ISA Projekt

Reverse-engineering neznámého protokolu (Ing. Koutenský)

Manuál

Ondřej Sloup (xsloup02)

Obsah

[O projektu 3](#_Toc86507468)

[Postup řešení 3](#_Toc86507469)

[Wireshark disektor 3](#_Toc86507470)

[Volba prostředí 3](#_Toc86507471)

[Použité zdroje 3](#_Toc86507472)

[Popis disektoru 4](#_Toc86507473)

[Vytváření klienta 5](#_Toc86507474)

[Volba prostředí 5](#_Toc86507475)

[Použité knihovny 5](#_Toc86507476)

[Použité zdroje 5](#_Toc86507477)

[Popis programu 6](#_Toc86507478)

[Popis vypracování 6](#_Toc86507479)

[Třídy 7](#_Toc86507480)

[Funkce 7](#_Toc86507481)

[Problémy s implementací 9](#_Toc86507482)

[Návratové kódy 9](#_Toc86507483)

[Testování 10](#_Toc86507484)

[Wireshark disektor 10](#_Toc86507485)

[Příklady spuštění 10](#_Toc86507486)

[Klient 18](#_Toc86507487)

[Příklady spuštění 18](#_Toc86507488)

[Reference 21](#_Toc86507489)

# O projektu

Cílem tohoto projektu bylo vytvořit disektor pro Wireshark v jazyku Lua a   
implementovat kompatibilního klienta pro připojení na poskytnutý server. Úkol projektu byl shrnut do následujících bodů:

1. Zachytit komunikaci mezi poskytnutým serverem a klientem
2. Implementovat vlastní Wireshark disektor pro daný protokol
3. Implementovat kompatibilního klienta pro daný protokol

# Postup řešení

Prvním krokem v mém řešení bylo zprovoznit virtuální stroj na kterém je funkční implementace klienta i serveru a zjistit, jak vypadají posílané packety. Zjistil jsem, že se jedná o výměnu informací na TCP spojení podobné emailu, kde klient se registruje, přihlašuje a posílá zprávy jiným uživatelům.

Použil jsem program Wireshark pro sledování a zachycení komunikace. Na server jsem skrz klienta poslal všechny dostupné příkazy a celou komunikaci jsem si uložil jako „.pcap“ soubor. Na této zachycené komunikaci jsem se rozhodl vytvořit disektor, který mě více seznámil s klientem, který budu implementovat v následujícím kroku.

# Wireshark disektor

## Volba prostředí

Rozhodl jsem se psát disektor v jazyku Lua, kvůli rozsáhlým příkladům dostupných na oficiálních stránkách Wiresharku a zmíněným odkazům přímo v zadání – (1), (2), (3). Ačkoliv jsem neměl zkušenosti, jak s psaním disektorů, ani v psaním kódu v jazyku Lua, myslím, že jsem udělal správně rozhodnutí, jelikož zpracování této části projektu proběhlo poměrně hladce.

## Použité zdroje

Před začátkem psaní kódu jsem si přečetl všechny dostupné příklady z oficiální dokumentace Wiresharku (1), (2), (3) a našel jsem YouTube video, které popisuje podobný příklad disektoru, který jsem implementoval (YouTube video – (4) a jeho zdrojový kód – (5)). Díky těmto zdrojům jsem byl schopný pochopit, jak script vytvořit a jaké jsou správné návrhové vzory.

## Popis disektoru

Skript je rozdělen na 3 funkce – parseMessage(), LaunchISAMaildissector() a Main().

Main() je hlavní funkce, které je spuštěna na začátku scriptu a jejím jediným úkolem je spustit funkci LaunchISAMaildissector().

Funkce LaunchISAMaildissector() spouští celý proces disektoru a definuje základní části pro rozpoznání protokolu. Nejprve definuje známé hodnoty jako jméno protokolu, proměnné označují úspěch a neúspěch odpovědi od serveru a možné výstupy do stromu Wiresharku. Dále je spuštěna vnitřní funkce isamailproto.dissector(), která již analyzuje zaslaná data.  
Následuje strom podmínek, které rozpoznají, zda je správa od klienta či od serveru a zda je úspěšná. Jednotlivé zprávy jsou poté zařazeny a zpracovány tak, aby byly čitelné pro program Wireshark, a i pro jeho uživatele. Pro rozpoznání jednotlivých příkazů jsem použil pokročilé “Lua patterns“, které mi usnadnili mapování zaslaných dat a rozpoznávání typů zprávy.

Jelikož se jedná o TCP protokol musel jsem implementovat znovu sestavení protokolu, pokud data přesáhnou délku 4096 bajtů. Pokud toto skript zjistí zavolá vnitřní funkci „DESEGMENT\_ONE\_MORE\_SEGMENT“, která se postará o to, aby skript dostal všechny data. Dále jsem implementoval zkontrolování délky packetu (7 bajtů), což odpovídá nejméně možné velikosti dat v packetu – prázdný list – „(ok ())”. Při neúspěchu se packet nerozpoznává a je nezměněn skriptem.

Celý Wireshark disektor je namapován na stabilní port „32323“, který je nastaven jako výchozí port na klientu. Rozhodl jsem se nerozpoznávat protokol heuristicky podle dat z důvodu možné kolize s ostatními, podobně vypadajícími protokoly, kde bych nemohl zaručit správnost. Uživatel toto nastavení bude muset mít na paměti.

Vše je popsáno a okomentováno ve zdrojovém kódu.

# Vytváření klienta

## Volba prostředí

Rozhodl jsem se plně použít veškerých výhod jazyka C++ a vypracovat celý projekt v něm. Chtěl jsem tento projekt takto zpracovat hlavně z toho důvodu, že to byl můj první projekt v jazyku C++ a díky disektoru, který jsem poměrně rychle implementoval jsem měl čas si promyslet celou strukturu a návrh klienta. Server pro klienta jsem si dokázal zprovoznit na svém počítači ve WSL2, což značně zjednodušilo debugování aplikace a její vypracování.

