VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

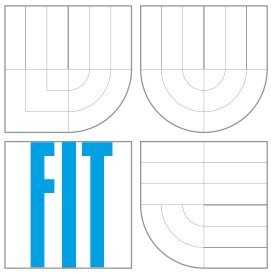
CENTRALIZACE SPRÁVY BEZPEČNOSTNÍCH POLITIK V JAVĚ

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE JAN KALINA
AUTHOR

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

CENTRALIZACE SPRÁVY BEZPEČNOSTNÍCH POLITIK V JAVĚ

JAVA SECURITY POLICY CENTRALIZATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAN KALINA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ZDENĚK LETKO, Ph.D.

BRNO 2014

Zadání bakalářské práce/15505/2013/xkalin03

Vysoké učení technické v Brně - Fakulta informačních technologií

Výzkumné centrum informačních technologií Akademický rok 2013/2014

Zadání bakalářské práce

Řešitel: **Kalina Jan**
Obor: Informační technologie
Téma: **Centralizace správy bezpečnostních politik v Javě**
Centralization of Java Security Policy Management
Kategorie: Informační systémy

Pokyny:

1. Nastudujte principy bezpečnostních politik v Javě a možnosti jejich aplikace.
2. Prostudujte možnosti správy vzdálených Java virtuálních strojů (JVM) nasazených v distribuovaném prostředí Red Hat JBoss.
3. Navrhněte systém pro centralizovanou správu a distribuci bezpečnostních politik v Javě. Systém bude umožňovat sledování aktuálně nasazených politik a distribuovat zvolené politiky na jednotlivé JVM respektive skupiny JVM. Systém bude obsahovat jednoduché webové rozhraní pro komunikaci s uživatelem.
4. Navržený systém implementujte.
5. V konzultantem doporučeném distribuovaném prostředí otestujte schopnosti systému sledovat a aplikovat bezpečnostní politiky na vzdálené JVM.
6. Zhodnoťte dosažené výsledky a diskutujte další možnosti rozšíření implementovaného systému.

Literatura:

- B. Eckel: Thinking in Java (4th Edition), Prentice Hall, 2006.
- S. Oaks: Java Security (2nd Edition), O'Reilly Media, 2001.

Při obhajobě semestrální části projektu je požadováno:

- První tři body zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Letko Zdeněk, Ing., Ph.D.**, VCIT FIT VUT
Konzultant: Škopek Peter, Mgr., RHcz
Datum zadání: 1. listopadu 2013
Datum odevzdání: 21. května 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Výzkumné centrum informačních technologií
Božetěchova 2, 612 66 Brno

prof. Ing. Tomáš Hruška, CSc.
vedoucí ústavu

Abstrakt

WildFly je platformou pro distribuované prostředí splňující specifikaci Java Enterprise Edition. Tato práce se zabývá možnostmi centrální správy bezpečnostních politik v tomto prostředí. Bezpečnostní politika je sada oprávnění, na které virtuální stroje Javy (JVM) omezují možnosti spuštěných aplikací. Právě možnosti používání bezpečnostních politik byly ve WildFly dosud silně omezeny. Výsledkem práce jsou rozšiřující moduly WildFly, doplňující programové rozhraní WildFly a webovou administrační konzolu WildFly o možnost centrálního nasazování bezpečnostních politik na jednotlivé servery domény WildFly bez potřeby jejich restartu. Součástí výsledku je také záplata samotného jádra WildFly, řešící problém výměny bezpečnostní politiky za běhu JVM.

Abstract

WildFly is a platform for distributed environment which meets specification of Java Enterprise Edition. This thesis deals with possibilities of centralized management of security policies in this environment. Security policy is a set of permissions to which the Java virtual machine (JVM) limits possibilities of running applications. Just possibilities of security policy using was in WildFly so far heavily restricted. The result of the thesis are extensions of WildFly which add possibility of central deployment of security policies to individual servers, without need for restart that server, into program interface of WildFly and into WildFly management console. Part of result is also patch of core of WildFly, which solve problem of exchange security policy at runtime of JVM.

Klíčová slova

WildFly, JBoss, Java, bezpečnostní politika, oprávnění, správce bezpečnosti, ochranná doména.

Keywords

WildFly, JBoss, Java, security policy, permission, security manager, protection domain.

Citace

Jan Kalina: Centralizace správy bezpečnostních politik v Javě, bakalářská práce, Brno, FIT VUT v Brně, 2014

Centralizace správy bezpečnostních politik v Javě

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zdeňka Letka, Ph.D. Další informace mi poskytl odborný konzultant Ing. Peter Škopek. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jan Kalina
17. května 2014

Poděkování

Zde bych rád poděkoval svému vedoucímu práce Ing. Zdeňku Letkovi a svému odbornému konzultantovi Ing. Petru Škopkovi za jejich čas, trpělivost, ochotu a cenné rady.

