VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Síťové aplikace a správa sítí – Programování síťové služby

Reverse-engineering neznámeho protokolu (varianta 3)

25. října 2021

David Hudák

Obsah

1 Úvod

Tato dokumentace se věnuje zpracování zadání 3. varianty zadání projektu do předmětu Síťové aplikace a správa sítí. Projekt i tato dokumentace byly vypracovány samostatně.

1.1 Zadání (výtah)

Projekt se zabývá reverse engineeringem protokolu poskytnutého v rámci virtuálního počítače obsahujícího server a referenčního klienta. Výsledkem projektu má být implementace „falešného“ klienta, který by měl co nejblíže fungovat jako klient referenční. Součástí implementace má být i základní analýza skrze Wireshark formou `pcap` souboru, který má obsahovat základní informace nezbytně nutné k implementaci protokolu. K pomoci v analýze má být vytvořen dissector pro Wireshark v jazyce **Lua**.

2 Základní informace získané z analýzy nástrojem Wireshark

Tato část dokumentace se věnuje základním detailům komunikace klienta a serveru. Součástí této sekce nejsou informace, jež se jeví jako nadbytečné a jež nejsou pro svou detailnost v dokumentaci důležité.

2.1 Spuštění serveru

Na základě spuštění serveru s argumentem `-h` lze rozlišit pouze dva hlavní argumenty, a to argument pro adresu a argument pro port. Defaultním portem a adresou, na které server naslouchá, je port `32323` (dá se případně argumentem změnit) a adresa `localhost` (v případě virtuálního stroje `IPv6` adresa `::1`). Co se týká možných nastavení, tak jako adresu lze použít jakoukoliv, kterou vlastní zařízení (IPv4 i IPv6), na kterém má být spuštěn server, a port jakýkoliv v rozumném rozsahu. Na této adrese server naslouchá klientovi (klientům), kteří mají výchozí nastavení kam odesílat stejné jako výchozí nastavení nastavení serveru. Samotný server se pak dá spustit příkazem `./server`.

Při opětovném spouštění serveru se projevil fakt, že server po vypnutí maže všechna svá data (nepamatuje si je a ani je nezapisuje do žádného externího souboru).

2.2 Spuštění klienta

Klient má základní možnosti spuštění podobné jako server. Může si nastavit adresu a port, ale má své implicitní údaje, kam bude zasílat své zprávy (opět `localhost = ::1` a port `32323`). Argumentem, který musí být přítomen vždy, je nějaký konkrétní příkaz a potřebnými argumenty. Klient se automaticky po provedení svých úkonů (například odeslání zprávy) automaticky vypne. Stejně jako server si vnitřně sám nic nepamatuje, ale při přihlášení do systému (respektive odhlášení) pracuje se souborem `login-token`, který obsahuje zašifrovaný token pro práci se serverem.

Pro spuštění klienta s výchozími parametry pak voláme klienta příkazem:

```
./client <příkaz> [<argumenty>]
```

kde příkaz může být cokoliv z množiny `{register, login, list, fetch, send, logout}` a argumenty jsou pak nutné podle toho, jaké argumenty žadá daný příkaz.

2.3 Typ spojení

Typem spojení je TCP, které provádí nejprve operace navázání spojení (Three-Way Handshake) a následně odesílá zprávu, na kterou, pokud je zapsána správně, server reaguje svou zprávou obsahující odpověď (případně při příliš velké velikosti zprávy rozdělí zprávu do více paketů, což je nutné v klientovi ošetřit). Následně je TCP spojení hned ukončeno.

2.4 Syntax zpráv

Syntax zpráv pro všechny příkazy je víceméně stejná. Jedná se o text v závorkách (nejvíce levá závorka počátek zprávy, nejvíce pravá závorka ukončení zprávy). Jako první po první levé závorce následuje typ odpovědi (v případě klienta příkaz, v případě serveru err či ok) a následně je buď v horních uvozovkách vyjmenován postupně každý odesílaný parametr (ať už rozvedení odpovědi serveru, text zprávy či chybové hlášení), nebo v případě například příkazu `list` server odesílá pro každou zprávu další závorku, ve které se pak v uvozovkách nachází jednotlivé hlavičky dané zprávy (více podrobností v příslušné sekci 3.3).

