Lab Assignment 1
DD2525

Alexandru Matcov[1] and Jakob Ström[2]

[1]matcov@kth.se
[2]jakstrom@kth.se

April 2025

1 Implementation & Approach

The following sections outline how we approached the implementation of each necessary component of the dating service. The order in which each part was implemented follows the ordering of these sections.

1.1 Server

Firstly, the server had to be implemented. We observed the following two key requirements of the server functionality which were the focus of the implementation:

- The server needs to be able to receive, process and store the clients and their profiles
- The server needs to be able to check if profiles are a match, and send the respective profiles to the matching clients

This resulted in the server function and the checkMatch function respectively. The server recursively handles new profiles with the *receive* construct and tries to match them with the existing profiles.

For the matching, the server uses the checkMatch function which is where the declassification operations occur for the server. Firstly, each agent is run on the person's profile within a *pini* block, to be able to determine if there is successful match result. This result, as well as the profiles, are then declassified as to be able to make a control flow decision, as well as be able to send the profiles. If there is a match, then the match notifications are sent to the appropriate clients.

This functionality combined with contacting a dispatcher for some initial clients is what constitutes our server logic.

1.2 Benign Clients

After finalizing the server, we moved on to the benign clients. We observed the following key aspects of what is needed of the client functionality:

- The client needs to send their profile data, their agent and their process identifier to the server
- The client to be able to receive possible matches
- The agent needs to determine if the other persons profile matches the user's preferences

The implementation of the first and second aspects were relatively straightforward, with it coming down to using the *send* expression with the respective data for the first aspect, and setting up a *receive*-loop waiting for new matches for the second.

The agent however, was slightly more complex. Aside from the evaluation of the interests, age, etc., we also had to work with the labels. Most profile entries are raised to the security level of its own user, which caused some issues when trying to send the profiles to other people. We needed to remove the security label of the user, and add the label of the recipient. After some testing, using *declassifydeep* with the labels of the other profile our client matched with. This helped assign the other profile's security label to all the nested data, more specifically client's interests, and have the *maybeProfile* ready to be sent to the other client it matched with

1.3 Malicious Client

Regarding the malicious client, numerous ideas on how to leak sensitive data were discussed. After some trial and error, we landed on implementing an intercepting agent which the malicious client sends to the dating server.

The agent function of the malicious client was to act similarily to the receive function of the server, except processing the profiles and leaking data to the malicious client, instead of to the server. The implementation is based on a *receive* and *send* functions implemented by the agent and deployed on the server. The character of the sensitive data is leaking the number of unique clients connecting to the server by counting the and sending the *lev* of each intercepted profile.

To demonstrate an active privacy breach we equip a "malicious" client with an *intercepting agent*: after the dating server spawns this agent it silently listens for `NEWPROFILE` messages from *all* other clients. Whenever a new profile arrives the agent extracts the profile's security–level tag (`lev`) and leaks it back to the attacker's *own* client process under the label `("LEAKED", lev)`. The attack therefore reveals how many distinct principals (and which tag names) appear on the server without crashing the server or violating the syntactic protocol. The two short Troupe fragments below implement the passive leak-receiver on the client side (Listing 1) and the active interceptor that runs on the server for every new client (Listing 2).

```
(*  Malicious client: print every leaked security label it receives. *)
fun loop () =
  let
    val _      = printString "Waiting for leaked levels..."
    val lev    = receive [ hn ("LEAKED", tag) => tag ]
    val _      = printWithLabels lev          (* e.g. {alice} or {bob} *)
  in  loop () end
```

Listing 1: Client side loop: wait for leaked levels from the malicious agent

```
(*  Agent code shipped by the malicious client; executed on the server. *)
fun maliciousAgent profileOther =
  let
    (* Sniff the next NEWPROFILE sent by any client. *)
    val intercepted             =
        receive [ hn ("NEWPROFILE", data) => data ]

    val (victimProfile, _, _)   = intercepted
    val (lev, _, _, _, _)       = victimProfile   (* steal the tag *)

    val _ = debugpc()           (* optional: see PC / BL for demo *)
    val _ = send (malicious_id, ("LEAKED", lev)) (* leak tag *)
  in
    (false, ())                 (* always reject => no visible match *)
  end
```

Listing 2: Server-side component embedded in `maliciousAgent`

The server and malicious client logs for the malicious implementation can be checked in the Appendix.

