VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

IMPLEMENTACE PLUGINU PORTMAP DO PROJEKTU BUSYBOX

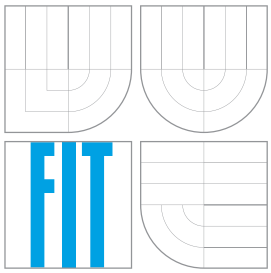
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUKÁŠ HUBA

BRNO 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

IMPLEMENTACE PLUGINU PORTMAP DO PROJEKTU BUSYBOX

A PORTMAP PLUGIN FOR BUSYBOX

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — LUKÁŠ HUBA
AUTHOR

VEDOUCÍ PRÁCE — Doc. Ing. TOMÁŠ VOJNAR, Ph.D.
SUPERVISOR

BRNO 2011

Abstrakt

Tato práce se zabývá implementací portmapu pro balík BusyBox. První část se zabývá teoretickou průpravou a konkrétně zasvěcuje čtenáře do technologie volání vzdálených procedur (RPC), datové reprezentace (XDR) a portmap protokolu. Druhá část práce stručně seznamuje čtenáře s projektem BusyBox a popisuje implementaci pluginu portmap. V závěru se pojednává o dosažených výsledcích.

Abstract

This bachelor thesis deals with an implementation portmap of the BusyBox package. The first part deals with the theoretical background of the thesis in particular, introducing the technology of remote procedures call, data representation, and the portmap protocol. The second part of the thesis briefly acquaints the reader with the BusyBox project and describes an implementation of the portmap plugin done within the the thesis. The conclusion of the thesis summaries the work that has been done and discusses possible future work in the area.

Klíčová slova

BusyBox, RPC, XDR, portmap

Keywords

BusyBox, RPC, XDR, portmap

Citace

Lukáš Huba: Implementace pluginu portmap do projektu BusyBox, bakalářská práce, Brno, FIT VUT v Brně, 2011

Implementace pluginu portmap do projektu BusyBox

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doc. Tomáše Vojnara. Další informace mi poskytla mgr. Ivana Vařeková, zástupkyně společnosti Red Hat. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Lukáš Huba
18. května 2011

Poděkování

Rád bych poděkoval mému vedoucímu doc. Tomáši Vojnarovi za odborné vedení, cenné rady a čas, které mi při tvorbě práce věnoval. Zároveň bych chtěl poděkovat mgr. Ivaně Vařekové, zástupkyni společnosti Red Hat, za podporu a poskytnuté cenné rady, které mi věnovala při tvorbě práce.

© Lukáš Huba, 2011.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Kapitola 1

Úvod

Bouřlivý vývoj informačních technologií sebou přináší neustále nové požadavky na hardwarové i softwarové vybavení. Od nízkoúrovňových programovacích jazyků, které jsou platformně závislé, avšak optimalizované pro danou architekturu, se postupně přešlo na vysokoúrovňové programovací jazyky, které nejsou závislé na hardwarové architektuře[1], ale zároveň nejsou tak efektivní jako nízkoúrovňové jazyky. Dnešní trend je zobecňovat a tvořit zdrojové kódy univerzální, nejlépe v objektově orientovaném konceptu, protože je to ekonomicky výhodnější – cena času vynaloženého pro řešení „na míru“ je mnohdy řádově vyšší, než cena výkonnějšího hardware. Tento přístup má za následek plýtvání systémovými prostředky.

Existují však systémy, které kladou důraz na požadavky opačného rázu. Namísto robustních multifunkčních programů, jsou vyžadovány co nejmenší a co nejefektivnější programy se základní funkcionalitou. K vývoji takto minimalistických programů jsou používány jazyky na nižší úrovni. Velice často používaným jazykem, splňující tyto požadavky, je jazyk C. Častokrát je také nutné tvořit vlastní funkce, či celé knihovny, které nahrazují běžné knihovní funkce jazyků a napomáhají tak zefektivnit a minimalizovat zdrojový kód. Řeší se každý řádek zdrojového kódu a každý byte, který přijde na zmar. Jeden z těchto projektů s takto minimalistickým přístupem je BusyBox. BusyBox (a podobné) systémy nacházejí své využití zejména ve vestavěných systémech, které mají častokrát velice omezené hardwarové vybavení.

Cílem této práce bylo vytvořit ve spolupráci s firmou Red Hat optimalizovaný plugin portmap, který by splňoval přísná kritéria projektu BusyBox. Součástí této práce je také oprava pluginu patch.

Vzhledem k tomu, že je pro komplexní pochopení této práce důležitá znalost volání vzdálených procedur (RPC), datové reprezentace (XDR) a portmap protokolu, je nutné nejprve čtenáře seznámit s principy a funkčností těchto technologií. První kapitola proto uvádí do teorie volání vzdálených procedur, která je pilířem celé vzdálené komunikace. Druhá kapitola popisuje, jakým způsobem jsou data reprezentována v systému RPC a portmap protokolu. Třetí kapitola zužuje svůj záběr a soustředí se na protokol, prostřednictvím kterého probíhá veškerá komunikace mezi portmapperem a aplikacemi, které užívají ke své činnosti metodu volání vzdálených procedur. Touto kapitolou končí teoretická část tohoto textu a následující kapitoly směřují přímočaře k cíli této práce. Čtvrtá kapitola seznamuje čtenáře s platformou, ve které probíhal vývoj a nastiňuje její charakteristické rysy. Rovněž ukazuje, jakým způsobem lze BusyBox získat, konfigurovat a instalovat. Následující část

[1] Závislost na architektuře řeší překladač.

textu se zabývá vývojem pluginu portmap pro balík BusyBox. Sedmá kapitola analyzuje funkčnost nově vytvořeného appletu a snaží se odhalit chyby vzniklé při implementaci. Popisuje a nastoluje také situace a rizika, kterým může být applet portmap vystaven vlivem potenciálních útoků. Ukazuje také různé typy útoků, které by mohly vyřadit nebo nabourat chod vzdáleného systému a jak jim implementace portmapu odolává. Další kapitola popisuje opravu appletu patch, která byla provedena v rámci seznamování s projektem BusyBox. Závěrečná kapitola shrnuje průběh práce, dosažené výsledky a diskutuje možné budoucí fáze vývoje.

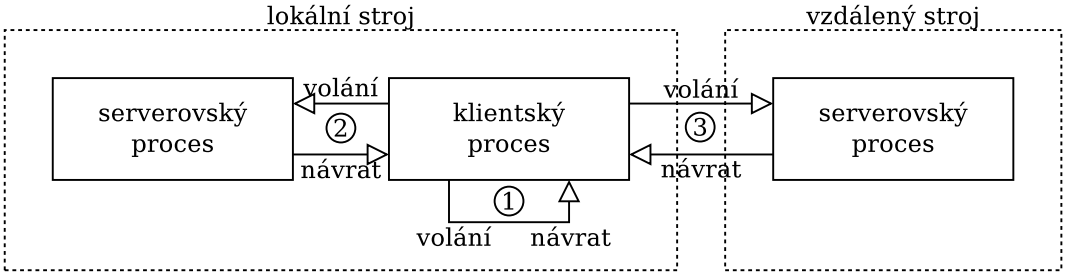
Kapitola 2

RPC – vzdálené volání procedur

Plugin portmap používá ke komunikaci s RPC aplikacemi techniku volání vzdálených procedur. Je to vrstva, která portmapperu umožňuje přenášet zprávy a autentizovat klientské aplikace. První části kapitoly se zabývají úvodem do RPC a principem této technologie. V další části nahlédneme do struktury RPC zpráv. V následující podkapitole se dozvíme, jakým způsobem RPC ověřuje své klienty. Další část kapitoly seznamuje se základními rysy jazyka RPC, pomocí kterého se popisují RPC programy (např. portmap). Na závěr je v krátkosti naznačeno, jak RPC aplikace psát.

2.1 Úvod do RPC

Vzdálené volání procedur (RPC [2], [10], [7], Remote Procedure Call) je technika, která umožňuje jednoduchým způsobem tvořit síťové aplikace bez potřeby znalostí nízkoúrovňového síťového programování. Klient volá transparentním způsobem proceduru, která je vykonána na straně serveru.

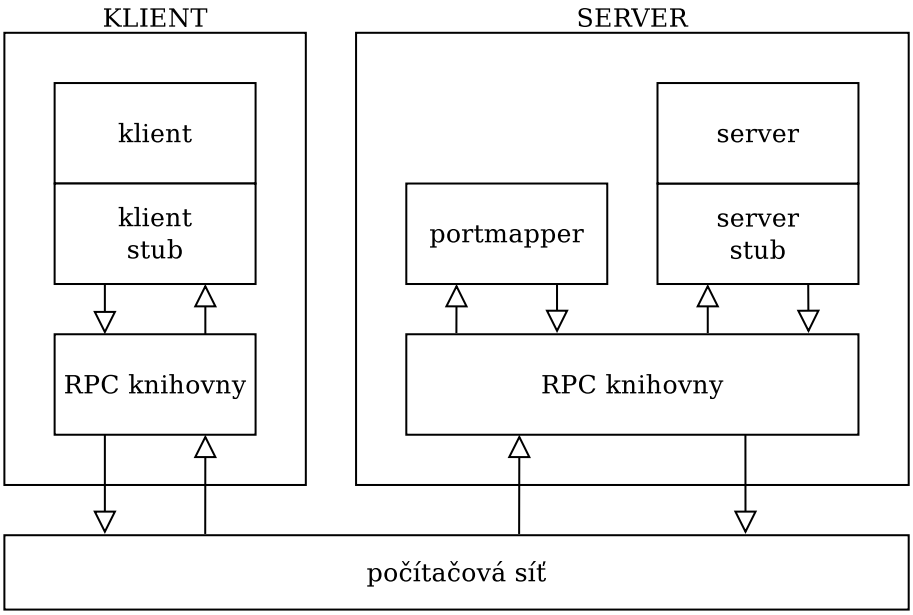


Obrázek 2.1: Volání procedury

Na obrázku 2.1 jsou znázorněny tři typy volání procedury. První typ ukazuje volání lokální funkce, druhý typ volání vzdálené procedury na lokálním počítači a třetí typ ukazuje volání procedury, která se nachází na vzdáleném systému. Při volání lokální funkce jsou ukládány parametry funkce na zásobník a poté předáno řízení proceduře programu. Po dokončení procedury přebírá řízení opět program. U vzdáleného volání pošle klient zprávu serveru, která obsahuje informace o tom, kterou proceduru klient volá a hodnoty předávaných parametrů. Server zprávu přijme, provede výpočet procedury a odešle odpověď zpět. Technologie RPC má ale oproti volání lokálních funkcí jisté nevýhody. Odezva na volání vzdálené procedury je řádově delší, nelze předávat ukazatele a je nutná autentizace.

2.2 Implementace RPC

Volání vzdálených procedur nabízí programátorům rozhraní mezi jejich programy a knihovnami RPC. Toto rozhraní je nazýváno `stub` a je integrováno v klientském i serverovském programu. Jde o mezivrstvu, která zajišťuje transformaci dat mezi aplikační a prezentační vrstvou, sestavuje a dekomponuje RPC zprávy, registruje procedury a zjišťuje porty vzdálených procedur. Z pohledu klienta `stub` zakóduje číslo vzdálené procedury a její parametry do formátu XDR (kap. 3), sestaví `call` zprávu a předá ji knihovnám RPC pro odeslání. Až obdrží odpověď, dekóduje výsledek a předá jej klientskému programu. Z pohledu serveru `stub` zaregistruje proceduru a uvede se do spícího stavu, kdy čeká na příchozí zprávy. Jakmile příchozí zprávu obdrží, dekóduje její parametry a zavolá obslužnou funkci. Výsledek obslužné funkce zakóduje do formátu XDR, sestaví zprávu `reply` a předá ji knihovnám RPC k odeslání.



