VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

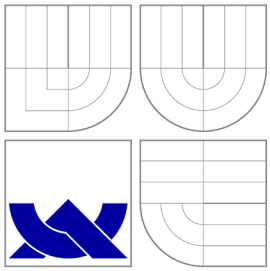
SADA TESTŮ PRO RED HAT TEST SUITE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

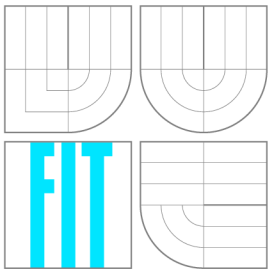
AUTOR PRÁCE
AUTHOR

KATEŘINA NOVOTNÁ

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

SADA TESTŮ PRO RED HAT TEST SUITE

A TEST SET FOR RED HAT TEST SUITE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — KATEŘINA NOVOTNÁ
AUTHOR

VEDOUCÍ PRÁCE — Ing. ALEŠ SMRČKA
SUPERVISOR

BRNO 2009

Abstrakt

Tato bakalářská práce se zabývá přípravou testů pro Red Hat Test System (RHTS). Vysvětluje základní metody testování a důvody pro jejich automatizaci, je popsán systém RHTS a způsob vytváření sady testů pro softwarový balíček. Vybrané balíky, kterými jsou wireshark a wireshark-gnome jsou popsány, jsou navrhnuty oblasti, pro které budou vytvořeny testy, tyto testy jsou konkrétně navrhnuty a poté implementovány. Testy jsou implementovány pomocí frameworku Dogtail pro automatizované testování GUI a knihovny rhtslib pro ostatní testy.

Abstract

This bachelor's thesis deals with test preparation for Red Hat Test Sytem (RHTS). Basic concepts of software testing explained, RHTS described in more detail, together with an approach how to make a test set for a software package. The main areas for test for selected packages, wireshark and wireshark-gnome in particular, are analysed and tests are designed and implemented using Dogtail framework for GUI testing and RHTS library for other tests.

Klíčová slova

softwarové testování, black-box testování, RHTS, wireshark, wireshark-gnome, Dogtail, GUI testování, rhtslib

Keywords

software testing, black-box testing, RHTS, wireshark, wireshark-gnome, Dogtail, GUI testing, rhtslib

Citace

Kateřina Novotná: Sada testů pro Red Hat Test Suite, bakalářská práce, Brno, FIT VUT v Brně, 2009

Sada testů pro Red Hat Test Suite

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením pana Aleše Smrčky. Další informace mi poskytli pan Jan Lieskovský a pan Ondřej Hudlický ze společnosti Red Hat. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Kateřina Novotná
20.5.2009

Poděkování

Chci poděkovat svému vedoucímu bakalářské práce Aleši Smrčkovi za vstřícnost, čas, který mi věnoval a jeho cenné rady. Dále bych ráda poděkovala Janu Lieskovskému a Ondřeji Hudlickému za poskytnutí dalších odborných rad a nových podnětů.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Tato práce se zabývá problematikou softwarového testování a vytváření testů pro programové balíky distribuce Ret Hat Enterprise Linux.

Součástí vývojového cyklu software je fáze verifikace a validace (v literatuře se tyto pojmy často zaměňují), jejímž cílem je zajistit kvalitu vyvýjeného software. Mezi metody validace řadíme metody formální verifikace (pomocí fomálních metod se dokazuje správnost programu vůči jeho specifikaci), metody statické analýzy (analýza zdrojového kódu) a metody dynamické analýzy (sledující běh programu). Testování můžeme zařadit mezi metody dynamické analýzy.

I když se u velkých projektů několikrát dokázalo, že fáze testování šetří důležitý čas vývoje, bývá u menších projektů často podceňována. Testování je neméně důležité pro úspěšné dokončení každého projektu a při správné strategii se může vývoj programu zefektivnit díky snadnějšímu odhalování zdrojů chyb.

Úkolem této práce je vybrat balíčky k vytvoření testů, pro tyto balíčky vytvořit návrh testů, tyto testy implementovat a připravit k používání v systému pro automatizaci testování programů - Red Hat Test System. Cílem je tedy vytvořit sadu testů pro podporu automatizovaného testování vybraných balíčků pro jejich jednoduché znovupoužití.

Úvodní část textu se zabývá vysvětlením principu testování, jeho procesu a vysvětluje různé přístupy k vytváření testů. Další část se věnuje představení Red Hat Test System a prostředkům pro vytváření testů do tohoto testovacího systému. Ústředními kapitolami práce jsou části, ve kterých jsou vybrány programové balíky k vytvoření testů, analýza částí balíků pro které budou vytvořeny testy, popis implementace a zhodnocení dosažených výsledků.

Kapitola 2

Základy testování

Tato kapitola vysvětluje zařazení testování z pohledu vývojového procesu software, dále definuje testování a vysvětluje jeho účel. Vysvětlí základy problematiky úrovní testování z pohledu stavu vývojového procesu. Zaměřuje se také na vysvětlení dvou základních přístupů testování a to black-box a white-box testování. Pohled na testovací proces se v různých publikacích liší. Odlišnost spočívá v nejednotnosti náhledu na testovací přístupy, které mohou patřit i do více než jedné kategorie. Zde si popíšeme přístup popisovaný v knihách [12] a [13] a v kurzu autora Cema Kanera [10]. Konec kapitoly se věnuje vysvětlení automatizace testování.

Softwarový proces se skládá z definování požadavků, návrhu, implementace, nasazení a údržby softwaru. Součástí implementace je také verifikace a validace, které probíhají současně s vývojovou fází. Verifikace a validace je proces ověřování, že software odpovídá specifikaci a umožňuje ho používat s účelem, se kterým byl vyvýjen. Verifikace ověřuje, zda návrh řešení splňuje požadavky na kvalitu a jestli je implementace provedena podle návrhu. Validace ověřuje, zda to co zákazník chce, také skutečně potřebuje a jestli implementace software má věškerou funkcionalitu, která je potřeba k jeho zamýšlenému používání [21]. Validace je důležitá a k jejímu provedení se používají tyto techniky: formální verifikace, statická analýza a dynamická analýza.

Formální verifikace ověřuje správnost algoritmu s ohledem na jeho formální specifikaci. Její největší výhodou oproti ostatním analýzám je, že teoreticky prozkoumá všechny možné případy, ale za cenu toho, že může být velmi náročná - exponenciální růst stavového prostoru způsobuje tzv. explozi stavového prostoru.

Statická analýza je analýza zdrojových kódů. Je určena pro hledání sémantických a logických chyb. Statickou analýzu lze automatizovat. Používané nástroje v této oblasti jsou buď pro určenou sadu programovacích jazyků jako třeba RATS [16] nebo Yasca [16], nebo pro konkrétní programovací jazyk. Pro jazyk C je to například Blast [9] nebo UNO [5]. Výhodami statické analýzy jsou její jednoduchost a rychlost při použití nástrojů k její automatizaci. Nevýhodou je, že může být také extrémně pomalá, nástroje k automatizaci nepodporují všechny programovací jazyky, a to, že nástroje mohou identifikovat tzv. false positives, tedy oznamují chybové chování, které při běhu programu nemůže nikdy nastat.

Dynamická analýza sleduje účinky pro konkrétní běh programu. Program je spouštěn se vstupy, které produkují zajímavé chování programu [18]. Do tohoto druhu analýzy můžeme

zařadit i testování. Jejími výhodami jsou jednoduchost, identifikace zranitelností při běhu programu, umožňuje analyzovat program i když nemáme přístup k jeho zdrojovému kódu. Nevýhodou je, že tato analýza se provádí vždy pro konkrétní běh a je potřeba odborníka, který testy vytvoří.

Tato práce se zabývá dynamickou analýzou, konkrétně její částí - testováním.

2.1 Co je testování

Testování je empirické technické zkoumání za účelem poskytnout informace o kvalitě produktu nebo služby zájmovým stranám [20]. Účelem testování není jen předat vývojáři seznam chyb, ale také ověřit, že testovaný objekt je validní, tedy že odpovídá požadavkům na jeho používání, nejenom návrhu.

Při testování je nejefektivnější „podezřívavý přístup“. Tedy ověřování toho, že program nejen že dělá to co má, ale především, že nedělá to co nemá. Především ověření druhé podmínky není vždy jednoduché. Rozdíl si můžeme uvést na příkladu - máme seznam všech validních voleb programu. Pro každou volbu vytvoříme test, který ověří že při vstupu validní volby dostaneme očekávaný výstup. Když se na to podíváme z druhé strany, jak vytvořit seznam všech nevalidních voleb programu? Sestavit takový seznam je nemožné, místo toho můžeme sestavit seznam všech možných skupin nevalidních voleb a dostat tak konečný počet takovýchto testů.

Testování může být zahájeno kdykoliv během vývojového procesu, nejdříve jakmile jsou přesně specifikovány požadavky na daný software. Obecně platí, že čím dříve je testování zahájeno, tím dříve dojde k odhalení chyb a tím méně času a peněz musí být na jejich opravu vynaloženo.