## Použité knihovny

Knihovny, které jsem použil jsou ze standartní knihovny. Jmenovitě jsou to tyto:

* sys/socket.h – Knihovna pro vytvoření soketu a definici všech struktur, které jsou potřebné pro navázaní spojení se serverem
* unistd.h – Knihovna pro deklaraci struktur a typů, které usnadní práci s připojením
* arpa/inet.h – Knihovna pro deklaraci portů a IP adresy
* netdb.h – Knihovna pro deklaraci funkcí a typů potřebné k navázaní kontaktu
* getopt.h – Knihovna pro zpracování argumentů dle Unixu
* regex – Knihovna pro rozpoznání textových řetězců regulárním výrazem. Je použit pouze pro verifikaci packetů
* iostream – Knihovna pro standartní vstup a výstup
* map – Knihovna pro slovníky
* vector – Knihovna pro „pokročilá pole”
* stdio.h, stdlib.h – Knihovny pro makra a pro typy proměnných
* string.h – Knihovna pro definici textových řetězců
* bitset – Knihovna pro operace s bity používaná pro base64 konverzi
* fstream – Knihovna pro práci se soubory používaná pro vytváření souborů s tokenem

## Použité zdroje

Při navrhování klienta jsem vycházel ze základní implementace TCP klienta, kterou jsem našel na internetu (6) a dále jsem ji přizpůsoboval svým potřebám. Velkou pomoc jsem našel na stránce (7), když jsem řešil rozpoznávání domén a IPv6. Řešil jsem problém s funkcemi gethostbyname(), která nepodporuje IPv6, gethostbyname2(), která podporuje IPv6, ale je složité ji implementovat a getaddrinfo(), kterou používám v projektu. (viz Zjišťování domén a IPv6)

## Popis programu

Aplikace klient navazuje TCP spojení na server, aby na něm mohl vykonat předdefinované příkazy. Uživatel může specifikovat adresu i port, ale aplikace má i své výchozí nastavení (adresa “localhost” a port “32323”). Dále uživatel specifikuje jeden ze 6 příkazů, který provede danou akci:

1. register – registruje uživatele za pomocí specifikovaného jména a hesla. Heslo je převedeno do base64 podoby před odesláním na server
2. login – Přihlašuje uživatele za pomoci jeho jména a hesla, které je stejně jako v případě registrace převedeno do base64. Vytváří login-token.
3. list – Nemá žádné další parametry. Vypisuje všechny zprávy, které dostal přihlášený uživatel
4. fetch – Umožňuje zobrazit uživateli zprávu pomocí její ID. Uživatel toto ID musí zadat. ID odpovídá pořadí zpráv v listu. Je dostupný pouze přihlášenému uživateli.
5. send – Uživatel musí zadat komu je zpráva poslána, s jakým předmětem a samotný text zprávy. Je dostupný pouze přihlášenému uživateli a pouze přihlášený uživatel si může zprávu přečíst.
6. logout – Maže login-token a tím odhlašuje uživatele z klienta.

## Popis vypracování

Aplikace je rozdělená na několik souborů, kde každý zastává určitou část.

* main.cpp – Vstupní část programu, která spouští jednotlivé části programu
* client.cpp/hpp – Zpracovává vstupní parametry a vytváří soket, který je použit připojení na server
* handler.cpp/hpp – Na základě předaných vstupních parametrů vytvoří třídu, která ze zpracovaných příkazů vytvoří textové řetězce, který je odeslán na server. Poté zpravuje příchozí zprávu ze serveru.
* requestmsg.cpp/hpp – Soubor obsahující jednotlivé třídy, které zpracovávají zprávy ze serveru a vytváří odesílané zprávy
* errorcodes.hpp – Seznam chybových kódů

### Třídy

V projektu jsem vytvořil dvě základní třídy:

* RequestHandler – Třída, která rozpoznává, která třída se má vytvořit podle zadaného parametru
* RequestMsg – Abstraktní třída, která má pod sebou dalších šest tříd odpovídající každému příkazu, který uživatel může zadat. Jednotlivé třídy mají stejné abstraktní funkce, které přímo zpracují zprávu, která se odešle na server.

### Funkce

#### main.cpp

* int main() – vstupní funkce projektu, která spouští všechny funkce tohoto programu

#### client.cpp/hpp

* printHelpMessage() – Funkce obsahující textový řetězec, který je vypsán, pokud uživatel se snaží vypsat pomocnou zprávu
* parseArgs() – Funkce, která se stará o pasování argumentů pomocí getopt a ověřuje jejich platnost
* createSocketAndConnect() – Funkce, která vytváří soket na kterém bude vytvořeno spojení. Zde jsem použil tento zdroj (7).

#### handler.cpp/hpp

Tento soubor obsahuje pouze zmíněnou třídu RequestHandler()

* buildClientString() – Funkce pro vytvoření textového řetězce, který bude zaslán serveru. Funkce nejdříve zjistí podle definované mapy, pod kterou třídu příkaz spadá
* exchangeData() – Funkce, která se pomocí vytvořeného soketu připojuje na server. Dokáže rozpoznat doménová jména a umí IPv4 i IPv6.