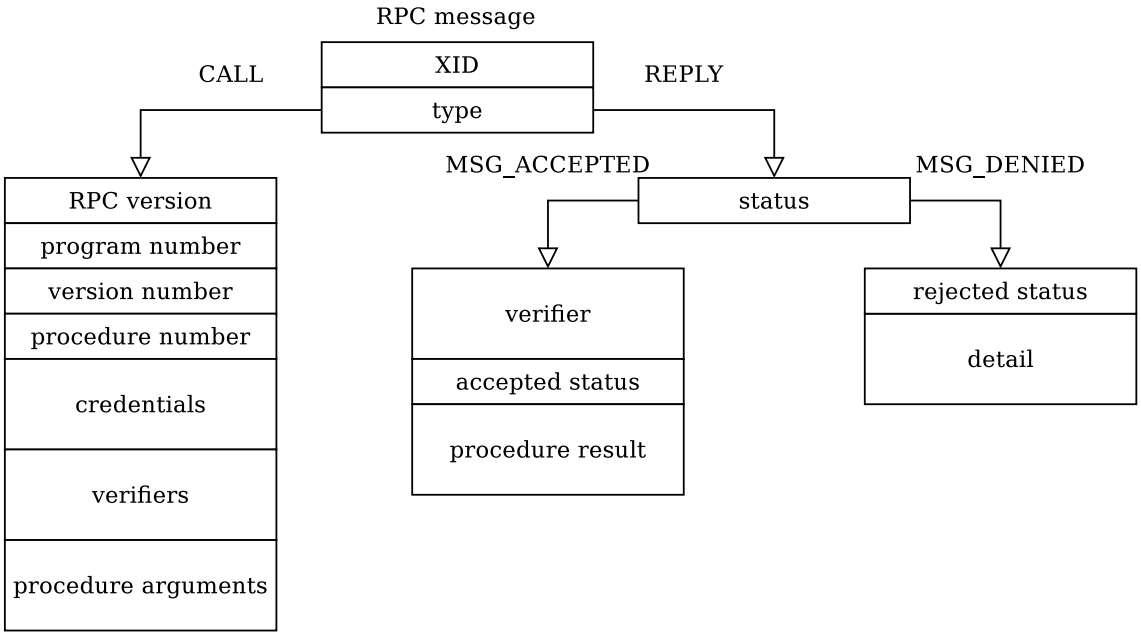
Obrázek 2.2: Komunikace v RPC

Obrázek 2.2 znázorňuje komunikaci v RPC, která probíhá v následujících krocích:

1. Spuštění serveru, server stub se registruje portmapperu.
2. Server stub naslouchá a čeká na požadavky.
3. Spuštění klienta, klient volá vzdálenou proceduru.
4. Klient stub se dotazuje portmapperu na číslo portu vzdálené služby.
5. Klient stub odesílá požadavek serveru.
6. Server stub požadavek přijímá a volá obslužnou funkci.
7. Obslužná funkce vrátila výsledek a server stub odesílá výsledek zpět.
8. Klient stub výsledek přijímá a předává jej klientskému programu.

2.3 RPC zprávy

V systému RPC se užívají dva typy zpráv a každý typ má jinou strukturu uložených dat. Pro volání RPC serveru se používá `CALL message` a server odpovídá zprávou typu `REPLY message`. Struktura jednotlivých zpráv je na obrázku 2.3.



Obrázek 2.3: Vnitřní struktura RPC zpráv

Oba typy RPC zpráv mají společnou část, která slouží k identifikaci a rozpoznání RPC zprávy. Hodnota `XID` je unikátní číslo relace a podle hodnoty `type` se určí následující tvar těla RPC zprávy. Hodnota `type` může nabývat pouze dvou hodnot: `CALL` pro volání a `REPLY` pro odpověď. Dále se těla jednotlivých typů zpráv liší.

2.3.1 `CALL message`

Hodnota `RPC version` udává verzi RPC protokolu a musí být vždy nastavena na hodnotu 2, jinak nebude zpráva akceptována a bude vráceno chybové hlášení `RPC_MISMATCH`. Hodnoty `program number`, `version number` a `procedure number` slouží k identifikaci vzdálené procedury. Pro ověření klienta je určena položka `credentials`, která v sobě uchovává autentizaci. Pole `verifiers` může být prázdné nebo obsahovat informace o autentizaci serveru vůči klientovi. Parametry předávané proceduře jsou zakódované v položce `procedure parameters` a server musí znát jejich přesnou strukturu.

2.3.2 `REPLY message`

První informace, která je specifická pro zprávu typu `REPLY`, je hodnota `status`. Na základě této hodnoty se rozhodne o následující struktuře zprávy. Položka `status` může nabývat pouze dvou hodnot: `MSG_ACCEPTED` nebo `MSG_DENIED`.

Odpověď s příznakem `MSG_ACCEPTED`

Každá zpráva, která byla akceptována, obsahuje pole `verifier`, které slouží pro zpětnou autentizaci. Další hodnotou je `accepted status`, která opět větví strukturu zprávy. Položka `accepted status` může nabývat šest hodnot: `SUCCESS`, `PROG_UNAVAIL`, `PROG_MISMATCH`, `PROC_UNAVAIL`, `GARBAGE_ARGS` a `SYSTEM_ERR`. Pokud má odpověď nastavenou `accepted status` na hodnotu `SUCCESS`, následující data tvoří výsledek volání vzdálené funkce. Jinak jsou následující data buď ignorována nebo obsahují informaci o minimální a maximální verzi programu.

Odpověď s příznakem `MSG_DENIED`

Pokud došlo k chybné autentizaci nebo nebyla verze protokolu kompatibilní, byla zpráva zamítnuta. Informuje nás o tom položka `rejected status`, která může nabývat hodnot `RPC_MISMATCH` a `AUTH_ERROR`. Následující informace ve struktuře zprávy uvádí detailnější informace o tom, proč nebyla zpráva akceptována v případě, že je hodnota položky `rejected status` rovna `AUTH_ERROR`. V opačném případě udává následující informace rozsah verzí RPC protokolu, s kterým dokáže serverovská aplikace pracovat.

2.3.3 Označování RPC zpráv

Pokud je přenos realizován nad streamovým přenosem dat (například TCP), používá RPC označování zpráv (RM, Record Marking), aby bylo možné se zotavit z případných chyb.

Záznam tvoří sekvenci 4 bytů. První bit v sobě nese informaci o tom, zda je RPC zpráva poslední – pokud je RPC zpráva poslední, je tento bit nastaven na hodnotu 1, jinak je vždy nastaven na hodnotu 0. Zbývající bity jsou vyhrazeny pro uložení délky RPC zprávy. K uložení délky zprávy je k dispozici 31 bitů. Z toho vyplývá, že maximální délka zprávy může být až $2^{32} - 1$ bytů. Ačkoliv jde o 4 bytovou sekvenci pro uložení dat, jak používá formát XDR, nejedná se o jeho reprezentaci.

2.4 Autentizace

Protokol RPC poskytuje autentizační mechanismus. Zabezpečení může být řešeno i na aplikační úrovni.

`CALL message` obsahuje dvě pole, která jsou vyhrazená pro autentizační údaje *credentials* (autentizace klienta vůči serveru) a *verifiers* (zpětná autentizace). `REPLY message` obsahuje pouze jedno pole pro autentizaci a jedná se o položku *verifier*. Mezi nejběžnější typ autentizace patří `AUTH_UNIX`.

```
enum auth_flavor {AUTH_NULL, AUTH_UNIX, AUTH_SHORT, AUTH_DES};
struct opaque_auth {
    auth_flavor flavor;
    opaque body<400>;
};
```

Obrázek 2.4: Autentizační struktura

Struktura dat uložených v hlavičce RPC zprávy je vidět na obrázku 2.4. První údaj specifikuje typ autentizace, podle kterého se určí následující uspořádání dat v hlavičce. Tělo autentizace je limitováno velikostí 400 B. Jak jsou následující data uspořádaná popisují následující podkapitoly, které detailněji popisují jednotlivé typy autentizací.

2.4.1 Autentizace `AUTH_NULL`

Pokud nám nezáleží na tom, kdo je klientem, můžeme použít prázdnou autentizaci. Můžeme ji použít například u portmapperu, protože pro dotazy klientů, kteří chtějí zjistit, na kterých portech naslouchá požadovaná služba, není ve většině případů potřeba žádné oprávnění. Co se týče nastavování a rušení mapovaných služeb portmapperem, musí mít klient za normálních okolností číslo portu rovno nebo menší číslu 1024, což znamená, že vzdálená aplikace má práva superuživatele a zároveň musí být na lokálním stroji – je ověřována IP adresa klienta. Je to jeden ze způsobů zabezpečení, které používá RPC. Toto standardní chování lze v obou případech potlačit.

```
struct opaque_auth {
    auth_flavor flavor = AUTH_NULL;
    opaque body {
        unsigned int length = 0;
    }
};
```

Obrázek 2.5: Tělo autentizační struktury `AUTH_NULL`

Na obrázku 2.5 je znázorněna struktura autentizace typu `AUTH_NULL`. Typ autentizace bude vždy nastaven na hodnotu `AUTH_NULL`. Tělo autentizační struktury obsahuje parametr délky těla, který bychom měli správně nastavit na nulovou hodnotu.

2.4.2 Autentizace `AUTH_UNIX`

Jak již bylo zmíněno v úvodu, autentizace typu `AUTH_UNIX` je nejpoužívanější autentizace v systému RPC, ačkoliv není příliš bezpečná a lze ji snadno podvrhnout. Stačí odposlechnout RPC zprávy klientů, kteří tento typ autentizace použijí. Pokud má potenciální útočník přístup k lokálnímu stroji, lze získat tyto údaje výpisem souboru `/etc/passwd`.

Další nevýhodou tohoto typu autentizace je nutná synchronizace uživatelů a skupin, například synchronizace údajů v souboru `/etc/passwd` mezi oběma stroji. Je nutné mít tyto data vždy aktuální, což může být v některých případech náročné.

Na obrázku 2.6 je zobrazená struktura autentizace typu `AUTH_UNIX`. Níže uvedené položky popisují význam jednotlivých proměnných ve struktuře.

`stamp` – libovolné číslo, které generuje volající klient.

`machinename` – název volajícího stroje (hostname).

`uid` – identifikační číslo uživatele.

`gid` – identifikační číslo skupiny.

`gids` – pole skupin, do kterých volající patří.

```
struct opaque_auth {
    auth_flavor flavor = AUTH_UNIX;
    opaque body {
        unsigned int stamp;
        string machinename<255>;
        unsigned int uid;
        unsigned int gid;
        unsigned int gids<16>;
    };
};
```

Obrázek 2.6: Tělo autentizační struktury `AUTH_UNIX`

U tohoto typu autentizace je doporučováno použít pro zpětné ověření (`verifier`) autentizaci typu `AUTH_NULL`. V případě, že server používá *autentizační cache*, může být vrácena v položce `verifier` zkrácená autentizace typu `AUTH_SHORT`, kterou se může klient v další komunikaci prokazovat namísto autentizací `AUTH_UNIX`. Tento způsob šetří systémové prostředky na obou stranách. Server může kdykoliv tento uchovávaný záznam zrušit. Pokud server záznam zrušil, klient obdrží při autentizaci chybové hlášení s příznakem `AUTH_REJECTEDCRED`. V tomto případě se může opět pokusit klient autentizovat původním `AUTH_UNIX`.

2.4.3 Autentizace `AUTH_DES`

Autentizace typu `AUTH_UNIX` trpí třemi zásadními problémy:

- Pojmenovávání strojů je příliš platformně závislé.
- Neexistují univerzální UID, GID.
- Není zde žádný ověřovatel, a proto lze *credentials* snadno podvrhnout.

DES autentizace se snaží tyto problémy řešit.

Pojmenovávání

První problém UNIXové autentizace je řešen nahrazením platformně závislým `machinename`, řetězcem, charakterizující klientskou stanici. Tento řetězec je známý jako `netname` (síťové jméno klienta). Řetězec `netname` by měl být vždy unikátní pro každého klienta v internetové síti a server tento řetězec smí použít pouze v případě identifikace klienta. Je na každém operačním systému na němž je implementováno DES, jakým způsobem zajistí unikátnost těchto jmen.