2.2 Úrovně testování

Úrovně testování jsou klíčovým konceptem testování. Představují určitou úroveň abstrakce nad testovaným objektem. Existují čtyři základní úrovně testování. Dále si vysvětlíme jednotlivé úrovně. V textu, který následuje, jsou v některých případech použity anglické termíny pro názvy metod. Je to především proto, že jsou již tak zažity a uvedení českých výrazů, které by je popisovaly by mohlo být matoucí.

2.2.1 Unit testování

Unit je nejmenší testovatelná jednotka testovaného objektu. V procedurálním programování takovou jednotkou může být funkce v objektovém metoda nějakého objektu. Cílem unit testování je ověřit, že každá jednotlivá část je funkční a odpovídá specifikaci.

Tyto testy jsou obvykle vytvářeny programátorem. Nevýhodou takového přístupu je, že testy mouhou být napsány tak, aby odpovídaly implementaci, ale už nemusí odpovídat specifikaci. Obvyklým řešením je, že testy jsou napsány dříve, než začne psaní kódu pro

testovanou část.

2.2.2 Integrační testování

Může být prováděno u takových modulů, u kterých bylo dokončeno unit testování. Je prováděno z důvodu ověření správné interakce a funkcionality mezi dvěma a více moduly. Integrační testování může být prováděno třemi různými způsoby, podle toho kde začneme testovat. Jsou to *Big Bang* testování, *Bottom up* a *Top down* testování [19]. Big Bang je přístup, kde jsou všechny nebo velká část modulů spojeny a testovány společně. Tento přístup je velmi efektivní, pokud jsou důkladně a podrobně zaznamenávány výsledky testů. V opačném případě je hledání zdrojů chyb velmi komplikované. Bottom up testování je přístup, kde se komponenty nejnižší úrovně spolu testují nejdříve a pokračuje se směrem ke komponentám vyšší úrovně. Top down testování je přístup, kde se postupuje od spojení komponent nejvyšší úrovně směrem k testování komponent nejnižší úrovně. Tyto testy jsou vytvářeny jak vývojáři tak testery. Mohou být napsány, jakmile je hotová podrobná specifikace.

2.2.3 Systémové testování

Jakmile je ukončeno integrační testování a jsou opraveny nalezené chyby, může se přistoupit k systémovému testování. Systémové testování je první fáze testování, které probíhá v prostředí, ve kterém bude software používán. Kompletní sestavený program je testován na dodržení funkcionálních požadavků, uplatňují se zde techniky black-box testování. Dále zde popíšeme nejběžnější testovací metody:

Zátěžové testování

Zátěžové testování zkouší modelovat obvyklé maximální zatížení systému.

Stress testování

Testuje chování a funkčnost programu v extrémních podmínkách. Liší se od zátěžového testování tím, že jde ještě za hranici předpokládaného maximálního zatížení systému. Například zpracování velkého souboru, velký počet uživatelů atd. záleží na konkrétním úkolu programu.

Performance testování

Testování výkonu aplikace. Například vytíženost paměti pro několik operací najednou.

Smoke testování

Slouží pro rychlý test zda vůbec lze aplikaci nainstalovat a spustit na ní základní úlohy. Pokud takový test skončí neúspěchem nemá smysl pokračovat v dalším testování.

Sanity testování

Má za úkol otestovat základní funkcionalitu programu. Sanity testy bývají obvyklé jedny z prvních, které se při testování spouštějí.

Scenario testování

Slouží k vykonání testů, které demonstrují reálné použití programu. Obvykle to bývá kombinace voleb z testování funkcionality programu.

Regresní testování

Toto testování je prováděno jakmile se učiní změna v kódu, typicky při vydání nových verzí programu. Taková změna může zapříčinit, že funkce, které dříve fungovaly, najednou fungovat přestanou. Takové chybě se říká regresní chyba.

GUI testování

Je testování grafického uživatelského rozhraní. Otestovat veškerou funkcionalitu GUI bývá obtížné, protože i relativně malé programy mají velké množství operací pro jejich ovládání. Tyto testy je nezbytné automatizovat.

Risk-based testování

Spočívá v analyzování, jak v programu vyvolat chybu. Testy jsou navrhovány tak, aby takové chyby odhalily.

Security testování

Testování, které je zaměřeno na hledání zranitelností v programu.

2.2.4 Acceptance testování

Acceptance testování se liší od systémového testování tím, že je testováno obvykle koncovým uživatelem. Je prováděno v koncovém prostředí a jeho cílem je zjistit jestli je software připravený pro reálné používání.

2.3 Testovací přístupy

Hlavní přistupy pro vytváření testů jsou white-box a black-box testování a liší se přístupem, jakým je na testovaný objekt nahlíženo.

2.3.1 White-box testování

White-box testování nebo také strukturální testování je přístup, ve ktrerém používáme znalost vnitřní struktury programu, při vytváření testů zkoumáme zdrojový kód programu. Proto jsou tyto testy obvykle vytvářeny programátorem. Nejčastěji probíhá na úrovni unit testování. Mezi techniky používané při funkcionálním testování patří *path* testování, *control flow* testování a *data flow* testování.

Obrázek 2.1: White-box testování

Path testování spočívá v ověření, že všechny nezávislé cesty v programu byly vykonány alespoň jednou. Nezávislými cestami se myslí každá cesta, která představuje novou podmínku v programu. K zobrazení a určení nezávislých cest se používají grafy toku. Všechny vybrané nezávislé cesty pak představují minimální testy, které by se měly vytvořit pro white-box testování.

Control flow testování je také reprezentováno grafy toku. Úkolem je ověřit, že máme odpovídající skupinu testů. V této souvislosti se uvádí metrika zvaná coverage (pokrytí). Je využívána ke zjištění, jestli vybíráme sadu testů, které spouštějí co největší část kódu. Rozlišujeme tři základní typy pokrytí. *Statement coverage*, které měří pokrytí všech výrazů v programu. *Branch coverage* měří pokrytí podmínek v kódu, které jsou vyhodnoceny jednou jako pravdivé a jednou jako nepravdivé. A nakonec *condition coverage*, které kromě podmínek vyhodnocuje i vnořené booleovské výrazy, na pravdu i nepravdu [22]

Data flow testování je založeno na kontrole toho, jak jsou proměnné v programu definovány a jak jsou používány v programu.

Mezi výhody white-box přístupu patří, že lze snadno určit testovací data, která budou program testovat efektivně a optimalizace kódu. Nevýhodou je, že k vytváření testů je potřeba tester který rozumí kódu, což zvyšuje cenu samotných testů.

2.3.2 Black-box testování

Black-box testování nebo také funkcionální testování je přístup bez znalosti vnitřní struktury programu. Při vytváření testů zkoumáme specifikaci programu, tedy nepracujeme se zdrojovým kódem. Tyto testy má už obvykle na starosti testovací tým. Tato metoda testování je využívána od integračního testování k vyšším úrovním.

Při použití black-box metod je na testovaný objekt je pohlíženo jako na uzavřenou krabici, testování je založeno na zadávání vstupů a kontrole výstupů z programu. Do black-box testování patří následující techniky - *equivalence partitioning*, *boundary value analysing* a *rozhodovací tabulka*. Dále si ty to techniky popíšeme.

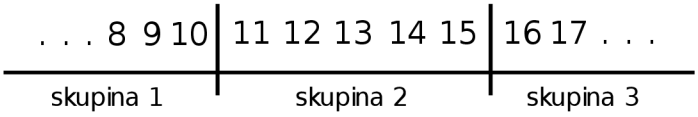
Equivalence partitioning & boundary value analysing

Tyto dvě techniky spolu úzce souvisejí. Úkolem techniky equivalence partitioning je rozpoznat všechny možné vstupní hodnoty a rozdělit je do skupin tak, abychom odchytili všechny možné scénáře, přičemž hodnoty v každé skupině produkují stejný druh výstupu. Dalším

Obrázek 2.2: Black-box testování

krokem je určit hodnoty, které nejlépe reprezentují danou skupinu a snížit tak počet potřebných testů na minimum. Hodnotami, které nejlépe reprezentují danou skupinu, jsou obvykle hodnoty hraniční, protože v hraničních hodnotách je program nejvíce náchylný na chyby.

Jako příklad si uveďme podmínku $10 < x <= 15$. Pak bychom mohli možné hodnoty x rozdělit do skupin podle Obrázku 2.3. Hraničními hodnotami pro validní vstupy by se staly čísla 10 a 16, protože se jedná o krajní hodnoty měnící platnost celé podmínky, a tak i výpočet programu.



Obrázek 2.3: Příklad equivalence partitioning

Výhodami při spojení těchto technik je redukování počtu testů na minimum a snadnost k naučení. Nevýhodou je, že chyby se nemusí nacházet jen v hraničních hodnotách, takové chyby pak tyto techniky neodhalí. Díky nejednoznačné specifikaci také můžou být špatně určeny hraniční hodnoty. Další věcí, na kterou je třeba pamatovat je, že ne vždy je rozdělení vstupů do skupin tak zřejmé jako v našem příkladě [10].