#### requestmsg.cpp/hpp

* třída RequestMsg – abstraktní třída, která zpracovává obsah příchozích a odchozích zpráv. Obsahuje definici funkcí, které pomáhají zpracovat zprávy serveru. Dále obsahuje informace o počtu argumentů a název příkazů
  + buildString() – abstraktní funkce, které je děděná všemi příkazovými třídami. Vytváří textový řetězce a zajištuje veškeré funkce, které jsou potřebné pro jeho vytvoření.
  + handleOutput() – abstraktní funkce, které je děděná všemi příkazovými třídami. Zpracuje textový řetězec, který je přijat od serveru a zavolá všechny potřebné funkce.
  + getError() – Vypíše specifickou chybovou hlášku, pokud je příkaz zadán se špatnými požadavky
  + getNumArg() – Navrátí počet argumentů potřebných pro zavolání příkazu
  + toBase64() – Konvertuje heslo do base64 podoby
  + getToken() – Navrátí token z uloženého souboru, který je potřeba pro interakci se serverem při přihlášeném uživateli
  + createToken() – Vytvoří token a uloží ho do souboru pro pozdější použití
  + removeToken() – Smaže soubor při odhlášení uživatele
  + resultParse() – Zjistí, zda data v packetu jsou ve správném formátu a jestli je možné je dále zpracovávat
  + printResult() – Vypíše na standartní výstup z vytvořený textový řetězec s hlavičkou “ERROR” nebo “SUCCESS”.
  + escapeChars() – Zajistí, aby znaky „\n”, „\t”, „\”, „““ byly escapované při posílání na server
  + unescapeChars() – Zajistí, aby znaky „\n”, „\t”, „\”, „““ byly převedeny při vypisování na standartní výstup
  + isNumber() – Zjistí, jestli daný textový řetězec je číslo
  + splitByRegex() – Funkce, jejíž podobnou verzi používám i v disektoru, zpracovává příchozí zprávy podle vytvořeného konečného automatu, který zjišťuje podle uvozovek, o jakou část zprávy se jedná
  + třídy Register, Login, List, Fetch, Send, Logout – obsahují abstraktní funkce buildString(), handleOutput() a getError(), které jsou upraveny specificky pro jejich potřeby

### Problémy s implementací

#### Zjišťování domén a IPv6

Při vypracovávání této části projektu jsem se snažil navrhnout klienta, tak aby podporoval vše jako referenční klient, a to IPv6 podporu a podporu doménových jmen. Na fóru jsem narazil na funkci gethostbyname() z knihovny netdb.h, kterou jsem poměrně rychle a jednoduše naimplementoval. V průběhu řešení jsem ale zjistil, že funkce nemá podporu pro IPv6, a proto jsem začal hledat další ekvivalenty a jako jeden z možných se naskytla funkce gethostbyname2(), která sice tuto podporu má, ale bylo složité ji naimplementovat tak aby jednoduše dosáhla mého žádaného výsledku. Při řešení problémů s touto funkcí jsem objevil funkci getaddrinfo() (7). Tato funkce podporovala vše, co jsem chtěl naimplementovat.

#### Zpracovávání zpráv od serveru

Zpracování packetů od serveru bylo nejsložitější, a i nejvíc časově náročné na celém projektu. Pokusil jsem se o 3 různé způsoby, jak navrhnout zpracování dat, tak, abych neudělal nikde chybu. Mým prvním nápadem bylo rozdělovat jednotlivé argumenty zprávy pomocí textového řetězce „“ “” – mezera ohraničená uvozovkami. Bohužel toto řešení nebylo správné, jelikož zde nastalo mnoho chyb, pokud uživatel chtěl zapsat jako poslední dva znaky právě uvozovku a mezeru.

Proto jsem se rozhodl zpracovat regulární výraz který by mi pomohl s tímto problémem a který fungoval skvěle, než jsem se pustil do rozsáhlého testování, kde jsem zjistil že kompilátor gcc má v sobě chybu při zpracovávání regulárních řetězců nad textových řetězcem větším než 27KiB a dochází k „stack overflow“ což je fatální chyba. (8) (9) (10)

Mým posledním a funkčním řešením bylo vytvořit funkci, která prochází řetězcem a kontroluje, zda se text nachází v uvozovkách či ne a přeskakuje znaky které jsou serverem (nebo klientem) escapovány.

#### Navazování spojení při zjištění chyby

Při zkoumání referenčního klienta jsem zjistil, že klient navazuje spojení i přestože nedochází ke komunikaci z důvodu chyby. Toto jsem se rozhodl neimplementovat abych nezatěžoval server.

#### Base64 konverze

Při vypracovávání klienta jsem si všiml že všechna hesla, se kterými se uživatel přihlašuje na server jsou konvertována do base64 pro základní bezpečnost. Tuto funkci jsem také implementoval s pomocí webové stránky (11), odkud jsem pochopil její základní princip a přepsal ho do jazyka C++.

### Návratové kódy

Soubor errorcodes.hpp obsahuje veškeré návratové kódy včetně vysvětlení jejich významů.