Ověřování

Narozdíl od autentizace typu `AUTH_UNIX`, má autentizace `AUTH_DES` ověřování. Ověřování spočívá v šifrování časového razítka. Server i klient zná klíč vygenerovaný pro konverzaci. Server může časové razítko vložené klientem dešifrovat a pokud je blízké reálnému času, pak je zašifrované správně a klient tudíž musí znát klíč, který byl vytvořen pro konverzaci. Jediný způsob, jak mohl klient správně zašifrovat časové razítko, je znalost tohoto klíče. Pokud klient správný klíč zná, pak musí jít o skutečného klienta, se kterým komunikujeme. Klíč pro konverzaci generuje klient a přenáší se v první RPC komunikaci bezpečným způsobem na server. Klíč je zašifrován pomocí dohodnutého klíče a rozšifrovat ho může jen účastník konverzace. Aby ověřování pomocí časových razítek mohlo fungovat, musí být klient i server časově zesynchronizován. K tomuto účelu může posloužit například NTP (Network Time Protokol). Pokud nelze synchronizaci tímto způsobem zajistit, je čas zesynchronizován s časem klienta. Ověřování identity serveru je řešeno tak, že server časové razítko klienta dešifruje, odečte jednu sekundu a opět zašifruje. Tím prokáže klientovi znalost klíče.

Přidělování *nicknames* a časová synchronizace

Při první komunikaci obdrží klient *nickname*, které pak může použít pro další komunikaci. Dojde tak k úspoře přenesených dat i procesorového času. Hodnota *nickname* je obvykle index do tabulky klientů, se kterými server komunikuje.

I když byly původně oba systémy časově synchronizovány, mohou se hodiny klienta a serveru rozejít. Pokud k tomu dojde, obdrží klient chybové hlášení s příznakem `RPC_AUTHERROR`. Chybu s příznakem `RPC_AUTHERROR` může klient obdržet i v případě, že je synchronizace v pořádku. Důvodem může být omezená tabulka klientů. Klient provede plnohodnotnou autentizaci (tak jak tomu bylo v první komunikaci), kde mu bude pravděpodobně přidělena nová hodnota *nickname*.

Distribuce klíče

Pro výměnu symetrického klíče, který slouží k zašifrování klíče pro ověřování účastníků komunikace, se používá algoritmus *Diffie-Hellman*. Společnost Sun Microsystems zvolila konstanty α a β pro výpočet klíčů.

$$\begin{aligned} PK(A) &= (\alpha^{SK(A)}) mod \beta \\ PK(B) &= (\alpha^{SK(B)}) mod \beta \end{aligned} \tag{2.1}$$

Nejprve si každý účastník komunikace vytvoří náhodně vygenerovaný soukromý klíč (SK). Podle rovnic 2.1 si klient i server vypočte veřejný klíč (PK) a vzájemně si jej vymění.

$$\begin{aligned} CK(A,B) &= (PK(B)^{SK(A)}) mod \beta \\ CK(A,B) &= (PK(A)^{SK(B)}) mod \beta \end{aligned} \tag{2.2}$$

Podle rovnic 2.2 si klient i server vypočtou společný klíč (CK), který použije klient k zašifrování klíče pro budoucí komunikaci. Klient pomocí společného klíče, který získal tímto způsobem, vygeneruje a zašifruje nový klíč, a pošle jej serveru. Server klíč rozšifruje, uloží si jej a dále už se používá tento nový klíč. Tím proběhla distribuce klíče serveru vygenerovaný klientem.

2.5 Jazyk pro popis RPC

Abychom mohli popisovat verze, procedury, datové struktury a pod., u RPC programů, bylo nutné vytvořit formální jazyk, který by tyto vlastnosti dokázal popsat. Jazyk pro popis RPC aplikací vznikl rozšířením jazyka pro popis XDR.

2.5.1 Specifikace jazyka

Jazyk RPC je identický jazyku pro popis XDR (kap. 3.2), který byl rozšířen o definici programu, verze a procedury.

```
program-def:
    "program" identifier "{"
    version-def
    version-def *
    "}" "=" constant ";"

version-def:
    "version" identifier "{"
    procedure-def
    procedure-def *
    "}" "=" constant ";"

procedure-def:
    type-specifier identifier "(" type-specifier
    ("," type-specifier )* ")" "=" constant ";"

proc-return:
    "void" | type-specifier

proc-firstarg:
    "void" | type-specifier
```

Obrázek 2.7: Rozšíření jazyka RPC

Poznámky specifikace:

- Klíčová slova `program` a `version` nelze použít jako identifikátory.
- Jméno a verze programu může být deklarována pouze jednou.
- Jméno procedury v rámci jedné verze programu může být deklarováno pouze jednou.
- Identifikátory, konstanty a datové typy jsou ve stejném jmenném prostoru.
- Konstanty, které definují program, verzi a proceduru, musí být vždy nezáporné.
- RPC kompilátory dovolují použít pouze jeden argument předávaný proceduře. Pokud potřebujeme předávat více hodnot, musíme je zabalit do struktury.

2.5.2 Praktická ukázka

V této kapitole si ukážeme a vysvětlíme popis velice jednoduché RPC aplikace. Na obrázku 2.8 je popsáný RPC program `RAND_PROG`, který má dvě verze `RAND_V1` a `RAND_V2`. Konstanta `RAND_VERS` udává nejvyšší verzi programu.

```
const RAND_VERS = 2;
struct range {
    int from;
    int to;
};
program RAND_PROG {
    version RAND_V1 {
        void RANDPROC_NULL(void) = 0;
        double RANDPROC_GEN(void) = 1;
    } = 1;
    version RAND_V2 {
        void RANDPROC_NULL(void) = 0;
        double RANDPROC_GEN(void) = 1;
        double RANDPROC_GEN2(range) = 2;
    } = 2;
} = 1;
```

Obrázek 2.8: Jednoduchý popis programu v jazyce RPC

První verze `RAND_V1` programu `RAND_PROG` má dvě procedury. Procedura `RANDPROC_NULL` slouží pouze k ověření spojení mezi klientskou a serverovskou aplikací. Druhá procedura `RANDPROC_GEN` slouží k vygenerování náhodného čísla v rozsahu $\langle 0,\ 1)$. Verze `RAND_V2` je rozšířená o možnost generovat náhodná čísla v uvedeném rozsahu a z důvodu kompatibility obsahuje obě funkce, které má předchozí verze.

Jak již bylo zmíněno v předchozí kapitole, kompilátory RPC neumožňují předávat proceduře více než jeden argument. Proto bylo nutné deklarovat nový datový typ `range`, který zajistí předání hodnot proceduře.

2.6 Nástroje pro generování RPC aplikací

V této kapitole si v krátkosti uvedeme nástroje pro generování `stub` RPC aplikací. Ukážeme si, jak vygenerovat `stub` pro klienta i server. Budeme k tomu potřebovat popis rozhraní RPC, k tomu nám poslouží deklarace rozhraní, kterou jsme si uvedli v předchozí kapitole 2.5.2. Program si pojmenujeme názvem „randprog“, který se promítne do názvu souboru.

Pro vytvoření RPC aplikace použijeme utilitu `rpcgen`. Budeme k tomu potřebovat specifikační soubor, který definuje RPC rozhraní, klientskou aplikaci, která volá vzdálené procedury a serverovskou aplikaci, která bude implementovat vzdálené procedury. Utilita `rpcgen` nám vygeneruje celkem čtyři soubory, které musí být součástí kompilace:

1. `randprog.h` – hlavičkový soubor, který je nutné začlenit do klientské i serverovské aplikace. Obsahuje deklarace datových typů a podobně.
2. `randprog_xdr.c` – pro kódování dat do formátu XDR.
3. `randprog_clnt.c` – klient `stub`.
4. `randprog_svc.c` – server `stub`.

Následující kapitoly 2.6.1 a 2.6.2 popisují co musíme implementovat do klientských a serverovských aplikací. Vzhledem k omezenosti rozsahu této práce, popisuji jen ty nejzákladnější rysy. Pro zájemce, kteří chtějí proniknout hlouběji, jsem vytvořil ukázkovou RPC aplikaci, která je umístěna na elektronickém médiu této práce. Zdrojové kódy umístěné níže jsou vystřižené z této aplikace a patřičně zjednodušené pro naše účely.

2.6.1 Klient

Nejprve je potřeba zavolat funkci `clnt_create`, která vytvoří spojení mezi serverem a klientem. Dále už můžeme volat jednotlivé funkce. Názvy funkcí jsou tvořeny `funkce_verze`. Například, pokud budeme chtít volat funkci `RANDPROC_GEN` verze 2, budeme volat funkci `double *randproc_gen_2(void *, CLIENT *)`.

Na obrázku 2.9 je krátká ukázka klientského zdrojového kódu:

```
clnt = clnt_create("localhost", RAND_PROG, RAND_VERS, "tcp");
result = *randproc_gen_2(NULL, clnt);
result = *randproc_gen2_2(&range, clnt);
```

Obrázek 2.9: Klientský zdrojový kód

2.6.2 Server

Z pohledu serveru je situace ještě jednodušší. Jediné, co musíme udělat, je deklarovat obslužné funkce. Jména funkcí mají tvar `funkce_verze_svc`.

Na obrázku 2.10 je krátká ukázka serverovského zdrojového kódu:

```
double *randproc_gen_2_svc(void *argp, struct svc_req *rqstp) {
    result = rand()/(RAND_MAX+1.0);
    return &result;
}
```

Obrázek 2.10: Serverovský zdrojový kód

Kapitola 3

XDR – datová reprezentace

Pro popis a kódování dat v systému RPC i portmap protokolu se používá standard XDR. XDR [9], [4], [7] (eXternal Data Representation) je standard určený pro popis a kódování dat, který je platformně nezávislý a slouží pro přenos dat v síti. Tato kapitola popisuje datové typy používané v XDR, základní rysy jazyka XDR a poslední část se zabývá datovým rozborem z praktického pohledu.

3.1 Datové typy

Standard XDR definuje celou řadu datových typů. Zde jsou uvedeny pouze nejpoužívanější datové typy. V tabulce 3.1 jsou základní datové typy rozděleny do tříd.

Třída	**Datové typy v XDR**
Celočíselné datové typy	[unsigned] [hyper] int
Reálné datové typy	float, double
Strukturované datové typy	string, enum, struct, union
Speciální datové typy	void, const

Tabulka 3.1: Nejpoužívanější datové typy

Vybrané datové typy jsou popsány v následující části textu.

3.1.1 `void`

Ve standardu XDR je datový typ `void` reprezentován nulovou délkou. Do XDR streamu se tedy nezapisuje vůbec. Datový typ `void` se používá k popisu operací, které nemají žádný vstup nebo výstup. Datový typ `void` tedy nemá vlastní identifikátor.

3.1.2 `bool`

Datový typ `bool` je v jazyce XDR často používaným datovým typem a je ve své podstatě identický výčtovému typu, který má deklarované dvě hodnoty `FALSE` a `TRUE`. Na obrázku 3.1 je ukázaná jeho deklarace včetně jeho identického zápisu jako výčtového typu.
Při uložení proměnné typu `bool` do datového streamu XDR se velikost streamu zvýší o 4 byte.

```
bool identifier;
enum {FALSE = 0, TRUE = 1} identifier;
```

Obrázek 3.1: Deklarace `bool`

3.1.3 `integer`, `unsigned integer`

Celočíselný datový typ `integer` se znaménkem má číselný rozsah $\langle -214748364, 214748364 \rangle$ a `unsigned integer` má číselný rozsah $\langle 0,\ 4294967295 \rangle$. Velikost integeru je 32 bitů a zabere tedy jednu jednotku ve formátu XDR.

3.1.4 `float`, `double`

Kódování vychází ze standardu IEEE[1] 754-1985 a obsahuje tři informace. První údaj (S) nese informaci o znaménku čísla a má velikost 1 bit. Druhým údajem (E) je exponent o dvojkovém základu, který je reprezentován 8 bity pro `float` a 11 bity pro `double`. Posledním údajem (F) je mantisa, která má základ rovněž dva a je reprezentována 23 bity pro `float` a 52 bity pro `double`. Výsledné číslo lze vyjádřit vztahem:

$$(-1)^S \cdot 2^{E-Bias} \cdot 1.F$$

Hodnota `Bias` je pro `float` nastavena na 127 a pro `double` na 1023. Velikost čísla `float` zakódovaného do formátu XDR zabere 32 bitů, kdežto u čísla `double` již 64 bitů.

3.1.5 Pole `array`

Standard XDR podporuje tři typy polí:

- Pole s předem známou a fixovanou délkou.
- Omezené pole s předem neznámou délkou.
- „Neomezené“ pole s předem neznámou délkou.

Všechny typy polí jsou indexovány od $\langle 0,\ n-1 \rangle$, kde n udává počet prvků v poli. Na obrázku 3.2 je uvedena deklarace všech typů polí. První řádek ukazuje deklaraci s předem známou a pevně stanovenou velikostí pole. Na druhém řádku je uvedena deklarace pole s limitovaným počtem prvků a poslední řádek deklaruje „neomezené“ pole s předem neznámým počtem prvků.