Rozhodovací tabulka

Rozhodovací tabulka (angl. decision table) je jednou z dalších technik, která je využívaná při vytváření testů. Slouží k vytváření testů s různou kombinací vstupů. Obsahuje podmínky a na ně navazující akce. Hodnoty v této tabulce jsou často reprezentovány N/Y (0/1 nebo true/false) hodnotou. Příklad rozhodovací tabulky můžeme vidět v Tabulce 2.1. Každý sloupec hodnot představuje jejich jedinečnou kombinaci a měli by tvořit jeden test.

Tento text se dále věnuje vytváření testů technikami black-box testování.

2.4 Automatizace testování

Rozlišujeme dva přístupy k provádění testů a to buď manuální nebo automatizovaný přístup. Automatizace testů spočívá v automatickém provádění testů. Automatizace v pravém slova smyslu přijde na řadu až ve fázi spouštění testů, jejich vykonávání a zobrazování výsledků.

	test1	test2	test3
Podmínky			
Vajíčka nejsou napadena	N	N	Y
Ptačice zůstane v hnízdě	N	Y	Y
Akce			
Ptáčata budou žít	N	Y	Y

Tabulka 2.1: Příklad rozhodovací tabulky - v tabulce jsou popsány všechny reálně možné kombinace tří podmínek. Každý sloupec tvoří jeden test. Například test2 testuje případ kdy vajíčka napadena jsou, ptačice zůstane v hnízdě a ptáčata budou žít.

Dále se budeme věnovat přístupu automatizovanému.

Automatizaci testování si můžeme definovat jako proces psaní programu nebo skriptu, který provádí úkony, které by jinak musely být prováděny manuálně. Mezi výhody automatizace testů patří úspora času jak testera tak vývojáře v případě, že testy by měly být prováděny opakovaně. Další výhodou je, že spouštění takových testů je levnější než manuální testování a vyhodnocení výsledků testů je přesnější.

V souvislosti s automatizací testů vznikají i různé testovací systémy, lépe známy pod anglickým termínem test automation framework, které navíc testy automatizovaně spouští a poté sami reportují výsledky a upozorní na problém. Mezi takové systémy patří například Novel Software Automated Testing System [17], či Software Testing Automation Framework [2] od IBM. Tato práce se zabývá testovacím systémem Red Hat Test System od firmy Red Hat.

2.5 Automatizace testování GUI

Automatizace testování GUI kromě nesporné výhody úspory času, sebou přináší řadu problémů. Už jenom pokrýt celou funkcionalitu GUI i relativně malého programu obnáší stovky operací.

Dalším problémem je závislost akcí konkrétní úlohy k tomu, aby se provedla správně. Pokud je cílem například ve Wiresharku začít zachytávat pakety, je potřeba kliknout na volbu v menu Capture, která musí vyvolat Capture dialog, vybrat správné zařízení a potvrzení dialogu musí vyvolat zachytávání paketů. Pokud jedna z akcí selže například kvůli zamrznutí programu, selžou i všechny následující akce celého testu.

Jiná obtíž, se kterou se musí automatizace testování GUI, potýkat je proměnlivost GUI. Jeho komponenty se mohou přesunout nebo úplně zmizet, požadované akce nebo sekvence akcí se najednou provádějí jinak. Existují i další problémy, v této práci však není potřeba se o nich zmiňovat.

Existuje velké množství komerčních i open-source nástrojů pro automatizaci GUI. Mezi komerční produkty patří například TestComplete [6] od AutomatedQA pro Windows aplikace nebo GUIdancer pro Java aplikace [7], který nevyžaduje žádné zkušenosti s progra-

mováním, vše co je potřeba je vybrat komponentu vybrat akci a zadat vstupní data. K zástupcům open-source projektů patří Linux Desktop Testing Project (LDTP) [14] a Dogtail [15] od společnosti Red Hat. Oba jmenované open-source projekty jsou na sobě nezávislé a oba používají technologie usnadnění, tj. technologie, které mají za úkol pomáhat při práci s počítačem lidem s postižením. Odlišnost spočívá v tom, že Dogtail zjišťuje komponenty, se kterými pracuje za běhu, zatímco LDTP si vytváří mapy komponent ještě před spuštěním testu. Dogtail je psaný v jazyku Python, LDTP v převážně v jazyku C.

Pro vytvoření testů pro Wireshark jsem vybrala nástroj Dogtail převážně kvůli podpoře RHTS pro jeho integraci.

2.5.1 Dogtail

Dogtail je open-source framework určený pro automatizaci testování GUI napsaný v Pythonu. Používá technologie usnadnění (angl. assistive technologies) pro komunikaci s komponentami GUI. Použití Dogtail spočívá v napsání vlastních skriptů, které využívají jeho funkce, v jazyku Python.

Dogtail neidentifikuje komponenty jejich polohou, ale jejich metadaty, takže není závislý na přeskupení komponent v GUI. Technologie usnadnění, kterou využívá je součástí GNOME Accessibility Project [1]. Kromě GNOME aplikací podporuje i mnoho jiných aplikací založených na GTK+. Pro použití Dogtailu je potřeba mít nainstalované následující balíčky - AT-SPI (Assistive technology), Python 2.3 nebo vyšší, ImageMagick 6.2 nebo vyšší, ElementTree a pyspi [8]. Názvy balíčků se mohou v různých distribucích lišit.

Dogtail poskytuje utilitu `sniff`, která zobrazuje strom elementů (komponent) každé spuštěné aplikace. Může být využita pro hledání jmen komponent, jejich popisu, a pro zjištění povolených akcí, které se dají s komponentou dělat. Tyto získané informace se dají dále využít při psaní testů, když se na komponenty odkazujeme.

Dogtail lze rozdělit na dvě části - jádro dogtailu a pomocnou knihovnu. Jádro obsahuje obecné utility, procedurální a objektové API. Pomocná knihovna obsahuje třídy a funkce specifické pro každou aplikaci a jejím cílem je učinit psaní testů snažší. Procedurální API využívá utilita `dogtail-recorder`. Je určena k nahrávání uživatelských akcí a tyto akce přepisuje do sekvence příkazů využívající právě procedurální API. Je možné ho využít vytvoření jednoduchých testů. Objektové API je určeno pro psaní spíše komplexnějších testů a pro vytváření tříd do pomocné knihovny.

Při vytváření testů bude použito právě objektové API.

Kapitola 3

Red Hat Test System

V této kapitole si představíme Red Hat Test System (dále jen RHTS), pro který budou v kapitole 4 navrhovány testy a jeho základní architekturu. Budou zde představeny prostředky, jakými lze tyto testy vytvářet, jejich možnosti spouštění a kontroly výsledků. Popis vychází z oficiální dokumentace RHTS, která není veřejně přístupná. Příbuzné informace jsou k nalezení na stránkách projektu Beaker, který je veřejně přístupnou verzí RHTS pro obecnou podporu testování linuxových aplikací [3].

3.1 Představení RHTS

Red Hat Test System je systém pro automatizaci testování programů pro distribuci Red Hat Enterprise Linux (RHEL). Zajišťuje programové rozhraní pro vývoj automatizovaných testů, reprodukci chyb, možnost otestování programů na různém hardware a provádění regresních testů.

Skládá se z několika komponent: plánovače testů, databáze, RHTS repozitáře, řídící jednotky pro cílové počítače, cílových počítačů a RHTS frameworku. Průběh vykonání testů je následovné: do RHTS repozitáře se uloží rpm balík s vytvořenou sadou testů. Po vytvoření testovací úlohy tuto úlohu převezme plánovač testů, který řídí veškeré úkony potřebné pro vykonání testu, jako výběr stroje na kterém budou testy probíhat, instalaci distribuce, její restart a přeinstalaci, pokud systém nemá žádnou odezvu a koordinaci testů. Výsledky z cílových laboratorních systémů jsou odeslány plánovači zpět, a ten je uloží do databáze pro pozdější prohlížení výsledků testů.

RHTS framework definuje API, formát metadat a zajišťuje prostředky pro vytváření spouštění a odesílání testů do RHTS repozitáře. Testy mohou být spouštěny dvěma způsoby - na pracovní stanici vývojáře nebo v prostředí laboratoře na vyhrazených systémech. Spouštění testů na vlastním stroji se používá především při vývoji testů. Po dokončení testů se odešlou do repozitáře balíčků , a pokud na nich není potřeba nic měnit, spouštějí se výhradně v prostředí laboratoře. Pro vývoj a spuštění testů na vlastním stroji je potřeba mít nainstalované tyto rpm balíky, které jsou volně ke stažení [4]: rhts-devel, rhts-test-repo, rhts-test-env a rhts-tools-repo.

3.2 Vytváření testů

Test psaný pro RHTS je skript nebo program, který provádí určité úlohy o jejichž úspěchu či neúspěchu podává zprávu. RHTS test se skládá minimálně ze tří souborů, prvním je shell skript runtest.sh, Makefile a dokumentační soubor PURPOSE.

runtest.sh

Je jádrem testu. Obsahuje samotný test nebo spouští jiný program nebo skript. Testy jsou tak nezávislé na programovacím jazyku a tester může použít ten, který se pro danou úlohu zrovna hodí. Každý test musí na začátku importovat rhts-enviroment.sh, což mu umožní používat proměnné RHTS API, které se starají o report výstupu a výsledků.