# Testování

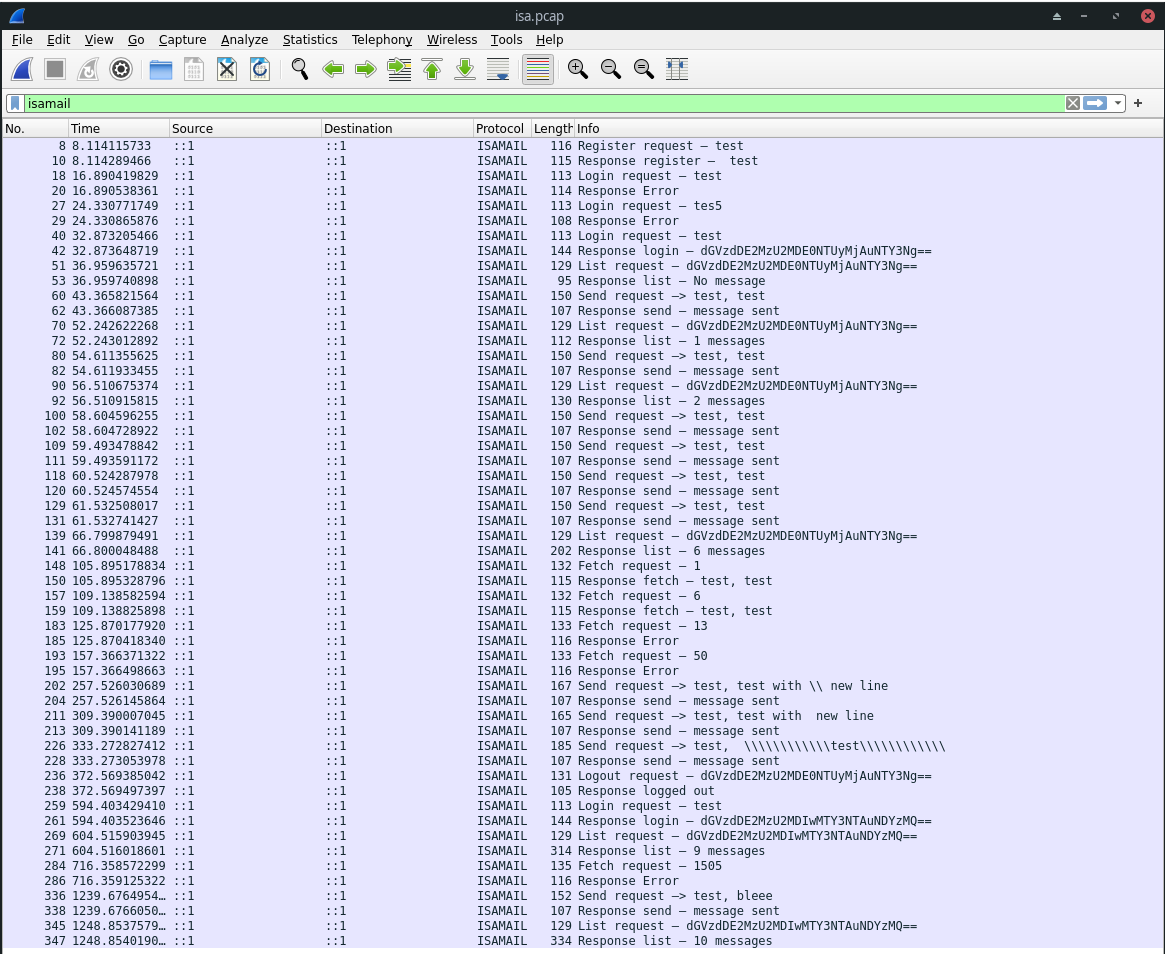
Aplikace jsem otestoval na těchto systémech:

1. Ubuntu 20.04.3 LTS on Windows 10 x86\_64 – (5.4.72-microsoft-standard-WSL2)
2. Manjaro Linux x86\_64, 5.13.19-2-MANJARO – referenční počítač

Bylo testováno samostatné sestavení projektu, a i jeho spuštění

## Wireshark disektor

Wireshark disektor jsem byl schopen otestovat i na systému Windows 10 Pro 21H1 – 19043.1288.  
Skript je nutné přidat do konfigurační Wireshark složky. Na Linuxu do „/root/.config/wireshark/” (na referenčním virtuálním stroji je nutné spustit wireshark jako root) a na Windows do „%APPDATA%\Roaming\Wireshark“. Je testován s referenčním clientem.



### Příklady spuštění

Kompletní Wireshark příklad spuštění na referenčním počítači na výchozím portu a adrese.

#### Login

Request a response:

Table

Description automatically generated

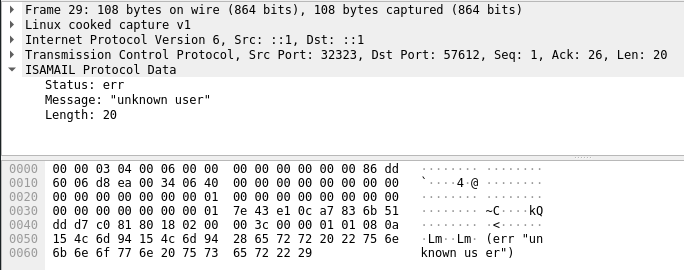
Table

Description automatically generated

Chybové stavy:

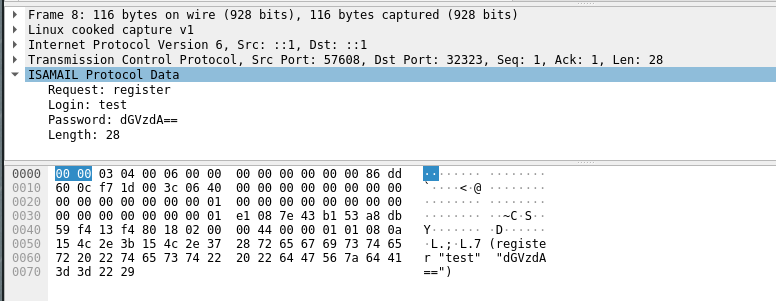
Table

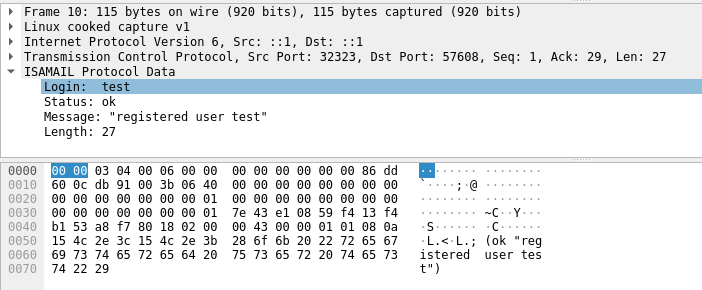
Description automatically generated



#### Register

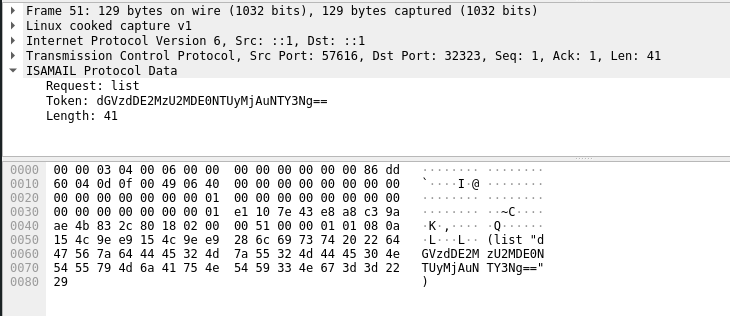
Request a response:

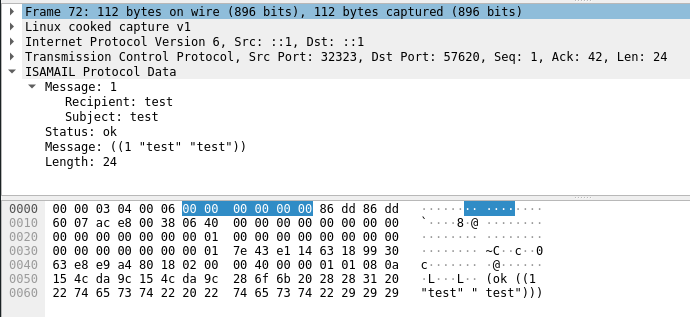


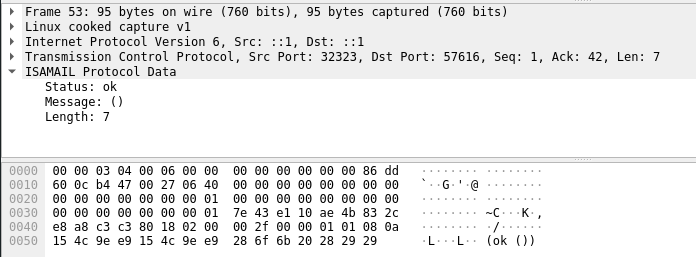


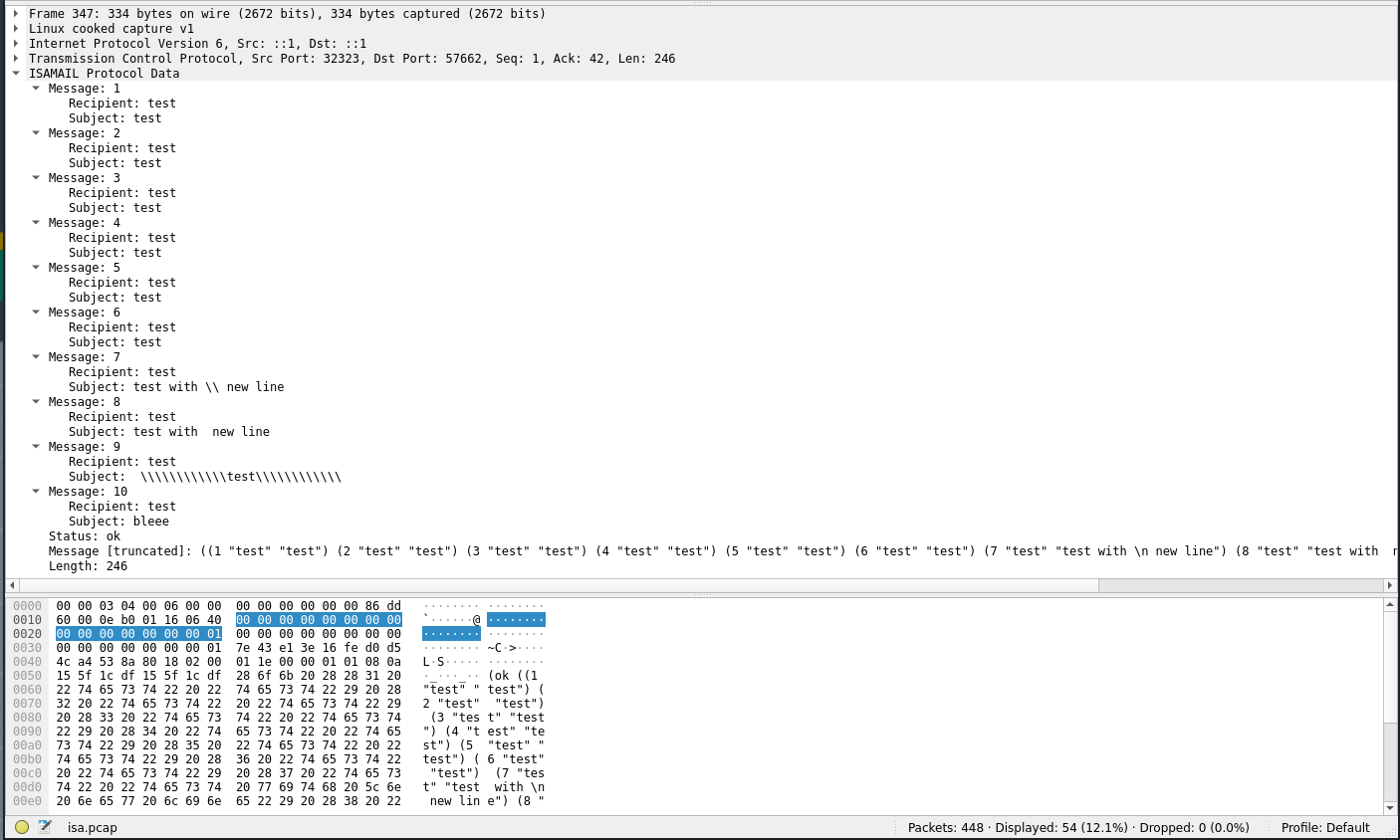
#### List

Request a response:



