Name of the ***auth file***. If not supplied, defaults to `bank.auth`

ATM Client

```
atm [-s <auth-file>] [-i <ip-address>] [-p <port>]
     [-c <card-file>] -a <account> -n <balance>
atm [-s <auth-file>] [-i <ip-address>] [-p <port>]
     [-c <card-file>] -a <account> -d <amount>
atm [-s <auth-file>] [-i <ip-address>] [-p <port>]
     [-c <card-file>] -a <account> -w <amount>
atm [-s <auth-file>] [-i <ip-address>] [-p <port>]
     [-c <card-file>] -a <account> -g
```

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 the **atm** is invoked with `-n` to create a new account; otherwise, card files are only read and not modified.

Any invocation of the **atm** that 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.

Required Command Line Parameter:

`-a <account>`: The customer's account name (the format for the account is given later).

Optional Parameters:

`-s <auth-file>`: The authentication file that the **`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 255.

`-i <ip-address>`: The IP address that the **`bank`** is running on. The default value is `127.0.0.1`.

`-p <port>`: The TCP port that the **`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`.

Modes of Operation:

In addition to the account name, an invocation must provide a "mode of operation". 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, the **`atm`** exits with a return code of 255. On success, both the **`atm`** and the **`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, the **`atm`** creates the ***card file*** for the new account (think of this as 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 the **`atm`** when the account was created). If any of these conditions do not hold, the **`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 a key `account`, and the deposit amount is a JSON number with a 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 non-negative. The ***card file*** must be associated with the specified account (i.e., it must be the same file produced by the **`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 a key `account`, and the withdrawal amount is a JSON number with a 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, the **`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 a key `account`, and the balance is a JSON number with a key `balance`.
Example: `{"account":"55555","balance":43.63}`

2.4 Valid Inputs & Outputs

Inputs

Any command-line input that is invalid according to the rules below (and above) should result in a **return** value of 255 from the invoked program and **nothing** should be output to **stdout**.

- Command line arguments cannot exceed 4096 characters (with additional restrictions below). You should allow 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.

- 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 a **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 a value equal to v/100 of a whole amount (akin to cents and dollars in US currency). Command line input amounts are bound from 0.00 to 4294967295.99 inclusively, but an account may accrue any non-negative balance over multiple transactions (that can be represented in an 8-byte datatype).

- File names are restricted to underscores, hyphens, dots, digits, and lowercase alphabetical characters (each character should match /[_\\-\\.0-9a-z]/). File names are to be between 1 and 127 characters long. The special file names "." and ".." are not allowed.

- Account names are restricted to the 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 inclusively.

Outputs

- Anything printed to stderr will be ignored (e.g., so detailed error messages could be printed there, if desired).

- All JSON output is printed on a single line and is followed by a newline.

- JSON outputs must show numbers (including potentially unbounded account balances) with full precision.

- Newlines are '\n' -- the ASCII character with code decimal 10.

- Both programs should explicitly flush stdout after every line printed.

- Successful exits should return exit code 0.

Erros

>> Protocol Error

- If an error is detected in the protocol's communication, **`atm`** should exit with return code `63`, while the **`bank`** should print "`protocol_error`" to stdout (followed by a newline) and rollback (i.e., undo any changes made by) the current transaction.

- A timeout occurs if the other program does not respond within 10 seconds. If the **`atm`** observes the timeout, it should exit with return code `63`, while if the **`bank`** observes it, it should print "`protocol_error`" to stdout (followed by a newline) and rollback the current transaction. The non-observing party need not do anything in particular.

- If the **`atm`** cannot connect to the **`bank`**, it should exit with return code `63`.

>> **Other Errors**

- All other errors specified throughout this document, or unrecoverable errors not explicitly discussed, should prompt the program to exit with return code `255`.

2.5 Examples

Here is an example of how to use **`atm`** and **`bank`**.

First, do some setup and run the **`bank`**.

```
$ mkdir bankdir; mv bank bankdir/; cd bankdir/; ./bank -s
bank.auth &; cd ..
     created
```

Now set up the atm.

```
$ mkdir atmdir; cp bankdir/bank.auth atmdir/; mv atm
atmdir/; cd atmdir
```

Create an account 'bob' with a balance of 1000.00 (NOTE: There are two outputs because one is from the **`bank`** which is running in the same shell).

```
$ ./atm -s bank.auth -c bob.card -a bob -n 1000.00
     {"account":"bob","initial_balance":1000}
     {"account":"bob","initial_balance":1000}
```

Deposit 100.

```
$ ./atm -c bob.card -a bob -d 100.00
     {"account":"bob","deposit":100}
     {"account":"bob","deposit":100}
```