Výsledná délka po převedení do formátu XDR bude u pole s předem známou velikostí $ns + r$, kde n je počet prvků v poli a s je velikost použitého datového typu ve formátu XDR. Proměnná r udává počet zarovnávacích bytů tak, aby platilo $(ns + r) \bmod 4 = 0$. U polí s předem neznámou velikostí musí být předřazená informace o tom, kolik prvků je v poli obsaženo. To znamená, že výsledná velikost pole po převedení do formátu XDR bude $4 + ns + r$. Prvků v poli může být maximálně $2^{32} - 1$.

[1] IEEE Standard for Binary Floating-Point Arithmetic

```
type-name identifier[];
type-name identifier<m>;
type-name identifier<>;
```

Obrázek 3.2: Deklarace pole v jazyku XDR

3.1.6 `string`

Datový typ `string` je ve své podstatě pole ASCII znaků, u kterého známe počet prvků. Jednotlivé znaky jsou indexovány od $\langle 0,\ n-1 \rangle$, kde n udává počet prvků v poli. U řetězce si můžeme zvolit maximální délku m. Na obrázku 3.3 je ukázka deklarace proměnné datového typu `string`. První řádek deklaruje řetězec, jehož délka je maximálně m znaků. Druhý řádek deklaruje „neomezený“ řetězec, který může mít až $2^{32} - 1$ znaků.

```
string identifier<m>;
string identifier<>;
```

Obrázek 3.3: Deklarace řetězce v jazyku XDR

Ve formátu XDR je řetězec reprezentován celočíselným, bezznaménkovým `integer`, který udává délku řetězce n a za ním následuje samostatný řetězec. Výsledná velikost řetězce po zakódování do formátu XDR tedy bude $4+n+r$, kde r je počet zarovnávacích bytů a musí platit $(n+r) \bmod 4 = 0$.

3.2 Základní specifikace jazyka XDR

Tato část kapitoly popisuje základní rysy jazyka XDR. V první části je seznam klíčových slov a v druhé syntaktická pravidla.

3.2.1 Klíčová slova

Klíčová slova jsou vyhrazená slova jazyka a nelze je použít pro jiné účely, například jako jméno identifikátoru.

Zde je uveden seznam klíčových slov jazyka XDR:

```
bool, case, const, default, double, quadruple, enum, float,
hyper, int, opaque, string, struct, switch, typedef, union,
unsigned, void.
```

3.2.2 Syntaktická pravidla

Deklarace identifikátoru

Deklarovat identifikátor je možné jedním ze způsobů, které jsou uvedené na obrázku 3.4, kde `value` je konstanta nebo identifikátor. Konstanta může být oktalová, decimální nebo hexadecimální číslice.

```
type-specifier identifier
type-specifier identifier "[" value "]"
type-specifier identifier "<" [ value ] ">"
"opaque" identifier "[" value "]"
"opaque" identifier "<" [ value ] ">"
"string" identifier "<" [ value ] ">"
type-specifier "*" identifier
"void" identifier
```

Obrázek 3.4: Deklarace identifikátoru

Identifikátory mohou obsahovat alfanumerické znaky a podtržítko. Názvy identifikátorů jsou *case-sensitive*. Názvy identifikátorů nesmí být shodné s názvy klíčových slov, které jsou uvedeny v kapitole 3.2.1.

Deklarace strukturovaných datových typů

Na obrázku 3.5 jsou uvedena syntaktická pravidla pro deklaraci struktury, výčtového typu a unionu. Za povšimnutí stojí především datový typ `union`, který se syntakticky zásadně liší od jazyka C.

```
"struct" "{"
    ( declaration ";" )
    ( declaration ";" )*
"}"
"enum" "{"
    ( identifier "=" value )
    ( "," identifier "=" value )*
"}"
"union" "switch" "(" declaration ")" "{"
    case-spec
    case-spec *
    [ "default" ":" declaration ";" ]
"}"
```

Obrázek 3.5: Deklarace složeného datového typu

3.3 Datový rozbor

V této kapitole se podíváme na celou problematiku XDR z praktického pohledu. Nejprve si nadefinujeme formát dat, který je napsaný v jazyce XDR. Tuto nově nadeklarovanou datovou strukturu naplníme daty a převedeme do XDR streamu, který následně zobrazíme z datového pohledu v hexadecimálním editoru.

3.3.1 Deklarace datové struktury

Na obrázku 3.6 je nadeklarovaná struktura, která popisuje základní vlastnosti souboru unixového operačního systému. Pro názornost jsem záměrně použil dva typy řetězců, abychom si ukázali odlišnost kódování do XDR streamu.

```
const MAX_FILE_LEN = 255;
const PERM_LENGTH = 10;
struct file {
    string filename<MAX_FILE_LEN>;
    unsigned int uid;
    unsigned int gid;
    string permission[PERM_LEN];
};
```

Obrázek 3.6: Deklarace struktury

3.3.2 Definice dat

Nadefinujeme strukturu dat tak, jak je tomu uvedeno na obrázku 3.7.

```
filename = "filename.ext";
uid = 1105;
gid = 1034;
permission = "drwxr-xr-x";
```

Obrázek 3.7: Naplnění struktury daty

Nyní bude datová struktura převedena do XDR streamu a v následující kapitole zobrazen obsah XDR streamu v hexadecimálním editoru.

3.3.3 Analýza XDR streamu

V tabulce 3.2 je ukázán výsledný XDR stream zobrazený v hexadecimálním formátu spolu s komentáři, ve kterých je zobrazen jejich charakter a obsah datového bloku.

Adresa	HEX	Komentář
00	00 00 00 0C	délka názvu souboru = 12
04	66 69 6c 65	text = „file“
08	6e 61 6d 65	text = „name“
12	2e 65 78 74	text = „.ext“
14	00 00 04 51	uid = 1105
18	00 00 04 0A	gid = 1034
22	64 72 77 78	text = „drwx“
24	72 2d 78 72	text = „r-xr“
28	2d 78 00 00	text = „-x“ + 2 B zarovnání

Tabulka 3.2: XDR stream

První datový blok v zobrazeném XDR streamu obsahuje délku názvu souboru. Délka řetězce je uváděna proto, že dopředu není známo, kolik znaků bude uchovávat. Následující tři bloky obsahují samostatný řetězec. Další dva bloky jsou celočíselné datové typy `unsigned int`, které nesou informace o *uid* a *gid*. Další typ řetězce je možné vidět v následujících blocích. Tento řetězec má vždy délku 10 znaků, a proto není nutné uchovávat informaci o jeho délce. Protože je délka řetězce rovna číslu 10 a nesplňuje podmínku $(n + r) \bmod 4 = 0$, pro $r = 0$, proto bylo nutné data zarovnat o $r = 2\,\mathrm{B}$.

Kapitola 4

Portmap protokol

Cílem této práce je implementace pluginu portmap, který využívá ke své komunikaci s klientskými i serverovskými aplikacemi RPC a portmap protokol popsaný v této kapitole. Portmap protokol [8] mapuje čísla programů a verzí na specifické čísla portů transportní vrstvy. První část kapitoly se věnuje popisu portmapperu v jazyce RPC a druhá část popisuje jednotlivé procedury portmap protokolu.

4.1 Popis portmapperu v jazyce RPC

Portmapper je program s pevně přiděleným číslem programu 100000 organizací IANA (Internet Assigned Numbers Authority) a jeho verze je vždy rovna číslu dva. Jedná se o RPC server, který má 6 procedur, které se starají o mapování programů na čísla portů.

```
const PMAP_PORT = 111;
program PMAP_PROG {
    version PMAP_VERS {
        void PMAPPROC_NULL(void) = 0;
        bool PMAPPROC_SET(mapping) = 1;
        bool PMAPPROC_UNSET(mapping) = 2;
        unsigned int PMAPPROC_GETPORT(mapping) = 3;
        pmaplist PMAPPROC_DUMP(void) = 4;
        call_result PMAPPROC_CALLIT(call_args) = 5;
    } = 2;
} = 100000;
```

Obrázek 4.1: Popis procedur portmapperu v jazyce RPC

Na obrázku 4.1 je popsána základní struktura portmapperu v jazyce RPC (kap. 2.5). Procedura `PMAPPROC_NULL` slouží pro ověření spojení. Pomocí procedury `PMAPPROC_SET` registrujeme a pomocí `PMAPPROC_UNSET` odregistrujeme serverovskou RPC aplikaci. Pro klientské RPC aplikace je určena zejména procedura `PMAPPROC_GETPORT`, jejíž pomocí si klient zjišťuje na jakém portu naslouchá její serverovská aplikace. Procedury `PMAPPROC_DUMP` slouží k výpisu všech mapovaných služeb. Pomocí procedury `PMAPPROC_CALLIT` lze volat procedury

RPC serveru bez znalostí jeho portu. Detailnější popis procedur si uvedeme v kapitole 4.2.

V datové struktuře `mapping`, která je zobrazena na obrázku 4.2, uchováváme informace o číslu programu, verzi, protokolu a portu, na kterém server naslouchá. Portmapper tyto struktury, nesoucí informace o jednotlivých programech, ukládá v jednosměrně vázaném lineárním seznamu `pmaplist`.

```
struct mapping {
    unsigned int prog;
    unsigned int vers;
    unsigned int prot;
    unsigned int port;
};
```

Obrázek 4.2: Popis datové struktury pro uchování informací

Následující dvě datové struktury `call_args` a `call_result` na obrázku 4.3 jsou určeny pouze k předávání parametrů či výsledku procedury `PMAPPROC_CALLIT`.

```
struct call_args {
    unsigned int prog;
    unsigned int vers;
    unsigned int proc;
    opaque args<>;
};
struct call_result {
    unsigned int port;
    opaque res<>;
};
```

Obrázek 4.3: Popis datových struktur procedury `PMAPPROC_CALLIT`

Hodnoty `prog`, `vers` a `proc` ve struktuře `call_args` charakterizují serverovský RPC program – tomuto programu se bude volání předávat. Datová struktura `args` je pro portmapper neviditelná, respektive nezná strukturu uložených dat – toto je věc klienta a serveru.

Odpověď volání procedury `PMAPPROC_CALLIT` reprezentuje struktura `call_result`. Obsahuje port volané služby a pro portmapper neznámou strukturu dat `res`, která nese výsledek volání. Formát dat je opět záležitostí klienta a serveru.

4.2 Procedury

V této části jsou popsány jednotlivé procedury portmap protokolu.

4.2.1 PMAPPROC_NULL

Tato procedura nic nevykonává a je zde zavedená z konvenčních důvodů. Nemá žádné vstupní parametry a nic nevrací. Slouží pouze pro ověření spojení – pokud je spojení mezi klientskou a serverovskou aplikací v pořádku, vrátí se kladně vyřízená zpráva, která neobsahuje žádný výsledek.

4.2.2 PMAPPROC_SET

V případě serverovské aplikace se každá aplikace při svém spuštění nejprve registruje portmapperu, který běží na lokálním stroji a naslouchá na portu 111. Program odešle portmapperu požadavek o jeho registraci, ve kterém je uvedeno číslo programu, verze, portu a protokolu. Číslo portu si serverovská aplikace volí sama. Tato procedura vrátí `TRUE` v případě úspěchu. Pokud je z nějakého důvodu žádost zamítnuta, vrátí procedura `FALSE`. K zamítnutí může dojít například v případě, že je tato aplikace již registrována.

4.2.3 PMAPPROC_UNSET

Dříve, než se serverovský program registruje portmapperu, je vhodné odeslat žádost o zrušení registrace stávajícího programu. Předávané parametry jsou identické volání procedury `PMAPPROC_SET`. Informace o protokolu a čísla portu nemusí být vyplněny, protože budou ignorovány. Procedura vrací `TRUE`, pokud byl program odregistrován, jinak vrací `FALSE`.

4.2.4 PMAPPROC_GETPORT

Očekávané parametry při volání této procedury jsou vyplněné položky `prog` `vers` a `prot` ve struktuře `mapping`. Výsledkem úspěšného volání je číslo portu, na kterém naslouchá požadovaná služba. Pokud není služba registrována, vrací procedura `PMAPPROC_GETPORT` v čísle portu hodnotu 0.

