Lab 1: Information Flow Control, DD2525

Stefan Garrido, Jens Ekenblad

April 18, 2023

Implementation

The following sections describes the details regarding how the assignment was approached in terms of preparations, design, execution, and reasoning in regards to security labels and declassification. Additionally, a section on malicious client is presented. Lastly, a short section is provided that states the contributions of each group member.

Approach & Design

None of the group members had any experience with the Troupe language which meant that preparations was key. As such, both members made sure to attend the necessary lab sessions, watch the Troupe tutorial, read through the Troupe manual, and read through the assignment description. Additionally, both members decided to work on all parts of the assignment together in order to have equal contributions. This meant meeting in person or using VS Code live share when working remotely.

The main approach for implementing the assignment was to utilize the given starting code, use the Troupe examples to better understand the syntax and layout of the language, and finally consult the manual when clarification was needed on certain functions or syntax.

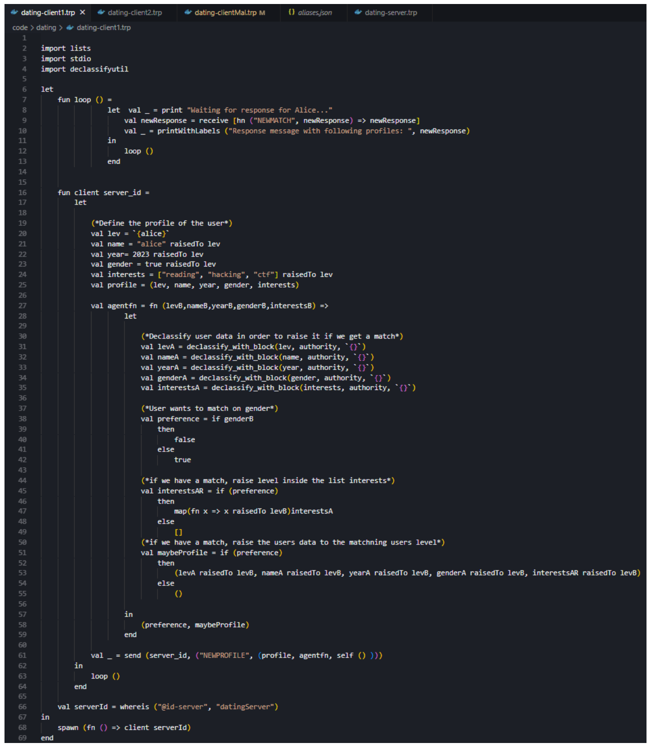
Looking at the design of the server it was set up as shown in figure 1. The main idea was to let the server run in a loop state to constantly check its mailbox for incoming messages from other processes. The server uses a handler to check its mailbox for pattern matching on the string "NEWPROFILE". Once a new profile is received the server adds it to a list that acted as a database for the dating server. The list is then passed into the matching function *match* to check the new profile against the previously stored profiles.

The function takes two user tuples consisting of their respectively $(profile, agent, pid)$ where the function lets each user's agent to check if there is a match on the other profiles. If both user's agents returns *true* then the server sends back messages to the two user processes with their respective matching profiles.