Makefile

Dalším souborem je makefile, který provádí několik úloh: kompiluje testy, vytváří rpm balíček s testem, pokud je potřeba stáhne externí soubory, které jsou potřeba pro vykonání testu, například vstupní data a v neposlední řadě spouští samotný test.

PURPOSE

Je textovým souborem, který slouží pro dokumentaci testu a pro zaznamenání věcí, které by uživatel testu měl vědět.

3.3 Nástroje RHTS

Pro vytváření testů poskytuje RHTS framework nástroje, které usnadňují jejich psaní a jejich uživatel pak není nucen se příliš zabývat implementačními detaily. Ještě uveďme, že pro psaní testů je potřeba minimálně znalost skriptování v shellu a dalších jazyků podle toho v čem budou testy psány.

3.3.1 rhts-wizard

Pro vytvoření testu je vhodné použít nástroj rhts-wizard, který je průvodcem při vytváření testu. Jedná se o textovou aplikaci pro terminál. Po zadání informací o typu testu, jeho popisu a počtu minut, po který se má test nechat běžet než bude ukončen, vytvoří všechny potřebné soubory a nastaví v nich potřebné cesty.

3.3.2 rhts-lib

Pro psaní testů se používá knihovna funkcí rhts-lib, která dělá z psaní testů téměř intuitivní záležitost. Obsahuje množství funkcí pro běžně používané operace při psaní testů. Nekteré kategorie funkcí si popíšeme:

Infrastruktura

Funkce pro zálohování souborů a adresářů, spouštění a restartování systémových služeb a pro mountování síťových disků.

Logování

Funkce pro jednotný záznam výsledků a report výsledků a především poskytuje funkce díky kterým je možné automaticky rozdělit celý test do podtestů a sdružovat tak například testy podobného zaměření do jednoho testu.

Testování

Funkce pro aritmetické porovnávání, testování existence souborů, pro spuštění programu s argumenty a watchdog, který spustí program a po určeném čase ho ukončí.

Rpm

Pro kontrolu přítomnosti požadovaného balíku.

Níže je ukázka části souboru runtest.sh, který používá funkce rhts-library. Je to ukázka inicializace testu, ve kterém se ověří přítomnost testovaného balíku (řádek 4), vytvoří se odkládací soubor (řádek 5) a inicializují se proměnné, které jsou využívány v další části testu (řádek 6 a 7). Nakonec se vytiskne log testu na standartní výstup (řádek 10). Celý tento blok Setup (řádek 2-9) se vyhodnotí jako jeden výsledek celého testu tedy jako jeden podtest.

```
1   rlJournalStart
2       rlPhaseStartSetup Setup
3
4           rlAssertRpm $PACKAGE
5           rlRun ,,tmpfile=\‘mktemp\‘‘‘ 0 ,,Creating tmp file‘‘
6           TESTOUT=../testsetpackets.pcap
7           GZTESTOUT=../testsetpackets.gz
8
9       rlPhaseEnd
10  rlJournalPrintText
```

3.4 Použití

Vytvořené testy po odeslání do RHTS repozitáře můžeme spustit přímo v RHTS.

3.4.1 Spuštění testů

Při spuštění testů v RHTS máme dvě možnosti. Zadání spuštění testu přes příkazovou řádku, pro což musíme mít nainstalován vývojový balíček RHTS nebo přes webové uživatelské rozhraní. V obou případech musíme určit balíček, který se bude testovat, typ testu, architekturu na které test má běžet, konkrétní testy které se mají spustit a verzi distribuce, na které se mají spouštět. Můžeme zvolit i testování více architektur najednou, či vykonání všech dostupných testů pro požadovaný balíček.

Odesílání požadavků probíhá pomocí skriptu zvaného `workflow`. Existuje několik druhů těchto skriptů. Druh `workflow` volíme podle druhu testu, který chceme spustit. Například pro spuštění jednoho nebo více testů na specifickém hardware zvolíme jednoduchý

`workflow`, při regresním testování balíku zvolíme porovnávací `workflow`, který porovná výsledky testů mezi starou a novou verzí balíku.

Po odeslání požadavku `workflow` kontaktuje plánovač, zadá mu požadavek na vytvoření úlohy a je vrácena identifikace úlohy, přes kterou můžeme později přistupovat k výsledkům testů. Po přijetí úlohy serverem začne instalace zvolené distribuce a po nainstalování potřebných balíků se spustí samotné testy.

3.4.2 Prohlížení výsledků

Sledovat průběh testů můžeme ve webovém uživatelském rozhraní po zadání čísla úlohy. Můžeme sledovat stav celé úlohy, kdy se teprve instaluje operační systém i stav jednotlivých testů. Je možné si také zobrazit log z každého testu.

Kapitola 4

Návrh testů a jejich implementace

V této kapitole bude představena aplikace Wireshark a obsah balíků této aplikace v distribuci RHEL. Pro každý balík bude následovat analýza již vytvořených testů a jejich možné využití. Budou navrženy oblasti k dalšímu testování. Dále budou popsány konkrétní testy pro každou testovací oblast, jejich popis, vstupy a očekávané výstupy.

4.1 Aplikace Wireshark

Wireshark je open-source program pro analýzu síťových paketů. V distribuci RHEL je tato aplikace rozdělena do dvou balíků. Balík Wireshark jehož základem je knihovna libpcap pro zachytávání paketů, obsahuje nástroje příkazové řádky, zásuvné moduly (angl. plugins) a dokumentaci. Druhým balíkem je balík Wireshark-gnome, který obsahuje grafické uživatelské rozhraní (GUI) pro Wireshark. Dále budeme tyto dva balíky analyzovat odděleně.

Hlavními úkoly Wiresharku je zachytávání paketů na specifikovaném síťovém rozhraní a zobrazení získaných dat. Wireshark umožňuje kromě zachytávání paketů i další funkce pro práci se získanými daty, jako zobrazení detailních informací o vybraných paketech, ukládání získaných dat, import a export dat do jiných programů, filtrování paketů, vyhledávaní paketů podle mnoha kritérií, obarvování paketů podle filtrů a zobrazování různých statistik. Je dostupný ve verzích pro UNIX a Windows. Testy budou zaměřeny na UNIX verzi Wiresharku.

4.2 Balík Wireshark-gnome

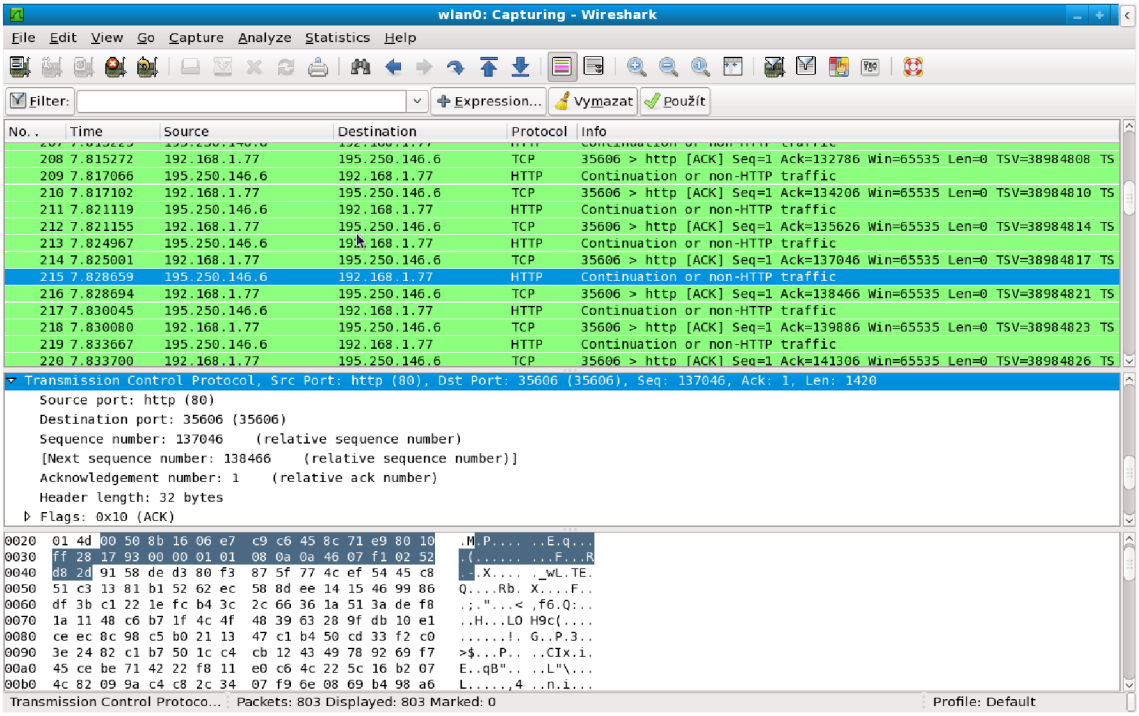
Balík Wireshark-gnome je grafickou nadstavbou wiresharku. Zajišťuje kompletní GUI pro ovládání mnoha jeho funkcí. Při každé změně balíku je potřeba testovat základní funkcionalitu GUI a cílem těchto testů je tuto úlohu automatizovat.

4.2.1 Představení GUI

V této podkapitole si krátce představíme grafické uživatelské rozhraní Wiresharku.