4.2.5 PMAPPROC_DUMP

Procedura `PMAPPROC_DUMP` nemá žádné vstupní parametry a slouží k výpisu všech registrovaných služeb portmapperem, včetně svých služeb. Vrací lineární seznam struktur `mapping`.

4.2.6 PMAPPROC_CALLIT

Tato procedura umožňuje komunikovat se serverovskou aplikací bez znalostí portu, na kterém naslouchá. Argumentem procedury je datová struktura `call_args`, která obsahuje položky `prog`, `vers` a `proc`, které identifikují vzdálenou službu. Argument `args` je určen pro argumenty vzdálené procedury. Protokol, na kterém vzdálený server naslouchá, není důležité předávat, protože komunikace mezi portmapperem a vzdálenou službou proběhne vždy přes protokol UDP. Jestliže vzdálená služba nepodporuje UDP přenos, dojde k chybě. Pokud dojde k chybě, neodesílá portmapper žádnou odpověď.

Procedura vrací datovou strukturu `call_result`, ve které je obsaženo číslo portu, na kterém vzdálená služba naslouchá a výsledek vzdálené služby v položce `res`.

Kapitola 5

BusyBox

Tato kapitola v krátkosti seznamuje s projektem BusyBox, na který byl v rámci této práce implementován BusyBox. Popisuje také jeden z možných způsobů, jak BusyBox získat a jak jej nainstalovat.

5.1 O BusyBoxu

BusyBox [1] [6] [3] je ve své podstatě svobodný program pod licencí GNU/GPL, který je určen pro unixové operační systémy. Integruje standardní unixové příkazy, podobné jako v *GNU Core Utilities* do jednoho celku. Jeho koncepce je modulární a lze si sestavit funkční balíček podle stanovených požadavků. Tím lze optimalizovat i výslednou velikost spustitelného programu na minimum.

Cílem BusyBoxu je vytvořit malý univerzální program s malými hardwarovými nároky, který ve spolupráci s Linuxovým jádrem poskytne použitelný operační systém, určený převážně pro vestavěné systémy.

Původní verzi BusyBoxu vytvořil Bruce Perens v roce 1996 jako snahu vytvořit kompletní bootovatelný systém, který by se vlezl na jednu disketu a bude sloužit jako instalátor a záchranná disketa systému GNU/Linux Debian. V roce 1998 tento projekt převzal Erik Andersen, který začal projekt upravovat pro všeobecné použití. Vytvořil webové stránky a repozitáře pro týmovou spolupráci. Tím se BusyBox otevřel veřejnosti a našel si spoustu příznivců, kteří se podílejí na jeho vývoji. V současné době spravuje BusyBox Denis Vlasenko. Projekt vyspěl a začal se používat v instalátorech mnoha distribucí systému GNU/Linux a své uplatnění nalezl také hlavně ve vestavěných systémech.

5.2 Instalace a použití

V první části této sekce bude ukázáno kde a jak lze BusyBox získat. Další část sekce je věnována základním rysům konfigurace. Třetí část stručně popisuje důležité parametry pro kompilaci projektu a poslední část popisuje instalaci BusyBoxu na médium.

5.2.1 Získání BusyBoxu

Balík BusyBox lze získat třemi různými způsoby. Lze získat jak již nakonfigurovanou a překompilovanou verzi, tak zdrojové kódy s tím, že si musíme BusyBox nakonfigurovat sami. Protože je druhý postup univerzálnější, budeme se jím dále zabývat.

Stáhneme aktuální verzi BusyBoxu a provedeme dekompresi:

```
# BUSYBOX=busybox-1.18.4
# wget http://www.busybox.net/downloads/${BUSYBOX}.tar.bz2
# tar -xjf ${BUSYBOX}.tar.bz2 && cd ${BUSYBOX}
```

5.2.2 Konfigurace

Dříve, než zahájíme kompilaci, musíme balík BusyBox nakonfigurovat. Veškerá manipulace s balíkem BusyBox je v tomto instalačním kroku prováděná prostřednictvím příkazu `make`. Nachází se zde více možností, a proto si uvedeme základní argumenty, které přijímá `make` BusyBoxu.

`make help` – Zobrazí všechny argumenty a jejich popis.

`make defconfig` – Nastaví defaultní hodnoty v nastavení balíčku.

`make allyesconfig` – Vybere všechny možnosti a nástroje, které balíček poskytuje.

`make allnoconfig` – Zruší všechny možnosti a nástroje.

`make oldconfig` – Používá se v případě, že jsme upgradovali verzi BusyBoxu a máme k dispozici starý konfigurační soubor `.config`. Volby se zkontrolují a pokud byly přidány nové volby do BusyBoxu, bude uživatel dotázán na jejich nastavení. Tyto dotazy lze přeskočit klávesou enter. Pokud tak učiníme, nastaví se jejich defaultní hodnota. Tuto volbu je také vhodné použít, pokud budeme soubor `.config` upravovat ručně, abychom udrželi nastavení v konzistentním stavu.

`make menuconfig` – Zobrazí grafické rozhraní, pro konfiguraci balíku BusyBox. Grafické rozhraní využívá knihovny `ncurses` a musí být nainstalovány v systému.

Pro konfiguraci spustíme grafické rozhraní:

```
# make menuconfig
```

5.2.3 Kompilace

Pokud chceme balík BusyBox používat ve vestavěném systému nebo někde, kde se nenacházejí knihovny systému nebo chceme, aby se binární spustitelný soubor zbavil veškerých závislostí, musíme jej překompilovat staticky. To znamená, aby binární spustitelný soubor obsahoval všechen kód, který používá a nepoužíval sdílené systémové knihovny. Tuto volbu zajistíme buď v grafickém režimu v nastavení `Busybox Settings -> Build Options -> Build BusyBox as a static binary`, a nebo upravíme konfigurační soubor `.config` ručně a nastavíme hodnotu `CONFIG_STATIC=y`.

Pokud máme v systému nainstalován *cross toolchain*, můžeme kompilovat BusyBox i pro jiné platformy. Prefix pro platformu lze nastavit buď jednorázově jako parametr pro `make` a to `CROSS_COMPILE=prefix` a nebo permanentně a to buď v konfiguračním souboru `.config` nastavením hodnoty `CONFIG_CROSS_COMPILER_PREFIX=prefix` a nebo v grafickém menu v nastavení `Busybox Settings -> Build Options -> Cross Compiler Prefix`.

Kompilaci pro architekturu x86_64 provedeme takto:

```
# make CROSS_COMPILE=x86_64-pc-linux-gnu-
```

5.2.4 Instalace

Instalace se rovněž provadí prostřednictvím `make` známým argumentem `install`. Nejprve však musíme nastavit cestu, kam chceme BusyBox instalovat. Cestu pro instalaci je možné opět nastavit třemi způsoby: v grafickem rozhraní, v konfiguračním souboru `.config`, a předáním parametru příkazu `make`. My si zde uvedeme pouze poslední způsob, protože jde o stejný princip jako v předchozích krocích. Předávaný parametr příkazu `make` je `CONFIG_PREFIX=path`, který nastavuje cestu pro instalaci. Cesta může být relativní i absolutní.

Instalace na medium:

```
# BB_PATH=/media/usb
# make CONFIG_PREFIX=${BB_PATH} install
```

Protože je výsledný binární soubor pouze jeden, je několik možností, jak BusyBox řeší volání jednotlivých appletů. BusyBox nabízí vazby s applety:

- neinstaluje vazby
- symbolickým odkazem (symlink)
- pevným odkazem (hardlink)
- skriptem (script wrappers)

První případ neinstaluje žádné vazby a jednotlivé moduly lze spouštět jako parametry binární aplikace BusyBox. Například, chceme-li volat příkaz `modprobe`, předáme jméno modulu binární aplikaci jako parametr `busybox modprobe`. Ostatní typy instalace umožňují volat moduly běžným způsobem.

Ukázka spouštění modulů všemi možnými způsoby:

```
# cd ${BB_PATH}
# PATH="‘pwd‘/bin:${PATH}"
# busybox lsmod
# lsmod
```

První dva řádky přidají cestu do globální proměnné `${PATH}`, abychom nemuseli psát cestu k binární aplikaci BusyBox. Třetí řádek ukazuje jak volat modul `lsmod` v případě, že jsme při instalaci zvolili instalaci bez vazeb. Protože jsme do proměnné `${PATH}` vložili cestu k BusyBoxu na začátek, proběhne při volání `lsmod` standardním způsobem volání utility `lsmod` z balíku BusyBox.

Kapitola 6

Návrh a implementace portmapu pro BusyBox

Tato kapitola je nejdůležitější částí této práce a popisuje specifické rysy návrhu a implementace pluginu portmap. První část kapitoly seznamuje s navrženými vstupními parametry pluginu portmap. Následující část kapitoly seznamuje s návrhem základní struktury a principem funkčnosti programu. Další část kapitoly popisuje návrh a implementaci datových struktur. Následující část kapitoly popisuje, jak byl navržen systém pro obsluhu požadavků a poslední část kapitoly vysvětluje, jak byl navržen mechanismus, který má za úkol zvýšit bezpečnost celého pluginu.

6.1 Navržené vstupní parametry

Plugin portmap přijímá čtyři vstupní parametry uvedené na obrázku 6.1. První parametr `-n` umožňuje spustit plugin portmap na popředí. Tato volba je vhodná především pro ladící účely nebo v případě, že má být plugin spuštěn jen krátkou dobu. Vstupní parametr `-n` si vyžádal správce BusyBoxu.

```
-n          Don't daemonize
-h ADDR     Listen on ADDR
-p          Allow set or unset from port >= 1024
-r          Allow set or unset from any host
```

Obrázek 6.1: Přepínače pluginu portmap

Druhý parametr `-h ADDR` umožňuje zadat adresu rozhraní, na kterém bude plugin portmap naslouchat – například, pokud víme, že budou služby RPC poskytovány jen v rámci lokální stanice, je vhodnější naslouchat pouze na virtuálním síťovém rozhraní *loopback* z důvodu vyšší bezpečnosti. Parametry `-p` a `-r` umožňují tzv. mód „insecure“, kdy je umožněno v prvním případě akceptovat požadavky na mapování a požadavky na zrušení mapování lokálním RPC aplikacím, které nemají dostatečné oprávnění. Druhá volba navíc umožňuje provádět tyto požadavky i vzdáleným klientům. Poslední dvě volby se z bezpečnostních důvodů důrazně nedoporučují používat.

6.2 Návrh základní kostry programu

Algoritmus 1 napsaný v pseudokódu vyjadřuje pouze obecnou strukturu hlavní kostry pluginu portmap. Jsou zde zanedbány vstupní parametry `-n`, `-p` a `-r`, inicializace globálních proměnných, nastavení logování a podobně. Detailnější popis procesu spouštění je uveden níže.

Algoritmus 1: Základní algoritmus pluginu portmap

```
   Input: adresa rozhraní, na kterém bude portmap naslouchat
 1 begin
 2     if adresa nebyla zadána then
 3         adresa rozhraní je INADDR_ANY
 4     end
 5     vytvoř schránky SOCK_DGRAM a SOCK_STREAM
 6     if schránky se nepodařilo vytvořit then
 7         vypiš chybové hlášení a ukonči program
 8     end
 9     schránky spoj s portem 111
10     if schránky se nepodařilo spojit then
11         vypiš chybové hlášení a ukonči program
12     end
13     naslouchej na definovaném rozhraní
14     registruj RPC službu
15     if registrace služby se nezdařila then
16         vypiš chybové hlášení a ukonči program
17     end
18     while čekej na příchozí požadavek do
19         obslož požadavek
20     end
21 end
```

Při spuštění pluginu portmap nejprve dochází k vyhodnocení vstupních přepínačů a pak se pomocí funkce `bb_daemonize_or_rexec` (pokud nebyl zadán přepínač `-n`) spustí jako *daemon*. Pokud nebyla zadána adresa síťového rozhraní, na kterém má plugin portmap naslouchat, je implicitně zvolena adresa `INADDR_ANY`. To znamená, že se bude naslouchat na všech dostupných síťových rozhraních. Pokud dojde v průběhu startu k chybě, zapíše se chybové hlášení do systémového logu a ukončí se běh pluginu. Plugin portmap vytvoří dvě BSD schránky typu `SOCK_DGRAM` a `SOCK_STREAM`, spojí je s portem 111 a začne naslouchat na tomto portu zvoleného síťového rozhraní. Následně proběhne vytvoření nové RPC služby pomocí funkce `svcudp_create` pro schránku typu `SOCK_DGRAM` a `svctcp_create` pro schránku typu `SOCK_STREAM`. Dalším krokem je provedena registrace RPC služby, kterou zajistí funkce `svc_register`. Pokud proběhla registrace úspěšně, je zavolána funkce `svc_run` a plugin portmap čeká na příchozí požadavky. Obsluha požadavku bude detailněji popsána v podkapitole 6.4. Použité funkce s prefixem `svc` jsou součástí knihoven RPC.