Withdraw 63.10.

```
$ ./atm -c bob.card -a bob -w 63.10
     {"account":"bob","withdraw":63.1}
     {"account":"bob","withdraw":63.1}
```

Attempt to withdraw 2000, which fails since 'bob' has insufficient balance.

```
$ ./atm -c bob.card -a bob -w 2000.00
$ echo $?
     255
```

Attempt to create another account 'bob', which fails since the account 'bob' already exists.

```
$ ./atm -a bob -n 2000.00
$ echo $?
     255
```

Create an account, 'alice', with a balance of 1500.

```
$ ./atm -a alice -n 1500.00
     {"account":"alice","initial_balance":1500}
     {"account":"alice","initial_balance":1500}
```

Bob attempts to access alice's balance with his card, which fails.

```
$ ./atm -a alice -c bob.card -g
$ echo $?
     255
```

2.6 Security Model

During the Break-It round, students will attack the implementations of the ATM protocol done by the other teams. The three types of attacks that can be performed may target correctness, integrity, and confidentiality violations. For some of these attacks, students will provide a program that runs as a man-in-the-middle (MITM). All tests require an input test file.

Correctness Violations

A correctness violation occurs when an implementation's behavior deviates from the above-described specification. To demonstrate a correctness violation, students will submit an input test file that contains a list of inputs to run on the **atm**. The professors may then run each input in the implementation to check the identified problem. If the outputs (including exit codes) differ between the expected and the implementation, a correctness violation has been successfully demonstrated. Correctness violations might also correspond to a crash (either in the **atm** or **bank**), which, when it happens in the **bank**, is considered a security problem because it violates the availability property.

The input test file is specified in a relatively simple manner; it consists of a sequence of argument lists to the **atm** command. Here is an example input test file for a correctness violation (where `"%PORT%"` and `"%IP%"` can have the actual values you used):

```
{
     "type": "correctness",
     "target_team": 9,
     "problem": "text explaining the problem, e.g., a crash
occurred in the atm",
```

```
     "inputs": [
          {"input":["-p",  "%PORT%",  "-i",  "%IP%",  "-a",
"ted", "-n", "10.3"]},
          {"input":["-p",  "%PORT%",  "-i",  "%IP%",  "-a",
"ted", "-d", "5.00"]},
          {"input":["-p",  "%PORT%",  "-i",  "%IP%",  "-a",
"ted", "-g"]}
     ]
}
```

Integrity Violations

An integrity violation occurs when an unprivileged attacker can **modify an account holder's balance**. To demonstrate an integrity violation, students will submit an input test file and an MITM program. The MITM program will intercept the traffic between the **`atm`** and the **`bank`**, and then it can forward it to the intended receiver or drop / delay / reorder it. The MITM program may also send requests to the **`atm`** or the **`bank`**. Details about the MITM program are provided below.

An integrity violation is demonstrated by running the implementation normally and the implementation+MITM. If both run all commands without error, but they differ on the balances of accounts at the bank (whose ***card files*** and ***auth files*** were not revealed to the MITM during the test) then the attack was successful.

Differing balances **do not count as a violation** in case of a timeout or protocol error detected by an **`atm`** or **`bank`**. In other words, only if differing outputs that are a result of **undetected** tampering by the MITM are considered integrity violations.

We will also consider a **weaker** form of integrity violation, where the attacker **is given access to the *auth file***. For example, the attack may be demonstrated by correctly guessing the contents of an account's card file and then performing the necessary steps to change the account balance as above.

As an example, imagine a test file that creates an account at the bank with 10 dollars in it, and then sends a message to deposit 50 more dollars. Note that the test file's type is `"integrity"`.

```
{
     "type": "integrity",
     "target_team": 9,
     "problem": "text explaining the problem",
     "inputs": [
          {"input":["-p",  "%PORT%",  "-i",  "%IP%",  "-a",
"ted", "-n", "10.00"]},
          {"input":["-p",  "%PORT%",  "-i",  "%IP%",  "-a",
"ted", "-d", "50.00"]},
```

```
        ]
}
```

Confidentiality Violations

A confidentiality violation occurs when an unprivileged attacker is able to uncover secret information. For our purposes, a secret is either the **name** of a non-public account, or an **amount** in a transaction or balance, or the contents of a ***card file***.

To demonstrate a confidentiality violation, breakers will submit an input test file and an MITM program. The MITM program will intercept all traffic between the **`atm`** and the **`bank`**. It can then modify, inject, drop, and delay the traffic.

At the start of a confidentiality test, one will generate two random values: an **amount** and an **account name**. If, by the end of the test, the attacker can prove it knows the actual value of either secret, it is considered a successful break.

