Project Rubric

Cryptography and Network Security I

Bank - ATM Protocol

Overview:

This project involves designing and implementing a protocol for communicating between ATMs and banks. Once you have finished securing the design it will be given to another team for testing. You will meanwhile be given the opportunity to attack another team's protocol. Teams will be selected at random.

Protocol Design / Specification:

You will implement 3 programs to simulate a realistic Bank-ATM system: An ATM "client", a banking "server", and a proxy. They will communicate over TCP, and you can assume for simplicity's sake that they will all be running on the same host. The programs *must be written in C/C++*.

The ATM will connect to the Proxy, which in turn, will forward all data to the Bank. During development, the Proxy is passive and unobtrusive, it forwards data "as-is" without modifying or disturbing it in any way. During the testing/attack phase, you will be free to modify the proxy in any way you wish.

The ATM system requires users to have ATM cards, simulated by a file. ATM cards must have PINs associated with them.

ATM:

The ATM program is analogous to a physical ATM machine. It is the "front-end" to the banking infrastructure that a user sees. Requirements are listed below.

1. The ATM takes **one** command line argument: the **port number** on which it will connect to the **proxy.**

2. The ATM must make use of user's ATM cards, simulated by files named **"xxxx".card**. The format of these cards is up to you.
3. ATM PINs must be reasonable and roughly follow "real-life" guidelines. Typically this means PINs are numeric, and roughly 4-6 digits in length.
4. The ATM must include a command shell that implements the following commands:
    a. **login** [*username*] - Prompt for PIN of **<username>**; establishes session with the Bank.
    b. **balance** - Print's the current user's balance
    c. **withdraw** [*amount*] - Verifies sufficient funds, prints "[amount] withdrawn", and correctly modifies balance.
    d. **transfer** [*amount*] [*username*] - Transfers **<amount>** from the current user's account to **<username>'s** account. Function should perform verification of **<username>** as well as verifications on balance.
    e. **logout** - Cleanly terminates the current user's session with the bank.

Proxy:

The Proxy program is analogous to any of the networking nodes an ATM's data flows through in real life. It cannot be implicitly trusted. During development, it should **only** transport data to and from the bank. During the offensive stage, you can modify it to do anything you like to the data passing through it.

Requirements:

1. The Proxy takes **two** command line arguments. The **port** to **listen** on, and the **port** to connect to the bank.

<u>Offensive Stage Limitations:</u>

During the offensive stage, the proxy can **only communicate with the bank program, and not any other part of the system.** You must treat it as if it is running on a separate physical box.

For example, you **cannot** use the proxy to read the binary or address space of the ATM or Bank in order to leak cryptographic keys. Similarly, you cannot read user's ATM cards with the proxy from the filesystem.

However, you **can** use the proxy to attack a poor Bank or ATM application. For example, if there exists an information-leak vulnerability in the Bank, you can leverage it to leak crypto keys from memory. Similarly, you **can** use memory corruption vulnerabilities to attack the Bank or ATM if they exist.

The focus of the offensive stage is on cryptographic and protocol vulnerabilities, but you are free to leverage any bugs/implementation flaws you find towards that end.

Bank:

The Bank program is analogous to a real-life ATM system “back-end”. It handles communication with the ATMs in its “network”, as well as tracking user accounts and all relevant information that goes with that (sessions, balances, etc).

Requirements:

1. The bank takes **one** command line argument, the **port number** to listen on.
2. The bank must implement a command shell that implements the following functions
    a. **deposit** [*username*] [*amount*] - Increase **<username>’s** balance by **<amount>**
    b. **balance** [*username*] - Print the balance of **<username>**

Other Requirements / Logistics:

- Only one bank and proxy may exist at any given time. However, multiple ATMs can connect to the bank simultaneously.
- Three Users must be created at **bank initialization**:
    a. Alice : Balance of $100
    b. Bob : Balance of $50
    c. Eve : Balance of $0
- **You <u>cannot</u> use existing protocols such as SSH or SSL. You must design your own protocol.** Borrowing ideas from existing protocols is fine, and encouraged.
- Cryptographic primitives must come from a standard library. You **should not** be implementing your own versions of block ciphers or hashes. The focus of the project is protocol design, rather than cryptographic implementation.

Attacking Protocols:

An important part of this project is evaluating and attacking another team's protocol and implementation. Once the deadline for development comes, your team will be given another random team's code, protocol, and binaries for evaluation. During this period of time, you will be able to modify the proxy server in any way you wish, **but only the proxy server.**

Some notes about the attack phase of this project:

- Modification/patching/tampering with the ATM and Bank binaries is not allowed.
- Reverse Engineering ATM/Bank Binaries to recover cryptographic material is not allowed. **However** reverse engineering for the purpose of identifying vulnerabilities **is allowed and encouraged, but not required.**
- You **may** freely modify the proxy to tamper with, drop, duplicate, modify, or create messages going in either direction (Bank - ATM, ATM - Bank).
- All protocol level attacks are allowed, including but not limited to:
    a. Memory Corruption (Buffer Overflows, Use-After-Free, etc)
    b. Integer Bugs (Boundary, ”Rollover”, etc)
    c. Cryptographic Weaknesses
    d. “Session” bugs (replay attacks, etc)
- You **must** submit a full evaluation of your adversary's protocol and implementation. This must discuss all vulnerabilities and attacks in sufficient detail such that they can be reimplemented. Submitting code that demonstrates the weakness is also highly encouraged. If you cannot find any weaknesses, describe the attacks you tried in detail, and describe why the implementation/protocol is secure against them.
- Bonus Points will be awarded for especially clever or devastating attacks.

Submission Requirements:

Naturally, you will end up with a large codebase that has unique dependencies. To streamline the attack phase, as well as make it easier for us to grade fairly, you must adhere to the following requirements.

1. Your code must work on a clean install of Ubuntu 14.04 32-bit.
2. You must submit a build script that will setup any external libraries you need and cleanly compile your code.
3. You **must** submit source code, along with compiled binaries. You **may not** obfuscate source code; it must be readable C/C++.
4. You **must** submit a full description/specification of your protocol in addition to your code. The format is up to you, but it must be detailed enough such that another person could implement a program that is compatible with your system.
5. You **must** submit a description of all vulnerabilities found in your assigned teams code. This includes vulnerabilities you found, but could not exploit for some reason. Code should be included for attacks that require it.

Due Dates:

- **To Be Announced**

Grading Policy:

Your grade for the project will be divided into two parts, weighted evenly. Each team-member will receive the same score barring extreme circumstances.

The **defensive** part of your grade will be based on an evaluation of your protocol's security by the professor and TA, as well as by how well it survives attack by your classmates.

The **offensive** part of your grade will be based on how effective you are at breaking the opposing team's protocol. If you are unable to find any exploitable vulnerabilities, submit a report describing in detail every attack that you considered and why their protocol is immune to it. If you are correct in your assessment, you **will not** lose points for being unable to break the protocol.