Each client was then set up with a similar approach to each other but with its own uniquely defined profile and agent function as showed in figure 2. Two clients was set up as *Alice* and *Bob* with their agents designed with different restrictive level on their matching criteria. The fundamental part of each client was to let it send its tuple $(profile, agent, pid)$ to the server and then enter a looping state where it let a handler checking the mailbox for incoming messages with pattern matching on the string "NEWMATCH".


```
(* Starting file for the server *)

import lists
import declassifyutil
let

    fun match user1 user2  =
        let
            val (profile1, agent1, pid1) = user1
            val (lev1, name1, year1, gender1, interests1) = profile1
            val (profile2, agent2, pid2) = user2
            val (lev2, name2, year2, gender2, interests2) = profile2

            val _ = printWithLabels ("Comparing names:", name1, name2)

            (* pinipush momenterly takes away BL labels in order to handle values preference and maybeProfile*)
            val tmp = pinipush authority (*declassify label BL from here...*)
            val (preference1, maybeProfile1) = agent1(profile2)
            val (preference2, maybeProfile2) = agent2(profile1)
            val _ = pinipop tmp (*...to here*)


            val _ = printWithLabels("Preference1: ", preference1)
            val _ = printWithLabels("Preference2: ", preference2)
            val _ = printWithLabels("Maybeprofile1: ", maybeProfile1)
            val _ = printWithLabels("Maybeprofile2: ", maybeProfile2)

            (*Declassify the boolean variables preferences and their blocking labels in order to check a match*)
            val _ = if declassify_with_block(preference1 andalso preference2, authority,`{}`)
                    then let
                        val _ = printWithLabels ("It's a match!")
                        val _ = send (pid1, ("NEWMATCH", maybeProfile2))
                        val _ = send (pid2, ("NEWMATCH", maybeProfile1))
                    in
                        ()
                    end
                    else let
                        val _ = printWithLabels ("No match!")
                    in
                        ()
                    end
        in
            ()
        end


    fun server db =
        let
            val data = receive [hn ("NEWPROFILE", data)
                                 => printString "New profile received"; data
                               ]
            val _ = map (match data) db
        in
            server (data::db)
        end


    (* Our main function starts the server and then requests the
       dispatcher to send some clients this way. *)
    fun main () =
        let
            val thisNode = node (self ())
            val _ = printString ("Running node with identifier: " ^ thisNode)
            val serverId = spawn (fn () => server [])
            val _ =  register ("datingServer", serverId, authority)
        in  (* TODO: Feel free to comment out the next line
                  while you develop your solution and work on a few
                  custom clients;  *)
           (* send (whereis ("@dispatcher", "dispatcher"), ("DISPATCH", thisNode)); *)
            ()
        end
in
    main ()
end
```

Figure 1: Implementation of the server.



```
import lists
import stdio
import declassifyutil

let
    fun loop () =
                let  val _ = print "Waiting for response for Alice..."
                    val newResponse = receive [hn ("NEWMATCH", newResponse) => newResponse]
                    val _ = printWithLabels ("Response message with following profiles: ", newResponse)
                in
                    loop ()
                end


    fun client server_id =
        let

            (*Define the profile of the user*)
            val lev = `{alice}`
            val name = "alice" raisedTo lev
            val year= 2023 raisedTo lev
            val gender = true raisedTo lev
            val interests = ["reading", "hacking", "ctf"] raisedTo lev
            val profile = (lev, name, year, gender, interests)

            val agentfn = fn (levB,nameB,yearB,genderB,interestsB) =>
                    let

                        (*Declassify user data in order to raise it if we get a match*)
                        val levA = declassify_with_block(lev, authority, `{}`)
                        val nameA = declassify_with_block(name, authority, `{}`)
                        val yearA = declassify_with_block(year, authority, `{}`)
                        val genderA = declassify_with_block(gender, authority, `{}`)
                        val interestsA = declassify_with_block(interests, authority, `{}`)

                        (*User wants to match on gender*)
                        val preference = if genderB
                            then
                                false
                            else
                                true

                        (*if we have a match, raise level inside the list interests*)
                        val interestsAR = if (preference)
                            then
                                map(fn x => x raisedTo levB)interestsA
                            else
                                []
                        (*if we have a match, raise the users data to the matchning users level*)
                        val maybeProfile = if (preference)
                            then
                                (levA raisedTo levB, nameA raisedTo levB, yearA raisedTo levB, genderA raisedTo levB, interestsAR raisedTo levB)
                            else
                                ()

                    in
                        (preference, maybeProfile)
                    end

            val _ = send (server_id, ("NEWPROFILE", (profile, agentfn, self () )))
        in
            loop ()
        end

    val serverId = whereis ("@id-server", "datingServer")
in
    spawn (fn () => client serverId)
end
```

Figure 2: Implementation of the client 'Alice'.

Security labels and declassification

A major part of the Troupe language is that it implements security by restricting the confidentiality of data using labels. This means that nodes can choose how much of their data is restricted by labeling it with security level (e.g '{alice}', or '{alice,bob}'). Therefore, while implementing the server certain restrictions occurs when accessing users data. In order to let the server and clients work as intended choices regarding declassification of security labels have to be made on both sides.