The MITM does not have access to the ***card file*** and ***auth file***. Finally, if the **`atm`** or **`bank`** detects a timeout or protocol error at any time during the protocol run (e.g., due to tampering by the MITM), the attack is considered invalid.

Confidentiality violations can also be demonstrated by correctly guessing the contents of an account's ***card file***. These contents may be printed by the MITM program.

NOTE: If you managed to perform an attack that cannot be described with one of the above formats, then you need to explain the attack in detail in your report.

Man-in-the-middle

A man-in-the-middle (MITM) is a program that intercepts and potentially modifies or adds to communication between the **`atm`** and the **`bank`**. Confidentiality and integrity attacks use an MITM designed/built by the students. The MITM may conform to the specification given below.

```
mitm [-p <port>] [-s <server-ip-address>] [-q <server-port>]
```

MITM implements a man-in-the-middle used for the attacks. The MITM starts up playing the role of the **`bank`**, listening on the TCP port specified by `-p`. When an **`atm`** connects to it, the MITM connects to the real **`bank`**, which is assumed to be listening at the IP address and TCP port specified by `-s` and `-q`. With these connections established, MITM intercepts messages between the **`atm`** and the **`bank`** and can choose to drop them, forward them unchanged, forward them with modifications, or send completely new messages, perhaps based on previously seen messages.

The MITM program should run until it receives a SIGTERM signal, at which point it should exit cleanly. The MITM must be able to be run from any working directory.

Command Line Options

`-p <port>`: The TCP port that MITM should listen on to intercept communications. The default value is `4000`.

`-s <server-ip-address>`: The IP address that the **`bank`** is running on. The default value is `127.0.0.1`.

`-q <server-port>`: The TCP port that the **`bank`** is running on. The default value is `3000`.

2.7 Deliverables

Part 1: Build it

You should submit:

- A **design document** (PDF) in which you:
  - Describe your overall system design in sufficient detail for a reader to understand your approach without reading the source code directly. This must include a description of the protocol between the **`atm`** and the **`bank`**, the protections that were implemented, and the format of the messages. Other information that might be important to understand your solution should also be included, including an explanation of the ***auth file*** and ***card file***.
  - List four specific attacks you have considered and describe how your implementation counters these threats (Please indicate the relevant lines of code that implement your security controls). If you could not completely (or at all) prevent a threat you identified, you may still mention it. In that case, describe any partial mitigation you implemented and explain anything you wanted to implement but were, for whatever reason, unable to. Please be clear about distinguishing what you have implemented vs. what you *would have* implemented.

- Your **implementation**, including all your code files and your makefile.

Part 2: Break it

We will assign you randomly **two** groups' implementations to examine. You should submit:

- A **vulnerability analysis document** (PDF). For each of your assigned implementations and describe:
  - Any attacks you found, including a **high-level summary** and **enough detail for someone to replicate** the attack.
  - Any vulnerabilities you found but **were unable** to exploit for whatever reason.
  - If you find no attacks or vulnerabilities, describe **how you looked for attacks**.

- Also submit **any code you wrote to implement your attack**. Ensure the vulnerability analysis explains how to use your code to launch the attack. Also, include the **test files that demonstrate** a successful attack. Attacks can include crashes, integrity violations, and confidentiality violations.

Please note that while simple correctness bugs are relevant, they do not count for your vulnerability analysis -- for this analysis, you **must identify security-relevant issues**.

If one of your assigned implementations does not work well enough to analyze, please request an alternate implementation from the professor.

3. Timeline

Each group should finish the first phase of the project by the **7th of April at midnight**. An electronic copy of the report should be sent **before** that time to the email of your professor (nuno@di.fc.ul.pt), and a copy of the report and implementation should be saved in Moodle.

The second phase should be completed by the **12th of May at midnight**. Again, an electronic copy of the report should be sent **before** that time to your professor (nuno@di.fc.ul.pt), and a copy of the report and other deliverables should be saved in Moodle.

No extensions will be provided.

4. Grading

Students will be evaluated by the following criteria:

Part 1: Build it

- Your design document:
  - the description of your approach
  - each of the four specific vulnerabilities you defend against. For maximum points, your defense should be correct, fully implemented, and well-explained. Well-explained attacks that are not fully implemented will receive partial credit.
  - Extra credit will be permitted for one extra attack/countermeasure beyond the original four.

- The evaluation of the implementation will include performance tests, as well as correctness tests. Faster implementations (according to the performance tests) and optional features will receive bonus points.

Part 2: Break it

- A successful attack with a well-written vulnerability analysis will receive more credits.

- If you do not have a successful attack, then you must write a complete vulnerability analysis for both implementations you were assigned. This analysis must explain either why you could not implement the attack you identified or provide a convincing argument that there were no exploitable vulnerabilities.