6.3 Návrh datových struktur

Pro uchovávání informací o jednotlivých RPC programech postačuje uchovávat čtyři informace: číslo programu, verze programu, typ protokolu, přes který RPC program komunikuje a port, na kterém naslouchá. Struktura, která uchovává informace o RPC programech, je uvedena na obrázku 6.2.

```
struct pmap {
    uint32_t pm_prog;
    uint32_t pm_vers;
    uint32_t pm_prot;
    uint32_t pm_port;
};
```

Obrázek 6.2: Datová struktura pro uchovávání informací o RPC programech

Vzhledem k tomu, že navržená datová struktura pro uchovávání informací o RPC programech byla identická se strukturou, kterou poskytují systémové knihovny portmap protokolu, byla proto v pluginu portmap použitá struktura z knihoven systému. Pro komunikaci s klientem jsou použity funkce knihoven RPC a pro kódování dat funkce z knihoven XDR. Proto je důležité, aby byly datové struktury jednotné.

```
struct pmaplist {
    struct pmap     pml_map;
    struct pmaplist *pml_next;
};
```

Obrázek 6.3: Datová struktura, která uchovává seznam mapovaných programů

Pro uchovávání seznamu registrovaných RPC aplikací používá plugin portmap jednosměrně vázaný lineární seznam, uvedený na obrázku 6.3. V tomto seznamu jsou ukládány úspěšně mapované RPC služby včetně služeb, které poskytuje plugin portmap. Každý záznam v seznamu je unikátní. Pokud má například RPC program dvě verze a podporuje transportní protokol UDP i TCP, budou v seznamu uloženy čtyři položky.

6.4 Návrh obsluhy požadavků

Tato část kapitoly popisuje navržený mechanismus pro obsluhu klientských požadavků. V první části je popsán hlavní rozhodovací blok, který zpracuje volání, zavolá příslušnou funkci a zajistí vrácení výsledku zpět klientovi. V druhé části jsou stručným způsobem popsány jednotlivé procedury.

6.4.1 Hlavní rozhodovací blok

Nejdůležitější částí pro obsluhu klientských požadavků je *callback* funkce `pmapproc`. Jedná se o funkci, která byla předána jako parametr funkci `svc_register` při registraci RPC služby. Funkce `pmapproc` je volána z funkce `svc_run`, která čeká na příchozí požadavky.

```
switch (rqstp->rq_proc) {
    case PMAPPROC_NULL:
        svc_sendreply(xprt, (xdrproc_t)xdr_void, NULL);
        break;
    case PMAPPROC_SET:
        res = (unsigned int)pmapproc_set(&pmap);
        svc_sendreply(xprt, (xdrproc_t)xdr_bool, &res);
        break;
    case PMAPPROC_UNSET:
        res = (unsigned int)pmapproc_unset(&pmap);
        svc_sendreply(xprt, (xdrproc_t)xdr_bool, &res);
        break;
    case PMAPPROC_GETPORT:
        res = pmapproc_getport(&pmap);
        svc_sendreply(xprt, (xdrproc_t)xdr_u_int, &res);
        break;
    case PMAPPROC_DUMP:
        svc_sendreply(xprt, (xdrproc_t)xdr_pmaplist, &G.pl);
        break;
    default:
        svcerr_noproc(xprt);
}
```

Obrázek 6.4: Rozhodovací blok pro obsluhu požadavků

Na obrázku 6.4 je uveden rozhodovací blok funkce pro obsluhu požadavků `pmapproc`. Ve funkci `pmapproc` je rozhodovacímu bloku ještě předřazen bezpečnostní mechanismus, který je detailněji popsán v kapitole 6.5.1. Pokud byly splněny všechny bezpečnostní podmínky, je provedeno rozhodování a volání příslušné funkce. Volání procedur `PMAPPROC_NULL` a `PMAPPROC_DUMP` je zpracováno přímo v rozhodovacím bloku. Procedura `PMAPPROC_CALLIT` není z důvodu úspory implementována. Pro odesílání odpovědí je použitá knihovní funkce `svc_sendreply`, kterou poskytují knihovny RPC. Vstupní parametry funkce `svc_sendreply` jsou informace o transakci, ukazatel na *callback* funkci a ukazatel na adresu, kde jsou uložena data určená k odeslání klientovi. Knihovny pro kódování do formátu XDR poskytují sadu funkcí pro všechny datové typy, které jsou součástí standardu XDR (základní datové typy jsou popsány v kapitole 3.1).

Pokud klientský RPC program volá proceduru `PMAPPROC_DUMP`, je funkci `svc_sendreply` předán ukazatel na funkci `xdr_pmaplist` knihoven XDR, spolu s ukazatelem na lineární seznam. Funkce `svc_sendreply` ve svém těle zavolá funkci `xdr_pmaplist`, která zakóduje lineární seznam do formátu XDR a odešle klientovi odpověď.

6.4.2 Funkce obsluhující jednotlivé procedury

Tato část popisuje obsluhu procedur `PMAPPROC_SET`, `PMAPPROC_UNSET` a `PMAPPROC_GETPORT`. Protože jsou některé části obsluhy procedur trochu rozsáhlejší, byl algoritmus vyjádřen obecně pomocí pseudokódu.

Princip funkce procedury `PMAPPROC_SET`

Tato procedura je volána, když klientská RPC aplikace žádá o nastavení nového mapování. Proceduru `PMAPPROC_SET` vykonává funkce `pmapproc_set`, která má deklarovaný prototyp `bool_t pmapproc_set(struct pmap *)`.

Algoritmus 2: Návrh obsluhy procedury `PMAPPROC_SET`

Input: informace o službě ve struktuře `struct pmap`, která žádá o mapování
Output: výsledek operace vrací hodnotu `FALSE` nebo `TRUE`

```
1 begin
2     while není konec seznamu do
3         if vstupní informace se shoduje se záznamem then
4             return FALSE
5         end
6     end
7     přidej nový záznam do seznamu
8     return TRUE
9 end
```

Algoritmus 2 popisuje princip funkce `pmapproc_set`. Funkce `pmapproc_set` prochází lineární seznam, ve kterém jsou uloženy již namapované služby pluginem portmap a každý prvek tohoto lineárního seznamu porovnává s informacemi ve struktuře, která byla předána jako parametr funkce a obsahuje informace o službě, která žádá o mapování. Porovnávají se pouze hodnoty `pm_prog`, `pm_vers` a `pm_prot`. Hodnota `pm_port` není pro vyhledávání důležitá. Pokud nalezne záznam, který se shoduje s požadavkem klienta, vrátí funkce `pmapproc_set` hodnotu `FALSE`, jinak uloží záznam do lineárního seznamu a vrátí hodnotu `TRUE`.

Princip funkce procedury `PMAPPROC_UNSET`

Procedura `PMAPPROC_UNSET` je volána pokud klientská RPC aplikace žádá o zrušení záznamu o mapování. Procedura `PMAPPROC_UNSET` je obsluhována funkcí `pmapproc_unset`, která má deklarovaný prototyp `bool_t pmapproc_unset(struct pmap *)`.

Základní princip činnosti funkce `pmapproc_unset` popisuje algoritmus 3. V těle funkce `pmapproc_unset` je nejprve důležité nastavit proměnnou, která bude udávat konečný stav operace. Tato funkce má vstupním parametrem strukturu `struct pmap`, ale klíčové jsou pro tuto proceduru pouze hodnoty `pm_prog` a `pm_vers`, ostatní hodnoty jsou ignorovány. Protože může nastat situace, kdy bude položek v lineárním seznamu, které mají shodné hodnoty `pm_prog` a `pm_vers`, více, je nutné projít celý seznam a až poté vyhodnotit výsledný stav funkce. Každá položka lineárního seznamu uchovávající záznamy o mapování, která bude shodná v klíčových hodnotách s požadavkem klienta, bude zrušena. Pokud byla alespoň jedna položka v seznamu zrušena, vrátí funkce `pmapproc_unset` hodnotu `TRUE`, jinak vrací hodnotu `FALSE`.

Algoritmus 3: Návrh obsluhy procedury `PMAPPROC_UNSET`

Input: informace o službě ve struktuře `struct pmap`, která žádá o zrušení mapování
Output: výsledek operace vrací hodnotu `FALSE` nebo `TRUE`

```
1  begin
2      výsledek ← FALSE
3      while není konec seznamu do
4          if vstupní informace se shoduje se záznamem then
5              smaž záznam v seznamu
6              výsledek ← TRUE
7          end
8      end
9      return výsledek
10 end
```

Princip funkce procedury `PMAPPROC_GETPORT`

Pokud obdrží plugin portmap žádost o zjištění portu, na kterém požadovaná služba naslouchá, je volána procedura `PMAPPROC_GETPORT`. Vykonání procedury `PMAPPROC_GETPORT` zajišťuje v pluginu portmap funkce `pmapproc_getport`. Funkční prototyp této funkce je deklarován `uint32_t pmapproc_getport(struct pmap *)`.

Algoritmus 4: Návrh obsluhy procedury `PMAPPROC_GETPORT`

Input: informace o požadované službě ve struktuře `struct pmap`
Output: číslo portu, na kterém požadovaná služba naslouchá

```
1  begin
2      while není konec seznamu do
3          if vstupní informace se shoduje se záznamem then
4              return číslo portu uvedené v záznamu
5          end
6      end
7      return FALSE
8  end
```

Algoritmus 4 znázorňuje základní princip funkce `pmapproc_getport`. Vyhledávání požadované služby v lineárním seznamu, který uchovává seznam mapovaných RPC programů, se provádí porovnáváním hodnot `pm_prog`, `pm_vers` a `pm_prot` v záznamu s hodnotami v předané struktuře, která obsahuje informace o požadované službě. Hodnota `pm_port` je ignorována – právě tuto hodnotu očekává klient. Funkce `pmapproc_getport` prochází seznam s mapovanými službami a pokud narazí na shodující se záznam v klíčových položkách s požadavkem, vrátí funkce číslo portu, na kterém registrovaná služba naslouchá. Jinak vrátí hodnotu `FALSE`, neboli hodnotu 0.

6.5 Návrh bezpečnostních opatření

Neméně důležitá část vývoje pluginu portmap je návrh aplikace tak, aby byla schopna odolávat potenciálním útokům zvenčí, případně i z lokální stanice. Byly navrženy tři různé

mechanismy, které pomáhají zlepšit bezpečnost celé aplikace. První část této podkapitoly rozebírá, co a proč, je třeba chránit, následující část popisuje, jakým způsobem se plugin portmap brání neprivilegovaným osobám na lokálním stroji. Třetí část podkapitoly popisuje způsob, jak lze rozpoznat, zda je klientská aplikace, která se připojila a odesílá požadavky, lokální či nikoliv. Poslední část této podkapitoly popisuje, jakým způsobem byl navržen mechanismus v případě selhání předchozích dvou restrikcí, aby bylo zamezeno alespoň zaplnění všech systémových prostředků a nedošlo tak k vyřazení systém z provozu.

6.5.1 Jak se lze před útokem bránit

Plugin portmap musí být spouštěn s právy uživatele `root`, aby mohl naslouchat na portu 111. Tento fakt sebou přináší jistá rizika. Aplikace spuštěná s právy uživatele `root` má téměř neomezené možnosti a v případě úspěšného napadení lze systém vyřadit z provozu. Vrstva RPC poskytuje mechanismus k autentizaci (detailněji popsáno v kapitole 2.4), ale pokud neprobíhá autentizace pomocí `AUTH_DES`, hrozí jisté riziko prolomení. Proto je nutné zavést různá omezení, které pomohou bezpečnost zlepšit.