Obrazovka je rozdělena na tři části - horní část je pro přehled zaznamenaných paketů, prostřední zobrazuje informace o vybraném paketu a dolní ukazuje konkrétní data vybraného paketu hexadecimálním formátu.

Položka menu Capture slouží pro nastavení zachytávání paketů. Umožňuje vybrat zařízení, podmínky stopu, do jakého souboru se mají získaná data uložit atd. Menu Go obsahuje položky pro nalezení specifického paketu. Položky menu Analyze umožňují nastavení zobrazovacího filtru a maker. Menu Statistics obsahuje položky pro zobrazení různých statistik například grafů vytížení sítě a statistik různých protokolů.



Obrázek 4.1: Wireshark - GUI

4.2.2 Testy

Při návrhu testů bylo zjištěno, že testy, které by testovali GUI pomocí Dogtail skriptů ještě navrženy nebyly, proto bylo potřeba začít základy.

Testy jsou navrhnuty tak, aby obsáhly množství obvyklých úloh, ke kterým se wireshark používá. Jsou zaměřeny na testování odpovědi GUI na požadované akce, tedy netestují funkce samotné. Na testování funkcí budou zaměřeny testy na balík Wireshark. Testovanými oblastmi budou zachytávání paketů, použití zachytávacích filtrů, procházení paketů, restartování úlohy, z analýzy potom použití zobrazovacích filtrů, maker pro filtry, zobrazení IO graphů, barvení paketů a zobrazení jednoduchých dialogů jako nápověda atd.

Sada testů pro zachytávání paketů

Test 1	
Popis testu:	Začne zachytávat pakety na určeném rozhraní.
Vstup:	Rozhraní, na kterém se budou zachytávat pakety.
Očekávaný výsledek:	Na zadané rozhraní přijdou pakety.
Aktuální výsledek:	PASS
Test 2	
Popis testu:	Test na zachytávací filtr.
Vstup:	Nový filter myfilter - dst 127.0.0.1
Očekávaný výsledek:	Filtruje pakety
Aktuální výsledek:	PASS
Test 3	
Popis testu:	Restartuje zachytávání paketů.
Vstup:	Rozhraní, na kterém se budou zachytávat pakety.
Očekávaný výsledek:	Proveden restart.
Aktuální výsledek:	PASS

Testování volby menu Go

Test 4	
Popis testu:	Vyzkouší všechny způsoby procházení paketů.
Vstup:	Zachycené pakety.
Očekávaný výsledek:	Projití všech voleb.
Aktuální výsledek:	PASS

Sada testů pro analýzu paketů

Test 5	
Popis testu:	Na sadě zachycených paketů zadá zobrazovací filtr.
Vstup:	Rozhraní, na kterém se budou zachytávat pakety, filtr.
Očekávaný výsledek:	Na zadané rozhraní přijdou pakety, poté se vyfiltrují.
Aktuální výsledek:	PASS
Test 6	
Popis testu:	Vytvoří makro na zobrazovací filtr.
Vstup:	Nové makro a pakety.
Očekávaný výsledek:	Makro vyfiltruje pakety.
Aktuální výsledek:	PASS

Test pro statistiku paketů

Test 7	
Popis testu:	Vyzkouší nastavit zobrazení IO grafu
Vstup:	Rozhraní, na kterém se budou zachytávat pakety a grafy kterých protokolů se budou vykreslovat.
Očekávaný výsledek:	Okno s grafem.
Aktuální výsledek:	PASS

Sada testů pro volbu menu View

Test 8	
Popis testu:	Vyzkouší nastavit barvu jednoho typu paketu.
Vstup:	Jméno a barvy pro nový paket.
Očekávaný výsledek:	Vybraný typ paketu má nastavenou barvu.
Aktuální výsledek:	PASS

Sada testů pro volbu menu About

Test 9	
Popis testu:	Zobrazí dialog 0 aplikaci.
Vstup:	
Očekávaný výsledek:	Dialog bude nalezen a zobrazen.
Aktuální výsledek:	PASS
Test 10	
Popis testu:	Zobrazí dialog o podporovaných protokolech.
Vstup:	
Očekávaný výsledek:	Dialog bude nalezen a zobrazen.
Aktuální výsledek:	PASS

4.2.3 Implementace testů

Tyto testy byly napsány v jazyce Python, s využitím frameworku Dogtail pro automatizované testování GUI. Organizaci samotných testů a kontrolu výsledků zajišťuje Python modul `unittest`. Pro účely snadnějšího testování Wiresharku byl vytvořen skript `wireshark_utils.py`, který obsahuje třídu `wiresharkapp` s metodami pro základní úlohy. Testy jsou rozděleny do šesti logických celků: zachytávací testy, testy pro analýzu paketů, test procházení paketů, statistiky, testy nastavení zobrazení a testy zobrazení informačních dialogů.

Pro účely ručního spouštění testů byla vytvořena stručná nápověda ke spuštění testů.

```
# ./wireshark.py --help
Usage: wireshark.py [options] arg

Options:
  -h, --help            show this help message and exit
  -f FILE, --file=FILE  write log to FILE
  -s SUITE, --suite=SUITE
                        choose SUITE to test, possible options are: capture,
                        go, about, analyzing, statistics, view. Default
                        option - all tests will start.
  -d DEVICE, --device=DEVICE
                        enter device with traffic
```

4.3 Balík Wireshark

Balík Wireshark obsahuje řadu nástrojů příkazové řádky, jejichž použití je vhodné při úzce specializovaných úkolech. Mimo jiné jsou jimi `capinfos`, `editcap`, `mergecap` a `tethereal`.

Capinfos je utilita, která tiskne informace o zadaném binárním souboru nebo souborech s pakety. Podporuje velké množství formátů souborů i z jiných paketových analyzátorů. Poskytuje informace o počtu paketů, typu souboru, průměrné velikosti paketů, velikosti souboru a množství dalších. Editcap poskytuje nástroje k odstraňování nebo výběru vybraných paketů ze souboru a může být také použit pro konverzi souboru na jiný formát. Mergecap je program pro slučování několika souborů s pakety do jednoho souboru. Obsahuje volby pro konverzi různých typů protokolů linkové vrstvy na společný formát. Tethereal je komplexní nástroj pro zachytávání paketů, jejich analýzu a ukládání.

4.3.1 Analýza

Nejrizikovějšími oblastmi pro vznik chyb, jsou takové části kódu, které jsou nejčastěji měněny nebo takové, které jsou naopak používány nejméně. Části, které jsou měněny často mohou způsobit tzv. vedlejší efekt tím, že například změna ve funkci může způsobit její nefunkčnost při jejím volání v kódu jinde. Obecně může platit, že čím častěji je nějaká funkce používána, tím je menší pravděpodobnost, že v ní bude objevena nová chyba. Naopak čím je funkce pokročilejší, netradičnější, složitější na použití, či neoblíbená funkce je, tím je větší riziko, že obsahuje chyby, protože ji jednoduše nikdo nepoužívá.

Mezi nejrizikovější oblast Wiresharku patří chyby dissectorů. Dissector má na starosti vlastní získání informací z paketu. O tom jaký protokolový dissector se použije rozhoduje Wireshark podle portu, na který paket přijde.

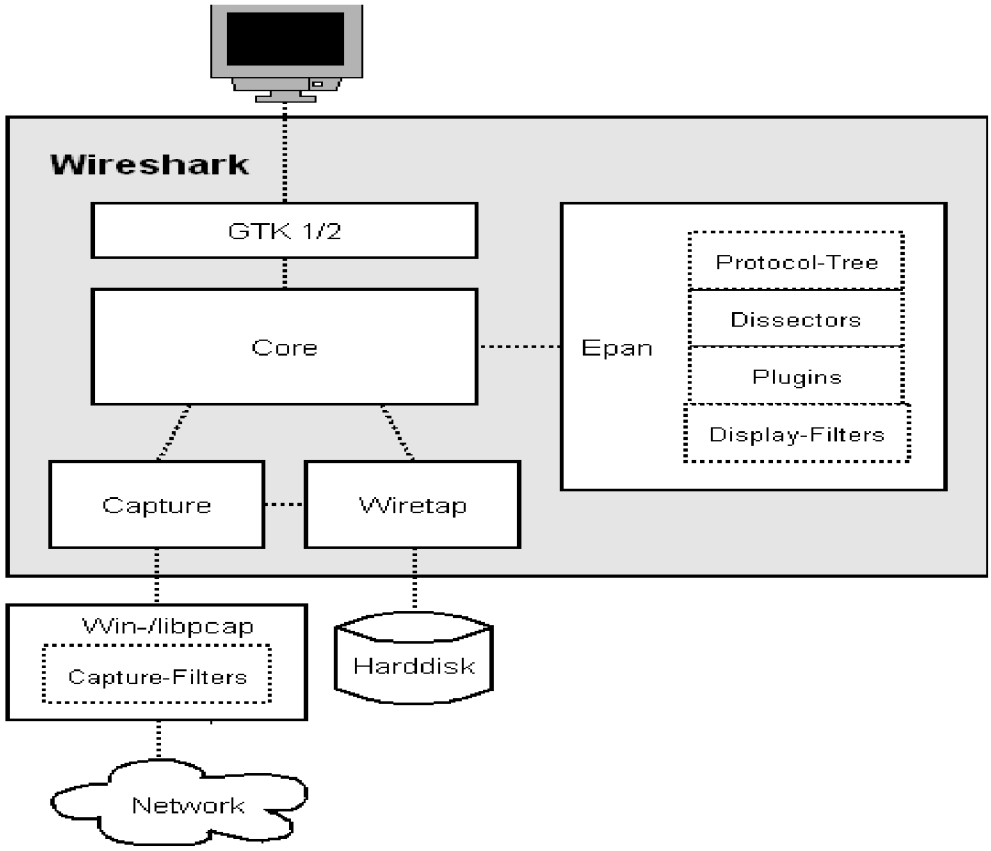
Dále bude vysvětlena úloha dissectorů ve Wiresharku podle Obrázku 4.2. Základem Wiresharku je síťová knihovna `libpcap`, která zajišťuje zachycování paketů ze síťových rozhraní. Knihovna `wiretap` má na starosti zapisování/čtení zachycených paketů do/ze souboru. Dále obsahuje komponentu analyzátor paketů, který kromě analyzování paketů z nich také čte všechny dostupné informace. Paketový analyzátor obsahuje kromě jiných komponent i dissectory. Každý protokol má svůj vlastní dissector.