2 Contributions

Alexandru Matcov implemented the server, both the benign and a malicious clients' functionality, and helped summarizing the report.

Jakob Ström assisted in implementing both benign and the malicious clients, and summarized the results for this report.

3 Appendix

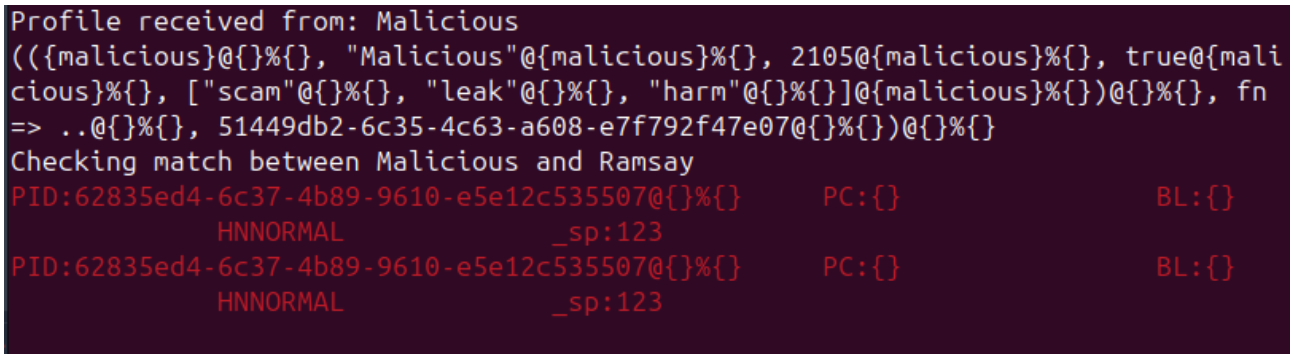

Figure 1: Server log: the malicious agent intercepts `NEWPROFILE` messages and forwards each victim's `lev`.

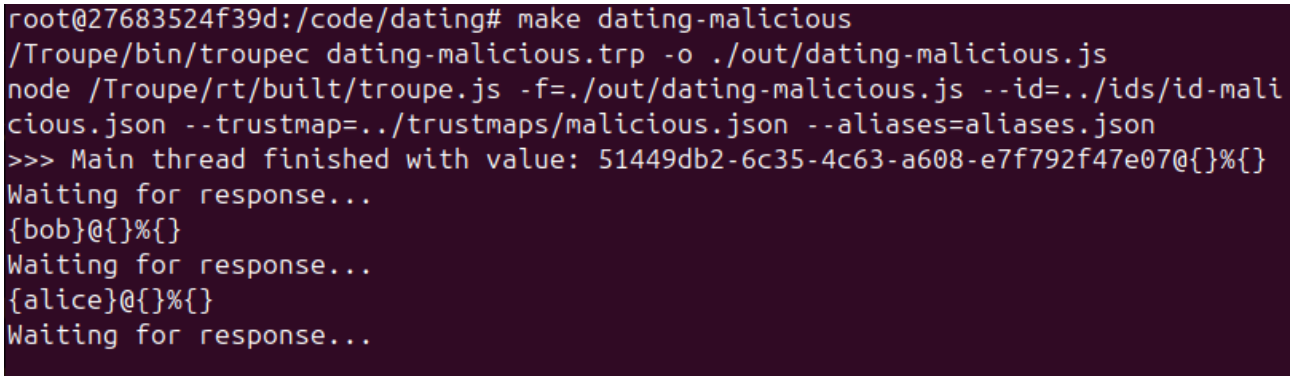

Figure 2: Malicious client log printing every leaked tag ({`alice`}, {`bob`}, ...).