Procedury `PMAPPROC_NULL`, `PMAPPROC_GETPORT` a `PMAPPROC_DUMP` není potřeba nijak zabezpečovat, protože u nich by k útoku dojít nemělo, respektive útok by neměl žádný větší význam. Jediný možný útok při volání těchto procedur je neustálé cyklické volání tak, aby plugin portmap vytížil procesor systému na 100 %. Větší hrozbou jsou procedury `PMAPPROC_SET`, `PMAPPROC_UNSET` a `PMAPPROC_CALLIT`.

```
if (rqstp->rq_proc == PMAPPROC_SET
 || rqstp->rq_proc == PMAPPROC_UNSET)
    if (check_security(xprt) == FALSE) {
        svcerr_weakauth(xprt);
        return;
    }
```

Obrázek 6.5: Blok kódu, který rozhoduje o kontrole bezpečnostních pravidel

Na obrázku 6.5 je ukázka zdrojového kódu, který vyhodnocuje, zda bude požadavek podroben bezpečnostní kontrole. Pokud je příchozí požadavek žádost o nové mapování nebo zrušení mapování, volá se pro ověření bezpečnosti funkce `check_security`, která obsahuje dva ověřovací kroky. Ověřuje se, zda je klient lokální a zda má dostatečné práva. Detailnější popis je uveden v následujících podkapitolách 6.5.2 a 6.5.3.

Zneužití procedury `PMAPPROC_SET`

Tato procedura alokuje systémové zdroje a kdyby neexistovalo žádné omezení, mohl by útok vyčerpat všechny systémové zdroje. Proto je důležité kontrolovat, jestli je port v definovaném rozsahu $\langle 1,\ 65535\rangle$, není roven portu, na kterém naslouchá portmap a je nutné tuto žádost podrobit kontrole bezpečnostních pravidel (viz. následující kap. 6.5.2 a 6.5.3). Záporné číslo portu nebo port větší než 65535 je nesmyslné a pokud není nijak limitováno, zvětšuje počet položek, které může plugin portmap uložit.

Zneužití procedury `PMAPPROC_UNSET`

Zneužití této procedury může vést k vyřazení všech poskytovaných vzdálených RPC služeb.

Zneužití procedury `PMAPPROC_CALLIT`

U procedury `PMAPPROC_CALLIT` hrozí riziko zacyklení – pokud by chtěl klient volat vzdálenou službu a uvedl by opět portmapper, proceduru `PMAPPROC_CALLIT` a upravil by vhodně argumenty, mohlo by dojít k zacyklení a vyčerpání systémových zdrojů. Tato procedura v současné době není implementována z důvodu úspory, ale je možné že v budoucích fázích vývoje bude doimplementována na základě požadavků uživatelů. Proto je důležité alespoň zmínit tyto rizika.

6.5.2 Kontrola oprávnění

Aplikace, které mají práva uživatele *root*, mohou používat porty s číslem 1024 a nižší. Z tohoto faktu vychází i toto bezpečnostní opatření. Pokud má lokální aplikace práva uživatele *root*, pak nemá smysl cokoliv omezovat, protože má aplikace téměř neomezené možnosti. Registrovat a odregistrovat RPC aplikace může tedy jen oprávněný program s příslušnými uživatelskými právy. To, zda RPC aplikace získá práva uživatele *root*, je už ponecháno na operačním systému.

```
if (!(option_mask32 & FLAG_SPORT))
    if (htons(svc_getcaller(xprt)->sin_port) > 1024)
        return FALSE;
```

Obrázek 6.6: Blok kódu, který je určen pro kontrolu oprávnění lokálních RPC aplikací

Implicitně je plugin portmap spouštěn v „secure“ módu a neumožňuje registrovat a odregistrovávat RPC služby aplikacím, které se připojí z portu většího než 1024. Jak je ale vidět na obrázku 6.6, který ukazuje blok kódu, který ověřování korektnosti portu zajišťuje, lze tuto ochranu potlačit při spuštění pluginu portmap (ověřování lze potlačit nastavením přepínače `-p` při spouštění pluginu portmap). Tato kontrola je z optimalizačních důvodů předřazena kontrole ověřování, zda je klientská RPC aplikace lokální.

6.5.3 Ověřování původu klientské aplikace

Pro ověření původu klientské RPC aplikace byl navržen mechanismus, který porovnává IP adresu příchozího požadavku s IP adresou lokálních síťových rozhraní. Tento mechanismus rozezná lokální od vzdálené RPC aplikace a umožní provedení žádosti pouze lokálním aplikacím. Tuto bezpečnostní kontrolu lze při startu pluginu portmap explicitně potlačit, ale důrazně je doporučeno tuto ochranu nevypínat – plugin portmap mapuje pouze lokální RPC aplikace a není tudíž důvod povolit mapování vzdáleným klientům. Na obrázku 6.7 je uveden blok zdrojového kódu, který zajišťuje ověřování původu klienta.

Globální proměnná `G.ifs` vyjadřuje počet síťových rozhraní a je nastavena při prvním spuštění pluginu portmap implicitně na hodnotu 5. Při první kontrole dojde k optimalizaci a upraví se její hodnota na aktuální počet síťových rozhraní inkrementovaný o jedničku. Na

```
if (!(option_mask32 & FLAG_SREMOTE)) {
    bool_t res = FALSE;
    struct sockaddr_in *tmp;
    struct ifconf ifc = {.ifc_req = NULL};
    int s = xsocket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    for (int cnt = 0; !cnt
        || ifc.ifc_len/sizeof(struct ifreq) == cnt;
        cnt += G.ifs) {
        ifc.ifc_len = (cnt+G.ifs)*sizeof(struct ifreq);
        ifc.ifc_req = xrealloc(ifc.ifc_req, ifc.ifc_len);
        if (ioctl(s, SIOCGIFCONF, &ifc) < 0) {
            bb_error_msg("ioctl(SIOCGIFCONF) error");
            goto out;
        }
    }
    G.ifs = ifc.ifc_len/sizeof(struct ifreq)+1;
    for (int i = 0; i < G.ifs-1; i++) {
        tmp = (struct sockaddr_in *)&ifc.ifc_req[i].ifr_addr;
        if (tmp->sin_addr.s_addr
         == svc_getcaller(xprt)->sin_addr.s_addr) {
            res = TRUE;
            goto out;
        }
    }
out:
    free(ifc.ifc_req);
    return res;
}
```

Obrázek 6.7: Blok kódu, který ověřuje původ klientské aplikace

základě této hodnoty se alokuje paměťový prostor pro struktury, které obsahují informace o síťových rozhraních. Proto je vhodné udržovat informaci o aktuálním počtu síťových rozhraní, aby nedocházelo k zbytečnému plýtvání systémových prostředků. Kdybychom neznali počet rozhraní, docházelo by často k dynamické realokaci, která je časově náročná. Kdybychom naopak použili předem definovanou velikost paměťového prostoru, který by musel být poněkud velký, nebylo by to úsporné.

Zjišťování údajů o síťových rozhraních je prováděno prostřednictvím volání systémové funkce `int ioctl(int d, int request, ...)`, která provádí operace nad *file deskriptory*. Abychom získali informace o síťových rozhraních, musíme předat funkci schránku, kterou jsme vytvořili, žádost `SIOCGIFCONF` a ukazatel na strukturu `struct ifconf`, která má alokovaný paměťový prostor pro uložení pole struktur `struct ifreq`.

V prvním cyklu zdrojového kódu, který je na obrázku 6.7, získáme seznam síťových rozhraní. V druhém cyklu porovnáváme IP adresu klientského požadavku s IP adresou

jednotlivých rozhraní. Pokud nalezneme IP adresu rozhraní, která se shoduje s IP adresou klienta, pak se jedná o RPC aplikaci, která běží na lokálním stroji. Existuje i způsob, jak lze podvrhnout IP adresu požadavku tak, aby měla lokální charakter. Z toho vyplývá, že ani tato ochrana není stoprocentně účinná.

6.5.4 Ochrana proti vyčerpání systémových zdrojů

Poslední z navržených ochran je určena až pro případ, že předchozí dvě ochrany nebyly účinné. Při instalaci BusyBoxu je volena maximální hodnota RPC služeb, které muže plugin portmap mapovat. Po vyčerpání limitu je registrace zamezená a je provedeno hlášení do systémového logu s varováním o potenciálním útoku. Na obrázku 6.8 je uvedena ukázka zdrojového kódu, který ověřuje ve funkci `pmapproc_set` počet aktuálně mapovaných RPC programů. Pokud byl dosažen limit, bude požadavek na nové mapování vždy neúspěšný.

```
if (G.pn >= CONFIG_PORTMAP_ITEMS_MAX) {
    bb_error_msg("All available resources are used! Count of \
                 "registered RPC services is limited to %i.",
                 CONFIG_PORTMAP_ITEMS_MAX);
    return FALSE;
}
```

Obrázek 6.8: Blok kódu, který kontroluje maximální počet mapovaných RPC programů

Kapitola 7

Testování nového appletu

Testování je jednou z nejdůležitějších částí životního cyklu software a je velice důležité tuto část nepodceňovat. Testování slouží k odhalení chyb software, které vznikly při jejím vývoji, a proto dříve než uvedeme aplikaci do reálného provozu, musíme důkladně ověřit její bezchybnou funkčnost. V případě, že se jedná o aplikaci, která může být napadnutá útočníkem zvenčí, což v našem případě portmap bezpochyby je, je nutné řádně otestovat i bezpečnost celé aplikace a její odolnost proti různým typům útoku. Plugin portmap byl podroben několikafázovému testování funkčnosti, stability, bezpečnosti, které je popsáno v této kapitole.

7.1 Testování funkčnosti

Testování bylo rozděleno do dvou kategorií. V první části je uvedeno nativní testování, které má za cíl ověřit základní funkcionalitu prostřednictvím standardních RPC aplikací. Druhou částí je testování syntetické, které si klade za cíl odhalit hlubší a bezpečnostní chyby v pluginu portmap.

7.1.1 Nativní

Pro otestování správné funkčnosti bylo nutné vytvořit jednoduchou aplikaci, která využívá RPC. Aplikace se skládá z klienta a serveru. Serverovská aplikace při svém spuštění nejprve odesílá požadavek na odregistrování své služby portmapperu a až poté požadavek na registraci své služby. Toto je klasické chování RPC aplikací a děje se tak proto, aby se předešlo problémům, které by vznikly v případě, že se od minulého spuštění aplikace neodregistrovala – například při jejím pádu. Kdyby serverovská aplikace již registrována byla a nepožádala o zrušení registrace, obdržela by negativní odpověď při registraci, tím by aplikace vypsala chybové hlášení a ukončila svůj běh. Při spuštění klientského programu se nejprve aplikace dotazuje portmapperu na číslo portu své serverovské služby. Poté už volá funkce, které jsou zpracovávány na RPC serveru a výsledky jsou odesílány zpět. Tento druh testování testuje pouze základní funkcionalitu aplikace.

7.1.2 Syntetické

Pro syntetické testování bylo zapotřebí vytvořit aplikaci, která bude odesílat námi definované RPC zprávy portmapperu a přijímat jejich odpověď. Je důležité, aby bylo možné měnit všechny možné parametry odesílané zprávy. Tato metoda testuje pokročilé vlastnosti

portmapperu. Prakticky jde o testování všech možných kombinací, které můžou nastat za předpokladu, že má RPC zpráva vhodnou délku. Tento druh testování proběhl pouze omezeným způsobem, protože počet možných kombinací je příliš mnoho. Testování bylo tedy provedeno tak, že se testovaly hodnoty v intervalu, které portmapper pro danou položku přijímá a dvě hodnoty, které nepřijímá – první hodnotu před, a druhou hodnotu za intervalem. Nejprve si definujeme, jak se má plugin portmap chovat v jednotlivých případech. Pak provedeme testování a porovnáme jeho výsledky s námi definovaným chováním. Pokud se výsledky shodují, můžeme prohlásit, že se portmapper chová správně. Toto testování je komplexní a netestuje pouze portmapper, ale testuje například i vazbu se systémem – zda se portmapper správně registroval při svém spuštění v systému RPC a podobně. K ověření správného chování je tato metoda dostačující.

Nyní následují jednotlivé kroky syntetického testování, ve kterých je definováno, jak se má systém a plugin portmap správně chovat v konkrétních situacích.

XID

Testování rozsahu hodnot `XID` nemá v tomto případě žádný hlubší význam, protože slouží pouze k udržení kontextu mezi klientem a serverem. Je to identifikátor, na základě kterého klient rozezná odpověď.