Jakmile přijde paket ve formátu, ve kterém není očekávaný, dissector ho neumí zpracovat a důsledkem je zamrznutí aplikace či její zahlcení. Paket může být buď poškozený nebo mít vadnou délku, může přijít na neobvyklý port, díky čemuž je zpracováván nesprávným dissectorem atd.

Mezi další oblasti, pro které budou vytvořeny testy patří vlastní funkcionalita výše představených nástrojů Wiresharku. Vytvořeny budou jak testy na ověření, že daná funkce opravdu dělá co má, ale také testy, které se budou snažit vyvolávat chybové stavy a kontrolovat jak si s tím funkce poradí.

Poslední kategorií bude vytvořit regresní security testy, které budou číst soubory paketů, které by měli daný bezpečnostní problém reprodukovat.

Při návrhu testů jsem uplatnila metody sanity testování, risk-based testování, scenario testování a security testování.



Obrázek 4.2: Základní architektura Wiresharku [23]

4.3.2 Testy

Na balík Wireshark jsou již některé testy vytvořeny. Tyto testy pocházejí přímo od vývojářů Wiresharku (upstreamu) a z RHTS.

Testy z upstreamu se věnují tetherealu, operacím souvisejícím se zachytáváním paketů na síťovém rozhraní, práci se soubory a jiným věcem. Tyto testy se ovšem nedají použít pro spuštění v RHTS. Hlavním problémem je, že mají textové uživatelské rozhraní a výsledky testů se nedají rozumně odchytávat. Upstream testy jsou proto využity alespoň ve formě určení testovacích oblastí, které jsou realizovány v testovacích skriptech.

V RHTS již existují některé testy pro testování Wiresharku. Patří mezi ně i test, jehož úkolem je číst pakety vygenerované Wireshark utilitou `randpkt`, která podporuje asi 20 různých typů paketů jako ICMP, UDP, DNS, ARP, syslog atd. Dalším testem je testování zobrazovacích filtrů Wiresharku metodou white-box testování, který zahrnuje různé typy paketů a na každý typ aplikuje příklady filtrů pro daný protokol tak, aby testoval možné cesty ve zdrojovém kódu. Obsahuje pakety typu IPX RIP response, IPv6, ARP, NFS, NTP, HTTP a TFTP.

V dalších podkapitolách budou pro každou testovací metodu vedeny testy, které ji budou

aplikovat. Pro každý test bude uveden jeho popis, vstupy, očekáváné a aktuální výsledky. Uvedené výsledky testů jsou platné pro verzi Wiresharku 1.0.6.

4.3.3 Sanity a scenario testy

Pro testování základní funkčnosti bude pro každou utilitu vytvořena sada sanity testů, které by měly pokrývat většinu funkcí uvedených programů. Do kategorie sanity pak můžeme ještě zařadit sadu scenario testů, které budou obsahovat testy na praktické využití a kombinaci parametrů.

Sada testů pro tethereal - zachytávání paketů

Test 1	
Popis testu:	Zaznamenat 10 paketů.
Vstupy:	`# tethereal -i 1 -c 10 -w $Tmpfile` `# tethereal -i 1 -c 10 -w -`
Očekávaný výsledek:	Program přijme 10 paketů a zapíše je do souboru. Program přijme 10 paketů a a zapíše je na stdout.
Aktuální výsledek:	**PASS**
Test 2	
Popis testu:	Zachytávací filtr.
Vstupy:	`# tethereal -i 1 -f icmp -c 10 -w $Tmpfile`
Očekávaný výsledek:	Soubor bude obsahovat pouze icmp pakety.
Aktuální výsledek:	**PASS**
Test 3	
Popis testu:	Podmínky stopu.
Vstupy:	`# tethereal -i 1 -a filesize:1 -w $Tmpfile` `# tethereal -i 1 -a files:2 -b filesize:1 -w my`
Očekávaný výsledek:	Program se ukončí jakmile soubor dosáhne velikosti 1kb. Jakmile získá 2 soubory velikosti 1kb.
Aktuální výsledek:	**PASS**
Test 4	
Popis testu:	Velikost ukládaného paketu - snapshot length.
Vstup:	`# tethereal -i 1 -s $mylength -c 5 -w $Tmpfile`
Očekávaný výsledek:	Velikost každého paketu v souboru nepřekročí velikost v bajtech definovanou parametrem -s.
Aktuální výsledek:	**FAIL - nejmenší velikost ukládaného paketu je 68 bajtů**

Sada testů pro tethereal - zpracování souborů

Test 4

Popis testu:	Čtení souborů.
Vstupy:	`# tethereal -r $TESTFILE`
	`# tethereal -r $TESTFILE -w $Tmpfile`
	`# tethereal -r $TESTFILE -w -`
	`# tethereal -r $TESTFILEGZ -w -`
Očekávaný výsledek:	Přečte obsah souboru.
	Přečte obsah souboru a zapíše ho do jiného.
	Přečte obsah souboru a vypíše ho na stdout.
	Přečte obsah komprimovaného gz souboru a vypíše ho na stdout.
Aktuální výsledek:	**PASS**

Sada testů pro capinfos

Test 5

Popis testu:	Zobrazí všechny informace o souboru.
Vstupy:	`# capinfos $TESTFILE`
	`# capinfos $TESTFILE $TESTFILE2`
	`# capinfos $GZTESTFILE`
	`# capinfos -$option $TESTOUT`
Očekávaný výsledek:	Zobrazí všechny informace o souboru.
	Zobrazí všechny informace o dvou souborech najednou.
	Zobrazí informace o komprimovaném souboru.
	Postupně vyzkouší každý přepínač zvlášť.
Aktuální výsledek:	**PASS**

Sada testů pro editcap

Test 6

Popis testu:	Výběr paketů.
Vstupy:	`# editcap -A '2009-04-17 17:42:53'`
	`-B '2009-04-17 17:42:54' $TESTPCAP $Tmpfile`
	`# editcap -r $TESTPCAP $Tmpfile 1-5 40-45`
	`# editcap -r $TESTPCAP $Tmpfile 40-45`
	`# editcap $TESTPCAP $Tmpfile 40-45`
	`# editcap $TESTPCAP $Tmpfile 1-5`
Očekávaný výsledek:	Vybere pakety jen ze zadaného časového intervalu.
	Vybere pakety ze zadaných rozmezí.
	Vybere pakety z komprimovaného souboru.
	Zahodí pakety vybrané z prostředku souboru.
	Zahodí pakety vybrané ze začátku souboru.
Aktuální výsledek:	**PASS**

Test 7

Popis testu:	Změna časového razítka paketu.
Vstupy:	`# editcap -t -3600 $TESTPCAP $tmpfile` `# editcap -t 3600 $TESTPCAP $tmpfile`
Očekávaný výsledek:	Odečte od časového razítka hodinu. Přičte k časovému razítku hodinu.
Aktuální výsledek:	**PASS**

Sada testů pro mergecap

Test 8

Popis testu:	Spojuje soubory s pakety různých typů protokolů linkové vrstvy.

Vstupy:	`# mergecap -v -T ppp -w $Tmpfile $Tmpfile1 $Tmpfile2`
	`# mergecap -v -T ieee-802-11-radiotap -w $Tmpfile $Tmpfile1 $Tmpfile2`
	`# mergecap -v -T ether -w $Tmpfile $Tmpfile1 $Tmpfile2`

Očekávaný výsledek:	Spojuje pakety typu Point-to-Point protokol. Spojuje pakety typu IEEE 802.11 plus radiotap WLAN header. Spojuje pakety typu Ethernet.
Aktuální výsledek:	**PASS**

Scenario testy

Tento test využívá nástrojů tethereal, editcap, mergecap a capinfos. Obsahuje akce jako zachytávání paketů do více souborů, zjišťování informací o těchto souborech, vybírá část paketů z každého souboru a spojuje je zase dohromady.