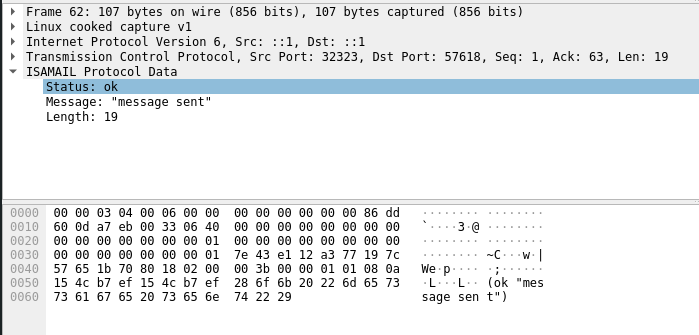


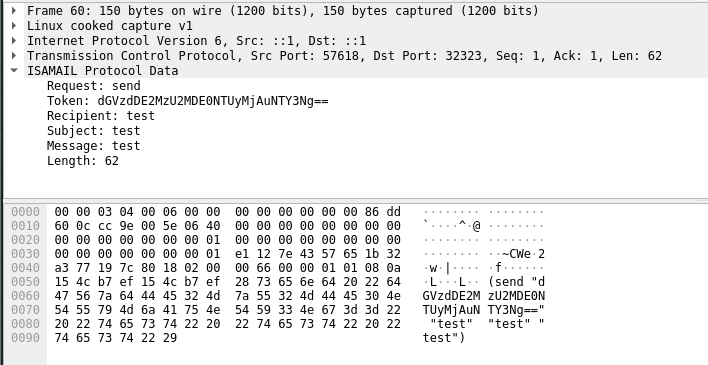




#### Send

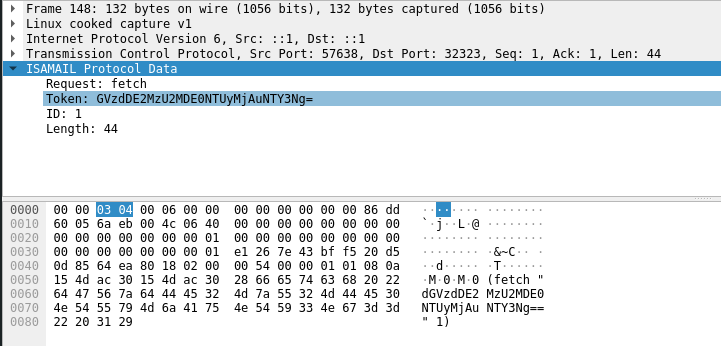
Request a response:

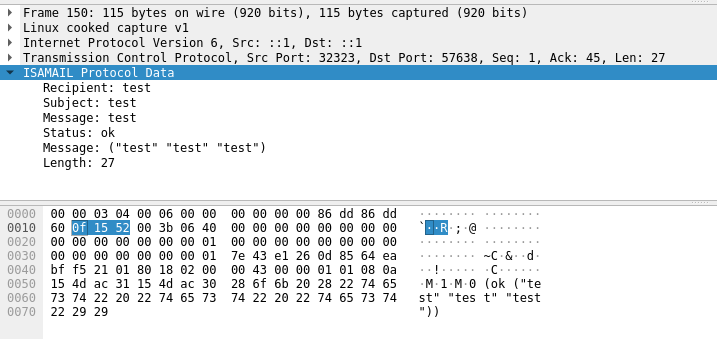




#### Fetch

Request a response:





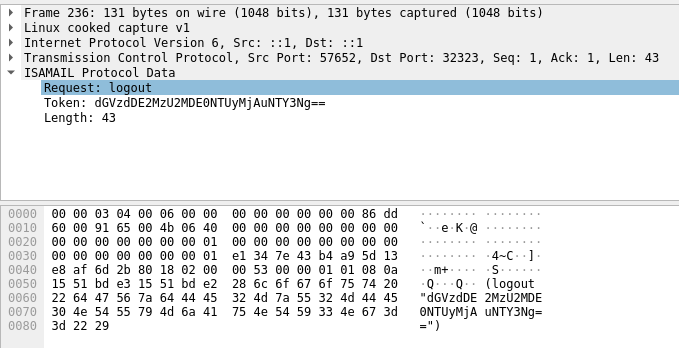
Chybové stavy:

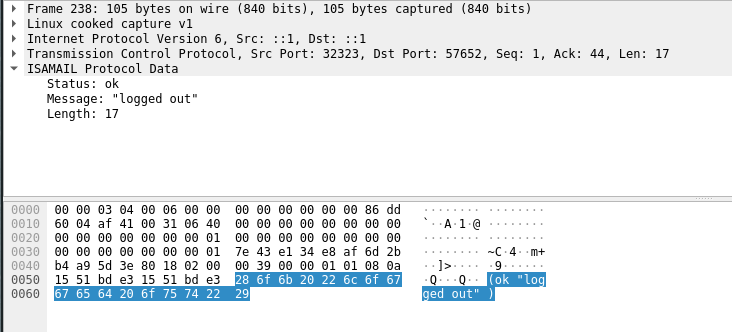
Calendar

Description automatically generated

#### Logout

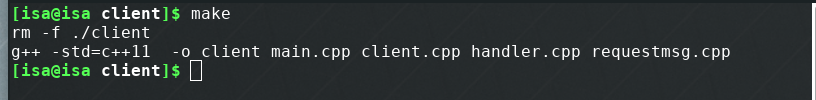
Request a response:





## Klient

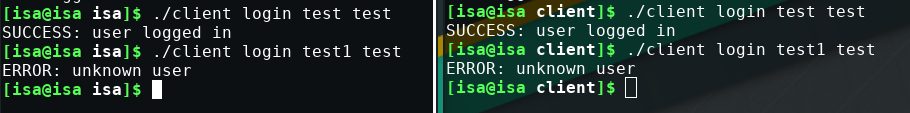
Klient se kompiluje pomocí vytvořeného makefile. Jsou dostupné příkazy „build” a „clean”, které vytvoří či smažou soubor „./client”.