Typ RPC zprávy

Protože plugin portmap nemá z důvodu úspory implementovanou funkci `PMAPPROC_CALLIT`, musí být příchozí RPC zpráva s požadavkem vždy typu `CALL`. Z toho vyplývá, že plugin portmap odpoví pouze v případě, že je typ RPC zprávy `CALL message`. Pokud není tato podmínka splněna, RPC zpráva bude zahozena.

Verze RPC protokolu

V současné době se používá verze RPC protokolu 2. Vyšší verze v současné době neexistuje, a verze 1 se nepoužívá. Z tohoto důvodu musí být verze RPC protokolu vždy rovna dvěma. Pokud není verze protokolu akceptovatelná, odešle systém odpověď, která má status `MSG_DENIED` s příznakem `RPC_MISMATCH`. Zpráva musí obsahovat informaci o nejnižší a nejvyšší verzi protokolu RPC, která je podporována.

Číslo programu

Číslo programu je u portmapperu pevně stanoveno organizací IANA na hodnotu 100000. Pokud vrstva RPC obdrží zprávu, která má v položce `program number` jinou hodnotu než je číslo programu portmap, zprávu akceptuje, ale nastaví u odpovědi příznak `PROG_UNAVAIL` a odešle klientovi.

Verze programu

Portmap má číslo verze 2. Pro verze 3 a 4 se aplikace nazývá rpcbind, která se od portmapu liší a má rozšířenější funkcionalitu. Proto by měla být verze programu rovna dvěma. Pokud není tato podmínka splněna, systém zprávu akceptuje, ale odpověď bude mít nastaven příznak `PROG_MISMATCH`. Zpráva musí obsahovat informaci o nejnižší a nejvyšší verzi programu, která je portmapperem podporována. Serverovská aplikace odpověď přijme a sníží své požadavky.

Číslo procedury

Ve standardním případě je číslo procedury `PMAPPROC_CALLIT` akceptovatelné. Implementace pro BusyBox tuto proceduru nemá z důvodu úspory implementovanou, a proto přijímá procedury pouze z intervalu ⟨`PMAPPROC_NULL`, `PMAPPROC_CALLIT`⟩. Pokud číslo procedury do intervalu nepatří, odesílá portmapper odpověď s příznakem `PROC_UNAVAIL`.

Procedura `PMAPPROC_NULL`

Tato procedura přijímá argument `void`. Pokud portmapper obdrží volání této procedury s libovolným argumentem, chování této procedury by mělo vypadat vždy stejně – korektní odpověď s položkou `result` typu `void`.

Procedury `PMAPPROC_SET` a `PMAPPROC_UNSET`

Žádost o nastavení nebo zrušení registrace programu může být vyhověna požadavkům splňujícím tyto kritéria:

1. Pokud není plugin portmap v módu „insecure“, musí být číslo portu žadatele rovno nebo menší číslu 1024 a žadatel musí být na lokálním stroji.
2. Port musí být v intervalu $\langle 1\ , 65535 \rangle$.
3. Port nesmí být portmapem již registrován.

Pokud jsou podmínky splněny, bude odpověď `TRUE`. Jestliže není splněný první bod podmínek, zpráva bude zamítnuta a v odpovědi nastaven příznak `RPC_AUTHERROR`. Pokud jsou splněny první dvě podmínky, ale není splněna podmínka třetí, bude vrácena odpověď `FALSE`.

Procedura `PMAPPROC_GETPORT`

Žádost bude vyhověna, pokud je číslo portu v intervalu $\langle 1\ , 65535 \rangle$ a zároveň je požadovaný program portmapperem registrovaný. Pokud je žádosti vyhověno, odpověď obsahuje číslo portu, na kterém požadovaný RPC server naslouchá. V opačném případě pošle odpověď s číslem portu roven nule.

Procedura `PMAPPROC_DUMP`

Procedura `PMAPPROC_DUMP`, stejně jako procedura `PMAPPROC_NULL`, se bude chovat vždy stejně bez ohledu na argumenty předávané proceduře – odpověď bude obsahovat seznam registrovaných RPC programů.

7.2 Testování stability

Portmap naslouchá na TCP i UDP portu. U UDP přenosu hrozí riziko, že bude příchozí packet poškozen, přijde jen část, případně nebude doručen vůbec. Testování probíhalo náhodným posíláním dat na port, na kterém naslouchá pluginu portmap délky 1 až délka běžné RPC zprávy.

7.3 Testování bezpečnosti

Protože je portmap vystaven potenciálním útokům, je nutné jej otestovat na různé typy útoku. V následující části sekce jsou uvedeny různé typy útoků, které byly aplikovány na implementovaný plugin portmap.

Útok zahlcením daty

Klient se připojí k portu pluginu portmap a odesílá velké množství dat, které by mohlo zaplnit dostupnou operační paměť a vyřadit tak systém z provozu. Toto testování bylo provedeno pomocí utility `netcat` [5] a libovolného velkého souboru či generátoru náhodných dat[1]. Například, první řádek ukazuje posílání náhodných dat portmapperu naslouchajícímu na TCP portu a druhý řádek naslouchajícímu na UDP portu.

```
# nc localhost 111 -t < /dev/urandom
# nc localhost 111 -u < /dev/urandom
```

Útok registrováním

Pokud je portmap v módu „insecure“, může nastat případ, kdy bude útočník neustále registrovat nové RPC programy a tím může vyčerpat systémové zdroje. Pro tento případ byla zavedena konstanta, která je volena při instalaci BusyBoxu. Tato konstanta udává maximální počet registrovaných programů portmapem.

7.4 Zhodnocení testů

Plugin portmap úspěšně obstál ve všech uvedených testech a můžeme o něm prohlásit, že je dostatečně stabilní a bezpečný pro provoz v reálných podmínkách.

[1] Tento způsob je příliš pomalý.

Kapitola 8

Oprava appletu v balíku BusyBox

V rámci seznamování s projektem BusyBox byla opravena chyba v pluginu `patch`, která byla uvedena na seznamu chyb v bugzille tohoto projektu. Chyba byla původně ohlášena jako chyba pluginu `diff`. Uživatel upozornil na odstraňování prázdných souborů a vložil citaci z manuálové stránky, která hovořila o normě POSIX a odstraňování prázdných souborů s využitím přepínače `-E` nebo `--remove-empty-files`.

Analýza

Byla provedena analýza chování `diff`u a nasimulována situace tak, aby bylo toto chování zrekonstruováno. Byly vytvořeny dvě adresářové struktury, které se lišily v obsahu svých souborů. Obsahovaly různé kombinace stavů.

Pak byl vytvořen patch pomocí pluginu `diff` a srovnán s výstupem referenčního `diff`u, který byl použit z distribuce Linuxu. Oba výstupy byly identické. Už z popisu bylo zřejmé, že se pravděpodobně nebude jednat o chybu v appletu `diff`, ale v appletu `patch`. Toto testování jen potvrdilo tuto hypotézu.

Ve druhé fázi testování byl vytvořený patch aplikován zpětně pomocí pluginu `patch` a výsledek srovnán s aplikací referenční utility `patch` z distribuce. Výsledek byl odlišný a tím se prokázala chyba v pluginu `patch`.

Řešení

Podle standardu POSIX se standardně po aplikaci patche neodstraňují prázdné soubory a složky. Toto implicitní chování lze změnit nastavením přepínače `--remove-empty-files` nebo jeho zkráceným ekvivalentem `-E`. Plugin `patch` z balíku BusyBox toto doporučení nerespektoval a nebyl ani tento přepínač k dispozici. Prázdné soubory a složky implicitně odstraňoval, což bylo v rozporu se standardem POSIX. Bylo tedy nutné stávající implementaci pluginu `patch` rozšířit o tuto volbu, aby v tomto ohledu splňovala doporučení standardu POSIX.

Na závěr byl vytvořen patch, který byl odeslán do mailing listu balíku BusyBox. Maintainer BusyBoxu mírně patch modifikoval pro větší úsporu systémových prostředků a aplikoval tuto záplatu do hlavní větve BusyBoxu.

Kapitola 9

Závěr

Cílem této práce bylo vytvořit ve spolupráci s firmou Red Hat plugin portmap pro BusyBox. Spolupráce s firmou Red Hat probíhala prostřednictvím osobních schůzek s odbornou vedoucí v místě sídla firmy Red Hat a hlavně také emailovou komunikací. První částí spolupráce byl teoretický návrh obecného portmapperu, který byl následně konzultován. Po schválení a doporučení některých změn v teoretickém návrhu byl započat vývoj portmapperu na standardní Linuxové platformě. Cílem tohoto kroku vývoje bylo vyvinout program portmap pro běžnou Linuxovskou distribuci.V dalším kroku byla provedena portace vytvořeného portmapu pro BusyBox – byly provedeny optimalizace, nahrazeny některé knihovní funkce funkcemi z BusyBoxu. Počet řádků kódu oproti dřívější implementaci se zredukoval zhruba na polovinu. Ve fázi, kdy byla dokončena implementace pluginu portmap, byla vytvořena záplata a odeslána do mailing listu BusyBoxu. Plugin portmap byl vystaven kritice vývojářů BusyBoxu. Správce BusyBoxu i ostatní vývojáři posílali připomínky ohledně zdrojového kódu a různá doporučení pro lepší optimalizaci. Všechny doporučení byly aplikovány.

Nejdůležitějším kritériem pro hodnocení kvality pluginu pro BusyBox, je jeho výsledná velikost v binární podobě, neboli přírůstek velikosti binární aplikace BusyBox. Implementovaný plugin portmap má ve výsledné podobě velikost 6,3 kB, což je poměrně hodně. Největší část velikosti pluginu (cca 70-80 %) tvoří knihovní funkce RPC. Hlavním důvodem, proč nebyl plugin portmap do BusyBoxu přijat, je použití knihovních funkcí RPC, které nejsou optimalizované a u funkcí, které byly použity, chybí podpora IPv6.

V budoucí části vývoje je možné provést reimplementaci základních knihovních funkcí, které jsou součástí RPC a přidat podporu pro IPv6, aby plugin splňoval všechny stanovené požadavky a kritéria BusyBoxu.

Literatura

[1] *BusyBox* [online]. `http://cs.wikipedia.org/wiki/BusyBox`, 2010-08-29 [cit. 2011-04-25].

[2] *RPC – Remote Procedure Call Protocol Specification Version 2*. RFC 1057, Sun Microsystems, June 1988.

[3] Andersen, E.: *BusyBox – The Swiss Army Knife of Embedded Linux* [online]. `http://www.busybox.net/`, [cit. 2011-04-25].

[4] Eisler, M.: *XDR – External Data Representation Standard*. RFC 4506, Network Appliance, May 2006.

[5] Giacobbi, G.: *The GNU Netcat project* [online]. `http://netcat.sourceforge.net/`, 2006-11-01.

[6] Kryzhanovskyy, M.: *Implementace pluginů pro ifplugd do projektu BusyBox*. Bakalářská práce, Fakulta Informačních Technologií, VUT v Brně, 2010.

[7] Matoušek, P.: *Studijní opora do předmětu ISA*. Fakulta Informačních Technologií, VUT v Brně, 2011.

[8] Srinivasan, R.: *Binding Protocols for ONC RPC Version 2*. RFC 1833, Sun Microsystems, August 1995.

[9] Srinivasan, R.: *XDR – External Data Representation Standard*. RFC 1832, Sun Microsystems, August 1995.

[10] Thurlow, R.: *RPC – Remote Procedure Call Protocol Specification Version 2*. RFC 5531, Sun Microsystems, May 2009.

Příloha A

Obsah DVD

DVD, které je přiloženo k této bakalářské práci obsahuje níže uvedené soubory.

Cesta	**Popis**
`/busybox/`	Zdrojové kódy balíku BusyBox.
`/busybox_patched/`	Zdrojové kódy balíku BusyBox včetně pluginu `portmap`.
`/rpc-app/`	Vzorová aplikace, která využívá RPC.
`/tex/`	Zdrojové soubory (LaTeX) této bakalářské prace.
`/patch.patch`	Opravná záplata pro plugin `patch`.
`/portmap.patch`	Záplata, která přídá do BusyBoxu plugin `portmap`.
`/projekt.pdf`	Tato bakalářská práce ve formátu PDF.