Pro ilustraci můžeme uvést odpověď serveru na dotaz obsahující poškozený token přihlášení (buď úpravou patřičného souboru s tokenem, nebo zavoláním klienta na stejného uživatele z jiného složky):

```
(err "incorrect login token")
```

3 Analýza příkazů klienta

3.1 Příkaz `register`

Příkaz má dva povinné argumenty, první pro jméno uživatele a druhý pro heslo. Heslo je šifrováno šifrou Base64[3].

Reakce klienta na daný příkaz se odvíjí od správnosti zadaných argumentů. V případě, že uživatel zadá méně argumentů či žádný, klient nenaváže spojení a vytiskne správnou syntax příkazu. V případě správně uvedených argumentů server zareaguje tak, že se podívá do své databáze (která se po smazání serveru smaže) a rozhodne, zdali už uživatel existuje, nebo zdali je vstupní jméno unikátní. Při registraci nedochází k přihlášení klienta (vzniku souboru `login-token`), tudíž je potřeba se po registraci přihlásit (pokud uživatel chce provádět nějaké operace).

3.2 Příkaz `login`

Příkaz má úplně stejný syntax jako příkaz `register` (opět používá i stejné šifrování). Liší se reakce serveru a také fakt, že při správném přihlášení klient vytváří soubor `login-token`, kde je v uvozovkách uveden řetězec vzniklý s pomocí šifry Base64, a to nějakou kombinací uživatelského jména a náhodného čísla (zřejmě z toho důvodu, aby nebylo token tak snadné uhodnout). Uživatel se může přihlásit pod stejným jménem vícekrát a dokonce při opětovném přihlášení je možné uvést špatné heslo (zřejmě bug referenčního klienta). Pokud uživatel není přihlášen a uvede špatné heslo, server zareaguje patřičnou zprávou. Server také může odmítnout uživatele, pokud neexistuje.

3.3 Příkaz `list`

Příkaz list sestává pouze z příkazu list a žádného argumentu. Ve zprávě zasílané serveru je pak nutné uvést jak příkaz list, tak i token ze souboru vzniklého po přihlášení (pokud token není, příkaz zhavaruje a ani se neodešle). Od tokenu se i odvíjí reakce serveru – pokud je chybný, odmítne, pokud je token správný, tak odešle hlavičky zpráv, které patří danému uživateli. Hlavičky jsou – id zprávy (generuje server, začíná od 1), zdroj zprávy (uživatel, který ji odeslal) a předmět zprávy. V případě, že je zpráv příliš mnoho, server zprávu rozbíjí na více paketů, cože musí klient náležitě obsloužit.

3.4 Příkaz `send`

Příkaz slouží k odeslání zprávy. Sestává se ze samotného příkazu a tří argumentů – cílový adresát, předmět zprávy a samotná zpráva. Zpráva může být neomezené délky. Server reaguje buď tak, že není zadán správný token přihlášení (je opět zahrnut do zprávy, kterou odesílá klient), nebo že adresát neexistuje, nebo potvrzením odeslání zprávy.

Pokud je příkaz úspěšný, odešle zprávu adresátovi (klidně i sobě samému) a zprávu si následně uživatel může zobrazit formou náhledu příkazem `list` (ukazuje odesílatele a předmět) a příkazem `fetch` s příslušným id zobrazit a přečíst. Zprávy nejsou nijak šifrovány.

3.5 Příkaz `fetch`

Příkaz slouží k načtení zprávy. Příkaz má jeden argument, a to id zprávy.

Nejprve je nutné zjistit žádané id (či ho případně uhodnout – čísluje se od 1, takže to není složité; normální variantou je příkaz `list`) a následně ho zadat. Pokud je id číslo, klient reaguje odesláním zprávy, která kromě příkazu a id obsahuje ještě token přihlášení. Opět je nutné, aby byl odesílaný token validní, jinak zprávu odmítne server. V případě, že se jedná o záporné id, tak to oznámí až server ve své odpovědi. Pokud id zprávy neexistuje, reaguje příslušnou zprávou. Pokud id existuje, server odesílá klientovi zprávu obsahující řetězce s odesílatelem, předmětem a samotnou zprávou.