4.3.4 Risk-based testy

Risk-based orientované testy budou kromě testů na nevalidní vstupy obsahovat i speciální případy paketů. Do této kategorie jsou zařazeny testy, které nemají bezpečnostní důsledky.

Sada testů nevalidní vstupy - tethereal

Test 9

Popis testu:	Nevalidní vstupy.

Vstupy:	`# tethereal -r none-existing-file.pcap`
	`# tethereal -$option`
	`# tethereal -f 'jkghg' -w './testout.pcap'`
	`# tethereal -N $i`
	`# tethereal -R 'jkghg' -w './testout.pcap`

Očekávaný výsledek:	Oznámí, že soubor neexistuje. Na chybné volby reaguje chybovou hláškou. Na zadaný chybný filtr zareaguje chybovou hláškou Nevalidní volby pro rezoluci jmen. Nevalidní zobrazovací filtr.
Aktuální výsledek:	**FAIL** - po zadání špatného filtru vrátí návratový kód 0

Test 10	
Popis testu:	Nevalidní rozhraní.

Vstupy:	`# tethereal -i bad-interface`
	`# tethereal -i 0`
	`# tethereal -i 4`

Očekávaný výsledek:	Oznámí špatné rozhraní, vrátí chybný návratový kód. Index rozhraní mimo rozsah. Index rozhraní mimo rozsah.
Aktuální výsledek:	**FAIL** - po zadání špatného jména rozhraní vrátí návratový kód 0

Sada testů speciální pakety - tethereal

Pro tyto testy byli vyhledány soubory, které obsahují nestandartní či poškozené pakety. Tethereal se je pak zkouší přečíst. Vstupem je příkaz `# tethereal -r $TESTFILE`. Očekávaným výstupem je indikace úspěchu přečtení souboru.

Speciální pakety

- DNS chyby
- Teardrop útok obsahující pakety s přesahujícími IP fragmenty
- chybové pakety Certificate Management protokolu
- MPEG2 Transport Stream protokol - příklad se ztraceným paketem
- UDP-lite protocol - 3 pakety s coverage délkou větší nez délkou paketu, checksum coverage hodnota mimo rozsah, nelegální checksum - 0

Sada testů - mergecap

Test 11	
Popis testu:	Kombinace protokolů linkové vrstvy s defaultním zapouzdřením.
Vstupy:	`# mergecap -v -w $Tmpfile $pppenc $wifienc`
Očekávaný výsledek:	Použije defaultní typ zapouzdření WTAP_ENCAP_PER_PACKET a skončí chybou
Aktuální výsledek:	**PASS**

4.3.5 Security testy

Security testy budou založeny na reprodukování Common Vulnerabilities and Exposures (CVE) chyb. CVE je standart pro zaznamenávání veřejně známých bezpečnostních děr. Do této skupiny jsou zařazeny testy, které nejen že způsobí pád aplikace, ale chyby, které nastanou mohou mít bezpečnostní následky. Například přístup k již uvolněným datům, zapisování více dat do bufferu než kolik je pro něj alokováno paměti atd. Požadavkem na tuto sadu testů je, aby bylo možné snadno přidávat budoucí nalezené chyby.

4.4 Implementace testů

Tyto testy byly napsány v bashi a využívají knihovnu funkcí rhts-lib. Jsou organizovány podle druhu testu do kategorií sanity, crash a security. Dále jsou děleny podle programu,

který testují. Testy byli implementovány podle jejich návrhu, který byl popsán v podkapitolách 4.3.3, 4.3.4 a 4.3.5.

Vstupem testů, které testují funkce pro zpracování zachycených dat, byl soubor se záznamem bežného provozu sítě.

Kapitola 5

Závěr

Tato práce se zabývala problematikou softwarového testování a vytvářením testů pro programové balíky distribuce Ret Hat Enterprise Linux.

V teoretickém úvodu bylo testování zařazeno z pohledu vývojového cyklu software, objasněny testovací techniky testování, úrovně testování a jejich případy užití. Podrorobněji byly popsány metody white-box a black-box testování, a určen cíl práce - vytvořit testy využívající metody black-box.

Byly vysvětleny důvody pro potřebu automatizovaného testování a byly představeny systémy pro automatizování testů a podrobně popsán účel Red Hat Test System, jeho architektura, použití a nástroje pro vytváření testů rhts-wizard a rhts-lib. Vysvětlili jsme si vytváření testů pro automatizované testování GUI jeho výhody a nevýhody. Uvedli jsme si řadu komerčních a open-source nástrojů z nichž byl pro vlastní realizaci testů vybrán framework Dogtail.

Dále byly představeny balíčky vybrané pro vytvoření testů Wireshark a Wireshark-gnome. Oba byly popsány a byly určeny oblasti, kterých se bude týkat návrh testů. Na základě tohoto rozboru a na základě teoretických znalostí metod testů byli navrženy konkrétní testy pro každý balíček.

Ve vlastní realizaci byla vytvořena sada testů na balíček Wireshark-gnome s využitím frameworku Dogtail a byla vytvořena třída `wiresharkapp`, která má usnadňovat vytváření a přidávání dalších testů. V testu je počítáno i s ručním spouštěním, kdy si uživatel může vybrat jakou oblast funkcí GUI chce testovat. Testy na Wireshark-gnome byly po té integrovány do RHTS.

Implementace testů pro balíček Wireshark byla provedena s využitím knihovny rhts-lib. Testy se dělí do několika kategorií, sanity testy, scenario testy, risk-based testy a security testy. Testy byly po té integrovány a vyzkoušeny v RHTS.

Testy integrované do RHTS přispějí k větší efektivitě testování při každé změně balíčku, díky možnosti automatizace testovacího procesu.

5.1 Nalezené chyby

Vytvořené testy odhalily celkem tři chyby v programu tethereal. Ve dvou případech se jedná o návratové hodnoty při chybě programu. Jde o vrácení návratové hodnoty 0 při zadání špatného jména rozhraní a při zadání špatného formátu zachytávacího filtru. Takové chování je zavádějící v případě, že návratové hodnoty použijeme k detekování chyby v programu. Tyto chyby byly reportovány do systému Bugzilla:
`https://bugzilla.redhat.com/show_bug.cgi?id=501200.`

Další chyba se týká parametru `-s` udávající velikost ukládaného paketu v bajtech. Pokud na rozhraní přijde paket s velikostí větší než je zadaná velikost, paket se automaticky zkrátí na velikost požadovanou. Ovšem pokud je nastavená velikost parametrem `-s` menší než 68 bajtů, toto chování již neplatí. Paket se vždy ořízne na minimální délku 68 bajtů. Toto chybové chování již bylo reportováno jako bug v minulosti [11], ve verzi 1.0.7 ještě přetrvává, ale ve vývojové verzi 1.1.3 je již ošetřené.

5.2 Návrh rozšíření

Vytvořené testy nejsou a ani neměli být vyčerpávající sadou testů pro aplikaci wireshark. Vytvářejí základ pro pozdější přidávání dalších testů.
Dalšími možnými návrhy pro testy v balíku wireshark-gnome jsou:

- změna dissectoru, který bude dekódovat zvolený paket
- volby menu „follow TCP/UDP/SSL stream“, které způsobí otevření nového okna a sledování paketů vybraného spojení
- uložení/otevření souboru
- statistika multicast proudu a dalších protokolů

Návrhy pro balík wireshark:

- test funkcí dalšího programu patřícího do tohoto balíku - `text2pcap`
- pokrytí zbývajících voleb testovaných programů
- regresní testy pro chyby, které nemají bezpečnostní charakter
- přidání dalších souborů s neobvyklými pakety

Literatura

[1] GNOME Accessibility. http://projects.gnome.org/accessibility, Naposledy navštíveno 14. 5. 2009.

[2] Software Testing Automation Framework (STAF). http://staf.sourceforge.net/index.php, Naposledy navštíveno 14. 5. 2009.

[3] Beaker [online]. https://fedorahosted.org/beaker, Naposledy navštíveno 8. 5. 2009.

[4] RHTS rpm balíky [online]. http://people.redhat.com/dmalcolm/tablecloth/tools, Naposledy navštíveno 8. 5. 2009.

[5] Uno Tool Synopsis [online]. http://spinroot.com/uno, Naposledy navštíveno 8. 5. 2009.

[6] AutomatedQA: TestComplete [online]. http://www.automatedqa.com/products/testcomplete, Naposledy navštíveno 8. 5. 2009.

[7] Bredex: GUIdancer [online]. http://www.bredex.de/en/home/first.html, Naposledy navštíveno 8. 5. 2009.

[8] DiMaggio, L.: Automated GUI testing with Dogtail [online]. http://www.redhat.com/magazine/020jun06/features/dogtail, Naposledy navštíveno 8. 5. 2009.

[9] Henzinger, T. A.; Beyern, D.; Majumdar, R.; aj.: BLAST: Berkeley Lazy Abstraction Software Verification Tool [online]. http://mtc.epfl.ch/software-tools/blast, Naposledy navštíveno 8. 5. 2009.