### Příklady spuštění

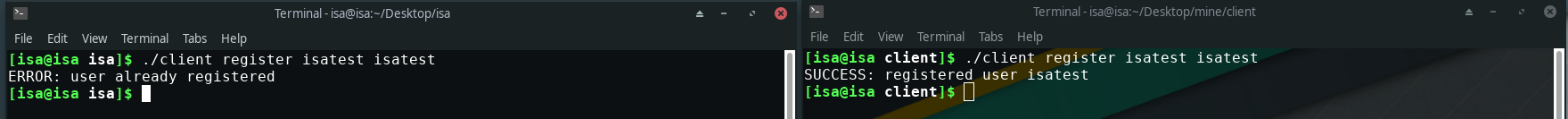
Moje aplikace je ve složce „client“ a referenční ve složce „isa”.

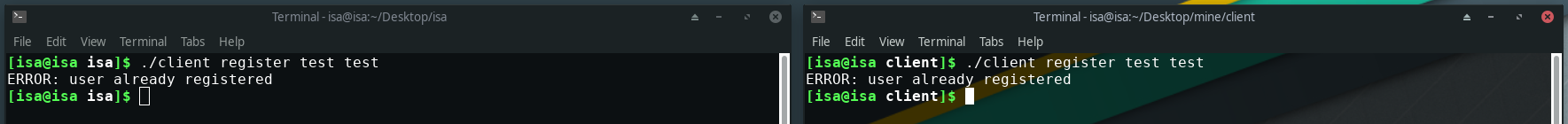
#### Login



#### Register

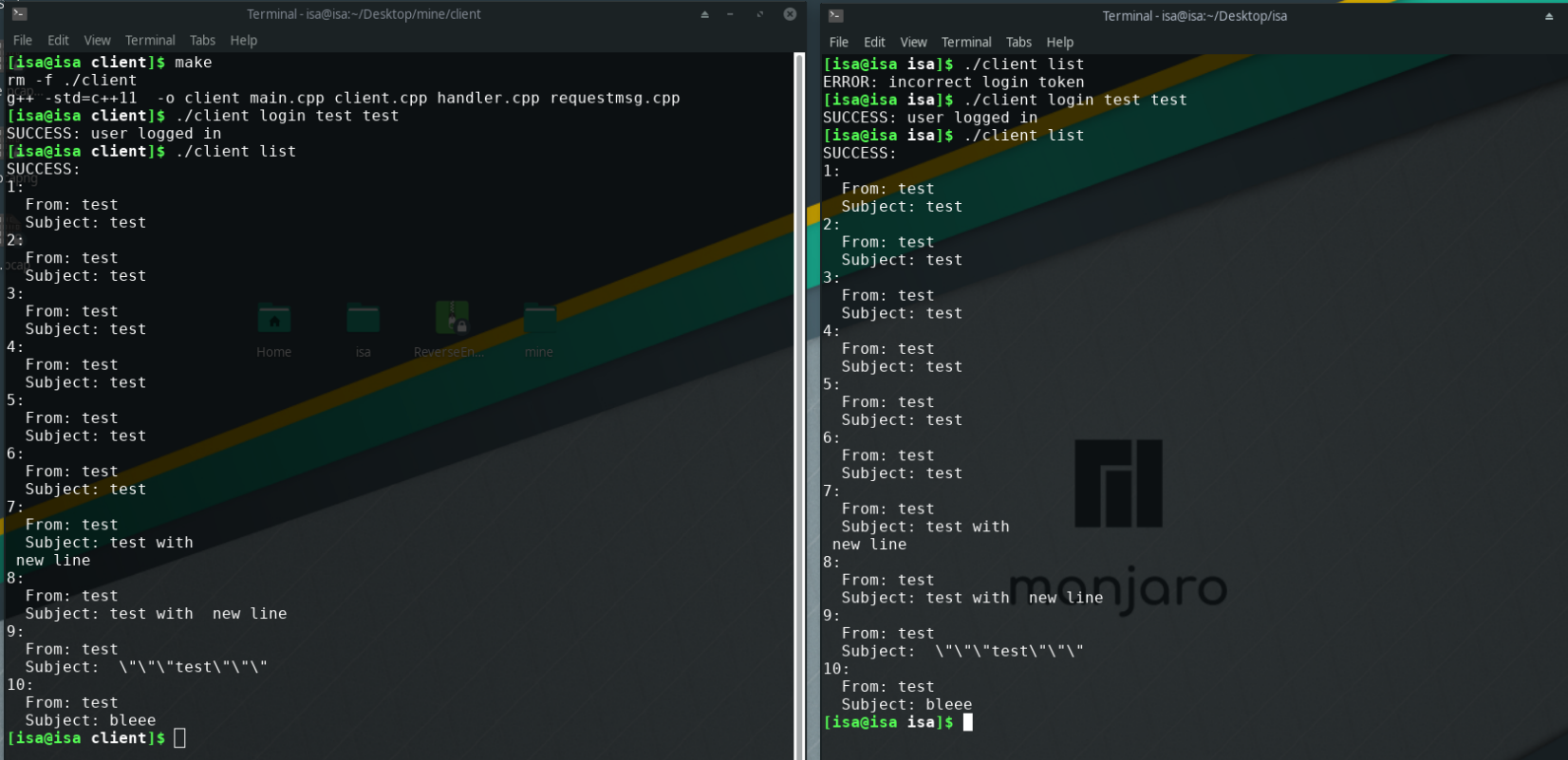
Tento příklad nebylo možné porovnat jelikož 2 uživatelé nemůžou být na server registrovaní 2x.