As previously mentioned, the server compares two user profiles at a time by calling the users agent functions. As a consequence, the blocking level is raised to the corresponding profile level due to the data in that profile being confidential. Since the blocking level never decreases, the server fails when a new comparison is performed. Therefore, the server implements *pinipush authority* and *pinipop* which temporally removes the blocking level labels when the user agents are called. Moreover, the user agents returns a tuple $(preference, maybeProfile)$ consisting of a boolean and the users profile raised to the argument profiles level. A conditional is used to check the users preferences, and if both are true, the matching profiles are sent back to the users. However, since the preferences are both raised to the different profiles levels the server needs to declassifies these. This is achieved by calling *declassify_with_block* when entering the conditional, which also completely removes the blocking levels in order to avoid the previously mentioned problem with blocking levels. The declassification is done on variables that we feel comfortable revealing, which in turn allows the server to relay back the information to its users.

In order for each client to be able to successfully retrieve a matching profile, the security level on all defined profile variables had to be first declassified from its own user level and then raised to the matching profile's security level. The function *declassify_with_block* was used on each variable as it not only removes the defined security level but also declassifies the blocking label that is triggered when the other user profile is passed as arguments to the agent function. Then each users profile is raised to the matching profile's security level in order to make it ready to be passed by the server to the recipient process.

Malicious client

The purpose of the malicious client is to reveal any confidential information that belongs to other users. In our case, the malicious clients can reveal if the user is either male or female by sending a *sleep* function to the server which is triggered depending on the variable that is checked. Two malicious that co-operates are implemented, where the first one trigger the server to go to sleep while the other malicious client checks how long time it takes until it gets a match with the first malicious client. As seen in figure 3, when "sleep" is triggered it indicates that the victim is a female, the elapsed time for the second malicious client to get a response is significantly longer than compared to when the victim is a male which is illustrated by the red and green circles. The client could also be modified to check the users age, however, this is not implemented in the solution.

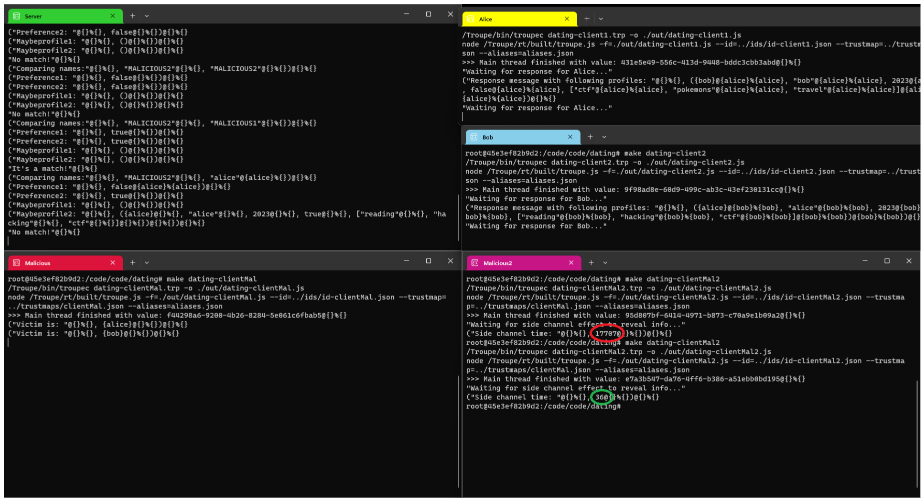


Figure 3: Time information leaked.

Contribution of group members

Both group members decided from the beginning to meet up online or in person and work on the assignment together. This made sure that both members had an equal opportunity to contribute to the assignment and learn about the given topic. As a result, both members worked on the server, clients, and malicious client implementations.

For the server, both members implemented the main function $match(user1, user2)$ and the overall modifications from the starting code in order to make the server work. As for the client implementations, the code is based on the client example found in $exercise - Troupe$. Both members implemented the main function $client(server\_id)$ and the overall modifications from the starting code in order to make the clients talk to the server. Additionally, both members worked on different solutions for the agent function needed for the assignment, which ultimately ended in mixing these into was is found in the submitted solution.

Lastly, an attempt was made from both members to successfully implement a malicious client. The submitted solution for the malicious client was mainly developed by member Ekenblad and assisted by member Garrido. The final contribution ratio was estimated to be 70/30.