3.6 Příkaz `logout`

Příkaz slouží k odeslání informace o odhlášení klienta. Kromě zaslání příkazu a tokenu maže soubor s tokenem u klienta. Server na zprávu reaguje tak, že buď token zná, nebo nezná a vnitřně uživatele odhlásí (zneplatní původní token přihlášení).

4 Dissector pro Wireshark pro protokol ISA

V této sekci se budu zabývat návrhem a implementací dissectoru pro nástroj Wireshark implementovaný v jazyce Lua. Při implementaci jsem primárně vycházel z návodu uvedeného zde[2]. Protokol byl z praktických důvodů nazván ISA.

4.1 Dissectory obecně

Dissectory slouží Wiresharku (a podobným nástrojům) pro zpracování dílčích paketů. Dokáží určit protokol, rozepsat jednotlivé parametry (zdroj, cíl, port, obsah) a pomáhají tak uživateli Wiresharku s rozpoznáváním paketů. Způsob jejich implementace je různý, použít se dají například jazyky C a Lua. Aplikace dissectorů je paralelní s přijímáním paketů a mohou být aplikovány i již nad doručenými pakety.

4.2 Zpracování dissectorů v projektu

Příjem paketů a jejich následné zpracování v dissectoru je celkově velmi lineární. Paket je nejdříve službou Wireshark rozlišen jako ISA paket (na portu 32323 a protokolu TCP) a následně se spouští dissector pro tento paket.

Pro rozlišení odesílatele paketu se nejdříve přečte přijatá zpráva. Pokud jsou první tři znaky po první závorce `"err"` nebo `"ok "`, pak dojde ke konstatování, že se jedná o zprávu ze serveru a provedou se příslušné kroky. U zprávy začínající na `"err"` je zpracování celkem prosté – pouze se přečte chybové hlášení obsažené v řetězci zprávy a vytiskne se pod vhodnou hlavičkou na výstup pro Wireshark (přidá se do substromu stromu protokolu).

V případě, že byla přijata zpráva `"ok "`, je nutné provést komplexní analýzu obsahu. Pokud se ve zprávě nenachází žádné dodatečné závorky (mimo ty, které jsou v rámci řetězce v uvozovkách), musí se jednat o odpověď na příkaz `login`, `register`, `sendn` nebo `logout`. Dalšího rozlišení se dá dosáhnout s pomocí spočítání počtu řetězců a následně se čtou samotné zprávy. Pro příkazy `fetch` a `list` je to snazší v tom, že v prvních závorkách stačí narazit na číslo (protože `list` používá pro odlišení jednotlivých zpráv indexaci). Při rozlišení jednotlivých příkazů se pak provádí částečně individuální analýza (někde stačí změnit jméno popisu, někde je potřeba analyzovat více řetězců), ale víceméně vždy jsou tyto funkce velmi podobné (implementace jednodušších konečných automatů).

Pro zprávy, které obsahují jako první po závorce hned napsaný příkaz, je zpracování poměrně jednodušší. Podle syntaxe daného příkazu lze rovnou volat příslušný konečný automat (implementovaný), který zpracuje zprávy a jejich obsah pro daná políčka vrátí přes textové řetězce.

5 Implementační detaily

Kódová část projektu (bez dissectoru viz 4) sestává z 5 souborů:

- `main.cpp` – zdrojové kódy projektu, obsahuje implementaci klienta
- `main.hpp` – hlavičkový soubor projektu, obsahuje hlavičky funkcí pro implementaci klienta
- `base64.cpp` – soubor byl převzat z GitHubu[1], obsahuje zdrojové kódy pro implementaci šifry Base64
- `base64.hpp` – soubor byl převzat z GitHubu[1], obsahuje hlavičky pro implementaci šifry Base64
- `Makefile` – soubor obsahuje příkazy pro sestavení projektu příkazem make

5.1 Zpracování argumentů

Pro zpracování argumentů byla využita knihovna `getopt`, která spoustu problémů se zpracováním argumentů řeší. Samotné příkazy jsou pak řešeny obyčejným stromem příkazů `if`.