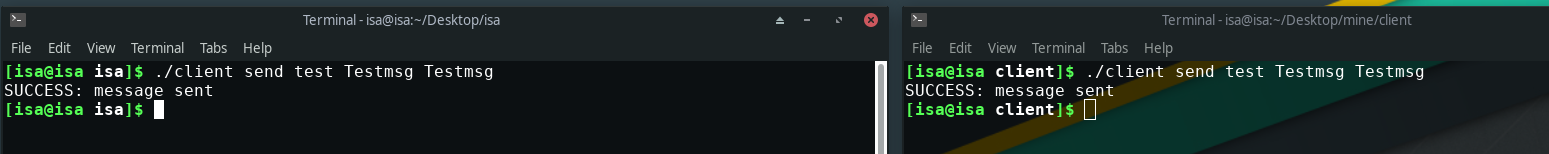


#### List

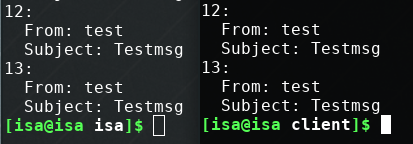
Tento fetch byl vytvořen send příkazy při testování disektoru.



#### Send



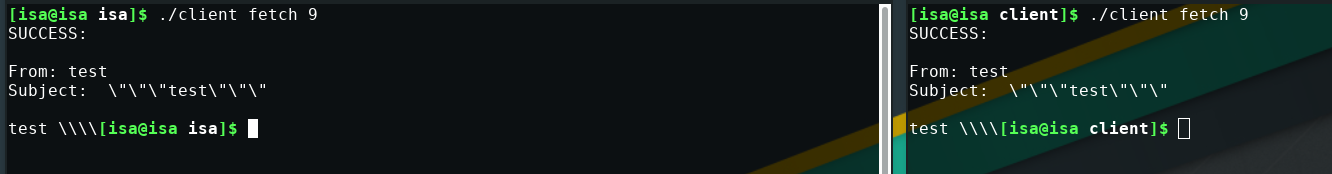
Validace, že obě zprávy byly poslány.

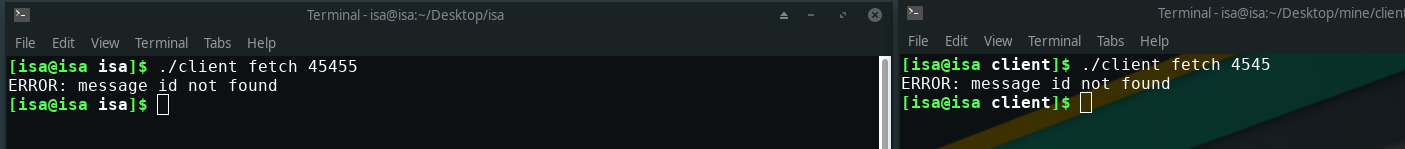


#### Fetch









#### Logout

#### 

# Reference

1. 1. **Wireshark Foundation.** https://gitlab.com. *Wireshark dissector – Examples.* [Online] 12. August 2020. [Citace: 26. October 2021.] https://gitlab.com/wireshark/wireshark/-/wikis/Lua/Examples.
2. 2. —. gitlab.com. *Wireshark wiki – Lua.* [Online] 8. October 2021. [Citace: 26. October 2021.] https://gitlab.com/wireshark/wireshark/-/wikis/Lua.
3. 3. —. https://gitlab.com. *Wireshark wiki – Dissectors.* [Online] Wireshark Foundation, 1. September 2020. [Citace: 26. October 2021.] https://gitlab.com/wireshark/wireshark/-/wikis/Lua/Dissectors.
4. 4. **Stevens (dist67), Didier.** https://www.youtube.com/. *Packet Class: Wireshark - Lua Protocol Dissectors.* [Online] 11. April 2014. [Citace: 26. October 2021.] https://www.youtube.com/watch?v=I4nf23HywmI.
5. 5. **Didier (dist67), Stevens.** https://didierstevens.com. *Didier Stevens Blog.* [Online] 10. August 2014. [Citace: 26. October 2021.] https://didierstevens.com/files/workshops/botnet01-dissector.lua.
6. 6. **Shukla 1, Yogesh.** https://www.geeksforgeeks.org. *GeeksForGeeks – TCP Server-Client implementation in C.* [Online] 23. September 2021. [Citace: 26. October 2021.] https://www.geeksforgeeks.org/tcp-server-client-implementation-in-c/.
7. 7. **Roymunson.** https://stackoverflow.com/. *Client in C++, use gethostbyname or getaddrinfo – StackOverflow.* [Online] 13. October 2018. [Citace: 26. 10 2021.] https://stackoverflow.com/questions/52727565/client-in-c-use-gethostbyname-or-getaddrinfo.
8. 8. **chaoskeeper.** https://gcc.gnu.org. *GCC Bugzilla – Bug 70459.* [Online] 30. March 2016. [Citace: 26. October 2021.] https://gcc.gnu.org/bugzilla/show\_bug.cgi?id=70459.
9. 9. **Seelig, Holger.** https://gcc.gnu.org. *GCC Bugzilla – Bug 86164.* [Online] 15. June 2018. [Citace: 26. October 2021.] https://gcc.gnu.org/bugzilla/show\_bug.cgi?id=86164.
10. 10. **Arciemowicz, Maksymilian.** https://gcc.gnu.org/. *GCC Bugzilla – Bug 61582.* [Online] 23. June 2014. [Citace: 26. October 2021.] https://gcc.gnu.org/bugzilla/show\_bug.cgi?id=61582.
11. 11. **Victor.** https://base64.guru. *Base64 Encode Algorithm.* [Online] 2019. [Citace: 26. October 2021.] https://base64.guru/learn/base64-algorithm/encode.