[10] Kaner, C.; Bach, J.: Black Box Software Testing: By Cem Kaner & James Bach [online]. http://www.testingeducation.org/BBST, Naposledy navštíveno 8. 5. 2009.

[11] Kucenski, M.: tshark/wireshark don't allow snaplen ¡ 68. https://bugs.wireshark.org/bugzilla/show_bug.cgi?id=2731, Naposledy navštíveno 14. 5. 2009.

[12] Marick, B.: *The Craft Of Software Testing, Subsystem Testing.* Prentice Hall PTR, 1995, ISBN 0-13-177411-5.

[13] Myers, G. J.: *The Art of Software Testing - second edition.* John Wiley & Sons, Inc., Hoboken, New Jersey, 2004, ISBN 0-471-46912-2.

[14] Project, G. D. T.: LDTP [online]. http://ldtp.freedesktop.org/wiki/About, Naposledy navštíveno 8. 5. 2009.

[15] Rousseau, E.; Cerza, Z.; Lee, C.; aj.: Dogtail - taking your application for a walk [online]. http://people.redhat.com/zcerza/dogtail/about.html, Naposledy navštíveno 8. 5. 2009.

[16] Secure Software, I.: Rats [online]. http://www.fortify.com/security-resources/rats.jsp, Naposledy navštíveno 8. 5. 2009.

[17] Songwen, P.; Baifeng, W.; Kun, Z.; aj.: Novel Software Automated Testing System Based on J2EE. http://dx.doi.org/10.1016/S1007-0214(07)70083-4, Naposledy navštíveno 14. 5. 2009.

[18] Wikipedia: Dynamic program analysis [online]. http://en.wikipedia.org/wiki/Dynamic_program_analysis, Naposledy navštíveno 8. 5. 2009.

[19] Wikipedia: Integration testing [online]. http://en.wikipedia.org/wiki/Integration_testing, Naposledy navštíveno 8. 5. 2009.

[20] Wikipedia: Software testing [online]. http://en.wikipedia.org/wiki/Software_testing, Naposledy navštíveno 8. 5. 2009.

[21] Wikipedia: Verification and validation [online]. http://en.wikipedia.org/wiki/Verification_and_Validation_(software), Naposledy navštíveno 8. 5. 2009.

[22] Williams, L.: White-Box testing. http://agile.csc.ncsu.edu/realsearch, Naposledy navštíveno 14. 5. 2009.

[23] Wireshark: How Wireshark Works. http://www.wireshark.org/docs/wsdg_html_chunked/ChWorksOverview.html, Naposledy navštíveno 14. 5. 2009.

Dodatek A

Příklad testu s využitím rhts-lib

Pro demonstrační účely ukážeme vytvoření tohoto testu:

Test 6	
Popis testu:	Výběr paketů.
Vstupy:	`# editcap -r $TESTPCAP $Tmpfile 1-5 40-45`
Očekávaný výsledek:	Vybere pakety ze zadaných rozmezí.
Aktuální výsledek:	**PASS**

Prvním krokem je import proměnných prostředí rhts-enviroment.sh a knihovny rhtslib, která je bude využívat.

```
. /usr/bin/rhts-environment.sh
. /usr/share/rhts-library/rhtslib.sh
```

Příkaz `rlJournalStart` inicializuje logování testu.

```
PACKAGE=,,wireshark‘‘

rlJournalStart
```

Část Setup je využita pro přípravu souborů potřebných k provedení testu. Vytvoříme dva dočasné soubory a uložíme jméno souboru s testovacími pakety.

```
    rlPhaseStartSetup Setup
        rlAssertRpm $PACKAGE
        rlRun ,,Tmpfile=\‘mktemp\‘‘‘ 0 ,,Creating tmp file‘‘
        rlAssertExists $Tmpfile
        rlRun ,,Tmpfile2=\‘mktemp\‘‘‘ 0 ,,Creating tmp file 2‘‘
        rlAssertExists $Tmpfile2
        TESTPCAP=../testsetpackets.pcap

    rlPhaseEnd
```

V této fázi začíná první test pro výběr více rozsahů paketů.

```
rlPhaseStartTest ,,Packet Selection MultipleRange‘‘
```

Příkaz `rlRun` spustí program se zadanými argumenty a do logu se zaznamená jeho očekávaný a skutečný návratový kód a popis akce, která se prováděla. Ze získaného souboru se načtou časová razítka paketů a testuje se, jestli soubor obsahuje všechny pakety, které má obsahovat a žádné jiné.

```
rlRun ,,editcap -r $TESTPCAP $Tmpfile 1-5 40-45‘‘ 0 ,,Selecting packets‘‘
tethereal -t ad -r $Tmpfile | awk ’{print $3}’ >$Tmpfile2
rlAssert0 ,,Saving timestamp‘‘ ‘echo $?‘
for i in 17:42:51.715978 17:42:51.715999 17:42:51.870585 17:42:51.870632
17:42:52.186750 17:42:56.484625 17:42:56.484903 17:42:56.484939
17:42:56.638460 17:42:56.638526 17:42:56.638772; do
    rlAssertGrep $i $Tmpfile2
done
capinfos $Tmpfile >$Tmpfile2
rlAssertGrep ’^Number of packets: 11[^0-9]*$’ $Tmpfile2
```

Tento test je ukončen příkazem `rlPhaseEnd` a `rlJournalPrintText`vytiskne celý výstup testu na stdout.

```
 rlPhaseEnd
rlJournalPrintText
```

Tento příklad lze nalézt na přiloženém cd jako součást souboru runtest.sh, v adresáři `testy/wireshark/Sanity/editcap/packet-selection`.

Dodatek B

Ukázka protokolu o testu

Toto je ukázka protokolu o testu pro test uvedený v dodatku A. Tento protokol lze nalézt také na přiloženém CD v adresáři `examples`.

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: [   LOG    ] :: TEST PROTOCOL
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [   LOG    ] :: Test run ID   : debugging
:: [   LOG    ] :: Package       : wireshark
:: [   LOG    ] :: Installed:    : wireshark-1.0.6-1.fc10.x86_64
:: [   LOG    ] :: Test started  : 2009-05-11 22:14:03
:: [   LOG    ] :: Test finished : 2009-05-11 22:14:25
:: [   LOG    ] :: Test name     :
:: [   LOG    ] :: Distro:       : Fedora release 10 (Cambridge)
:: [   LOG    ] :: Hostname      : b07-1015b
:: [   LOG    ] :: Architecture  : x86_64

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: [   LOG    ] :: Test description
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

PURPOSE of /CoreOS/wireshark/Sanity/editcap/packet-selection
Description: Suite for test selection funcionality of editcap.
Author: Katerina Novotna <knovotna@redhat.com>

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: [   LOG    ] :: Setup
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [   PASS   ] :: Checking for the presence of wireshark rpm
:: [   PASS   ] :: Creating tmp file
:: [   PASS   ] :: File /tmp/tmp.a5F5LlogGG should exist
:: [   PASS   ] :: Creating tmp file 2
:: [   PASS   ] :: File /tmp/tmp.Dppph25e96 should exist
:: [   LOG    ] :: Duration: 1s
```

```
:: [   LOG    ] :: Assertions: 5 good, 0 bad
:: [   PASS   ] :: RESULT: Setup

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: [   LOG    ] :: Packet Selection MultipleRange
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [   PASS   ] :: Selecting packets
:: [   PASS   ] :: Saving timestamp
:: [   PASS   ] :: File '/tmp/tmp.Dppph25e96' should contain '17:42:51.715978'
:: [   PASS   ] :: File '/tmp/tmp.Dppph25e96' should contain '17:42:51.715999'
:: [   PASS   ] :: File '/tmp/tmp.Dppph25e96' should contain '17:42:51.870585'
:: [   PASS   ] :: File '/tmp/tmp.Dppph25e96' should contain '17:42:51.870632'
:: [   PASS   ] :: File '/tmp/tmp.Dppph25e96' should contain '17:42:52.186750'
:: [   PASS   ] :: File '/tmp/tmp.Dppph25e96' should contain '17:42:56.484625'
:: [   PASS   ] :: File '/tmp/tmp.Dppph25e96' should contain '17:42:56.484903'
:: [   PASS   ] :: File '/tmp/tmp.Dppph25e96' should contain '17:42:56.484939'
:: [   PASS   ] :: File '/tmp/tmp.Dppph25e96' should contain '17:42:56.638460'
:: [   PASS   ] :: File '/tmp/tmp.Dppph25e96' should contain '17:42:56.638526'
:: [   PASS   ] :: File '/tmp/tmp.Dppph25e96' should contain '17:42:56.638772'
:: [   PASS   ] :: File '/tmp/tmp.Dppph25e96' should contain '^Number of packets:
11[^0-9]*$'
:: [   LOG    ] :: Duration: 4s
:: [   LOG    ] :: Assertions: 14 good, 0 bad
:: [   PASS   ] :: RESULT: Packet Selection MultipleRange

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: [   LOG    ] :: Cleanup
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [   PASS   ] :: Removing tmp file
:: [   PASS   ] :: Removing tmp file 2
:: [   LOG    ] :: Duration: 0s
:: [   LOG    ] :: Assertions: 2 good, 0 bad
:: [   PASS   ] :: RESULT: Cleanup
```