Problémem při využití knihovny `getopt` se stala situace, kdy došlo k zadání příkazu:

```
./client fetch -10
```

Knihovna `getopt` se k tomuto staví až příliš proaktivě a začne zpracovávat jak fetch, tak i -10 jako argumenty, což není žádoucí z hlediska zpracování příkazů. Z toho důvodu bylo nutné obejít tento problém s využitím zabudované proměnné `optind`, která obsahuje číslo prvního následujícího argumentu. V implementaci je tak od ní odečítána ve větvi „switch nenalezen“ (sloužící pro řešení zpracování argumentů, které nejsou podporovány, například při zadání `./client -d`). Díky tomu lze zjistit řetězec posledního vstupního argumentu. Pokud je tento poslední vstupní zpracovávaný argument některý z příkazů (viz 3), tak se vytvoří značka `overflow`, která vyvolá v cyklu `getopt` příkaz `break`.

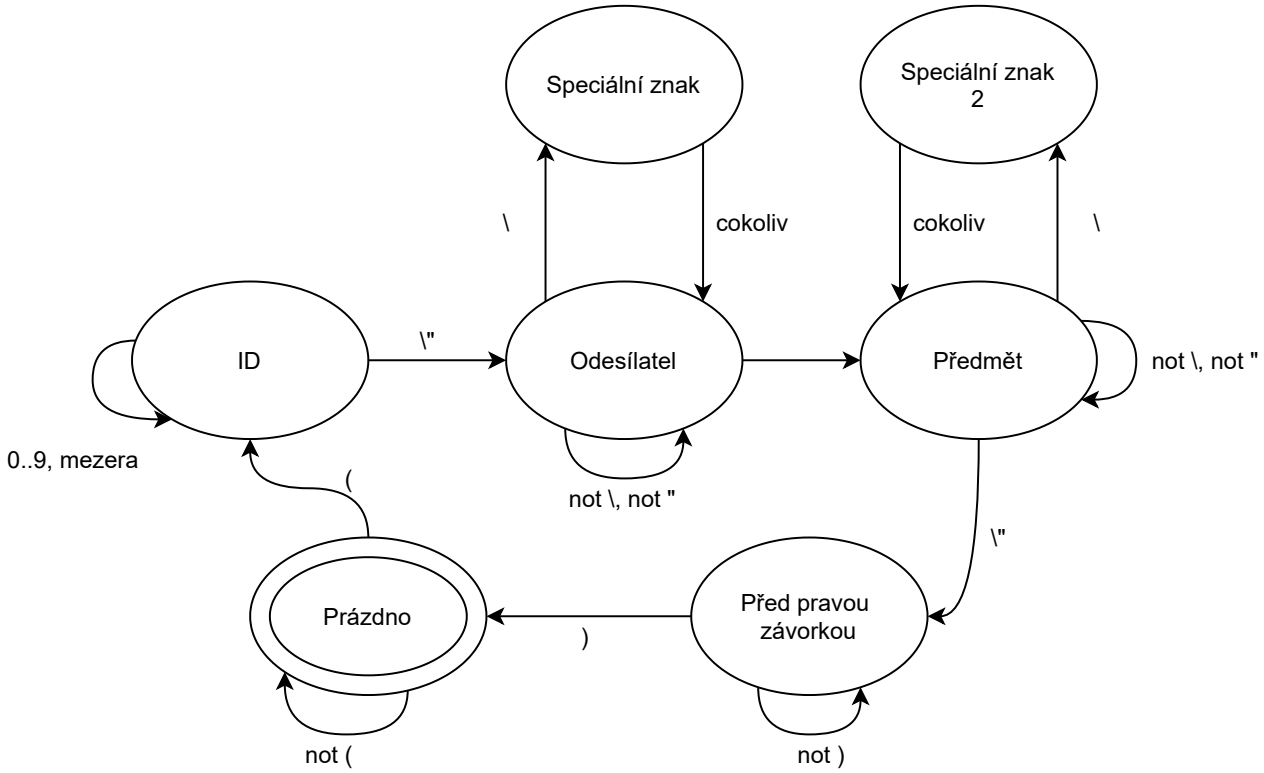
Aktualizace: Tato subsekce se stala neaktuální. Řešení nabízí `getopt` sám v rámci své funkcionality. Do řetězce krátkých možností stačí na první pozici přidat znak `+`.

5.2 Zpracování příkazů

Každý příkaz má svou vlastní funkci. Před zavoláním každé této funkce je nutné vytvořit socket a ustavit spojení, načež všechny funkce implementující příkazy přijímají jako svůj parametr číslo otevřeného socketu. Všechny funkce provádí vytvoření řetězce zprávy, odeslání řetězce zprávy, přijetí řetězce zprávy (u příkazů `list` a `fetch` je za tímto účelem zařízen cyklus pro přijímání zprávy).

5.3 Escape sekvence

Každá zpráva odesílaná serveru má speciální zápis speciálních znaků. Například `'\n'` není odesílán tak, jak bychom jej zapsali v C (buď uvedenou escape sekvencí, nebo případě dohledáním čísla ASCII kódu v tabulkách), nýbrž rozepsáním do zprávy na `'\'` a `'n'`. Je to celkově zvláštní a je s tím u mnoha znaků nutné počítat. Z toho důvodu byly implementovány překládací funkce, které mají za úkol pouze převést každý znak do takovéto formy (možná mírně redundantní) a naopak.



Obrázek 1: Zjednodušený konečný automat implementovaný v projektu (chybí některé stavy pro mezery)

5.4 Zpracování přijatých zpráv

Nejzajímavější částí řešení je zpracování přijetých zpráv. Z tohoto důvodu byly implementovány (cca) dílčí konečné automaty, které ošetřují všechny stavy, které mohou nastat. Nejsložitější konečný automat byl implementován v případě zpracování odpovědi na příkaz `list`, kde je nutné rozlišovat jednotlivé části zprávy (id, zdroj, předmět) a zprávy samotné od ostatních zpráv. Viz 1.

Tento automat neslouží primárně k tomu, aby verifikoval, zdali jsou zaslané zprávy korektní (při zpracování klienta se počítá s tím, že server nemá chyby), ale primárně využívá možnosti ke každému přechodu přiřadit nějakou užitečnou funkci. Například při přechodu ze stavu `Prázdno` do stavu `ID` se vyčistí všechny řetězce pro ukládání zprávy. Při cyklení ve stavu se naopak přidávají znaky do řetězců příslušných daným stavům (`ID` má řetězec na ukládání ID, `Odesílatel` pro ukládání loginu odesílatele a `Předmět` pro ukládání předmětu zprávy).

6 Použité knihovny

6.1 Sockety a síťové utility

Pro navázání spojení byly využity standardní BSD sockety[4]. Jejich použití je probíráno v rámci předmětu, ve kterém je vytvářen tento projekt. Využity jsou primárně funkce `sock()`, `connect()`, `send()` a `read()`. Pro zpracování IP adresy a portu do strojově zpracovatelné formy byla využita funkce `getaddrinfo()`.

6.2 Implementace knihovny pro práci se šifrou Base64

V implementaci projektu byla využita knihovna veřejně dostupná na GitHubu[1] s licenčně umožněným použitím pro účely zmíněné v projektu. Z této knihovny byla využita funkce pro šifrování dle algoritmu Base64.

7 Omezení

V době psaní této dokumentace nejsou známa žádná omezení oproti zadání.

8 Závěr

Výsledkem projektu je „falešný klient“ vykonávající optimálně co nejstejnější kroky jako klient referenční. Součástí řešení je též dissector pro usnadnění analýzy zpracovávaného protokolu a základní záznam analýzy protokolu z nástroje Wireshark.

9 Reference

[1] Nyffenegger, R.: cpp-base64. [online], Naposledy navštíveno 24. 10. 2021.
URL https://github.com/ReneNyffenegger/cpp-base64

[2] Sundland, M.: Creating a Wireshark dissector in Lua. [online], Naposledy navštíveno 24. 10. 2021, mail@sund.land.
URL https://mika-s.github.io/wireshark/lua/dissector/2017/11/04/creating-a-wireshark-dissector-in-lua-1.html

[3] Wikipedia: Base64. [online], Naposledy navštíveno 24. 10. 2021.
URL https://en.wikipedia.org/wiki/Base64

[4] Wikipedia: Berkeley Sockets. [online], Naposledy navštíveno 24. 10. 2021.
URL https://en.wikipedia.org/wiki/Berkeley_sockets