Obsah

Kapitola 1

Úvod

Okolo bezpečnosti serverů poskytujících své služby počítačům v síti bylo již zpracováno mnoho různých prací. Tato práce se však nezabývá problémem, jak zabránit útoku zvenčí útočníkům v síti, ale jak předejít nebezpečí ze strany aplikací nainstalovaných na tomto serveru. Zabývá se tedy tím, jak omezit možnosti aplikací běžících v daném systému a zabránit jim v přístupu ke zdrojům systému, na které nemají nárok.

Tato omezení mohou plnit různé účely – od zabránění vzájemnému přístupu k datům jedné aplikace druhou aplikací jiného provozovatele, v rámci serveru sdíleného mezi více provozovateli aplikací, po zabránění škodám na jiných aplikacích a službách v případě úspěšného prolomení zabezpečení jedné z aplikací běžících na serveru sdíleného více aplikacemi.

Pro podobné účely se obvykle využívají oprávnění na úrovni operačního systému. Ty však, ať už se jedná o jednoduchá unixová oprávnění, podrobnější NTFS (New Technology File System) oprávnění, či ještě podrobnější oprávnění systému SELinux (Security Enhanced Linux), nejsou schopny rozlišit jednotlivé aplikace běžící v rámci jediného procesu. Takovým procesem, v rámci něhož může běžet více aplikací, je například proces aplikačního serveru WildFly.

Tento problém je možné vyřešit na úrovni virtuálního stroje Javy, uvnitř kterého aplikační server běží, za pomoci tzv. správce bezpečnosti (Security Manager) a bezpečnostních politik (Security Policy). Ty umožňují omezovat oprávnění kódu jednotlivých tříd v Javě. Jejich obecné použití je podrobně popsáno v kapitole 2, jejich použitím v rámci aplikačního serveru WildFly se pak zabývá kapitola 3.

V prostředí datových center jsou aplikační servery spojovány do centrálně spravovaných distribuovaných prostředí, takzvaných domén. V tomto prostředí se může jako velmi užitečná projevit možnost centrální správy těchto bezpečnostních politik napříč servery této domény.

Řešení umožňující centrální správu bezpečnostních politik je hlavním tématem této bakalářské práce. Je implementováno jako rozšíření aplikačního serveru WildFly a jeho webové uživatelské rozhraní jako rozšíření webové administrační konzole WildFly. Jeho návrh je nastíněn v kapitole 4, jeho implementace je popsána v kapitole 5 a jeho testování v kapitole 6.

Implementace tohoto řešení byla úspěšná a výsledkem této práce je tedy rozšíření WildFly, umožňující využívat, za běhu aplikačního serveru nastavovat a sledovat takto nasazené bezpečnostní politiky Javy. Je tak přímo uplatnitelné ve stávajících aplikacích aplikačního serveru WildFly, kde je kladen velký důraz na bezpečnost.

Kapitola 2

Bezpečnost v Javě

2.1 Bezpečnost

Tato práce se zabývá bezpečností, což je pojem, který lze v různých souvislostech chápat zcela odlišně. Je nanejvýš vhodné nejprve specifikovat, co termínem bezpečnosti míníme a z jakého pohledu se jí budeme zabývat.

Scott Oaks definuje ve své knize Java Security bezpečnost jako souhrn následujících kritérií: [21]

- **Bezpečí vůči zákeřnému software** – programy by neměly být schopny poškodit prostředí hostitelského počítače.
- **Bezpečí vůči Velkému bratru** – programům by mělo být bráněno ve sledování uživatele – programy by neměly být schopny svévolně číst soukromé informace na počítači, na kterém běží, ani na síti, ke které je tento počítač připojen.
- **Autentizace** – identita autorů programu by měla být ověřována (typicky pomocí digitálního podpisu).
- Šifrování – data odesílaná a přijímaná programem by měla být šifrována.
- Auditovatelnost – potenciálně škodlivé operace by měly být vždy zaznamenávány.
- Specifikovanost – program by měla doprovázet specifikace bezpečnostních pravidel, které program dodržuje.
- Verifikovanost – pro prováděné operace by měla být stanovena pravidla, proti kterým by měly být verifikovány.
- Dbaní na dobré vychování – programům by mělo být bráněno v užívání příliš mnoha systémových prostředků.

V souvislosti s víceuživatelskými prostředími se pak pojem bezpečnosti vyskytuje ještě v další rovině, kdy nejde o bezpečí systému před běžící aplikací, ale o způsob jakým může naopak program ověřit, kdo je jeho uživatelem a zdali má právo po něm žádat vykonání operace, o jejíž vykonání žádá. Touto stránkou bezpečnosti se však tato práce zabývat nebude.

Tato práce se zabývá bezpečností ve smyslu prvních tří bodů uvedeného výčtu (zvýrazněny tučně). Bude se tedy zabývat způsobem ochrany prostředí hostitelského počítače

před jeho poškozením nebo neoprávněným sledováním ze strany běžících programů. Právě oprávnění programu pak souvisí s autentizací. Autentizací se zde myslí ověřování autorství programu. Právě od toho, kdo je autorem programu, se totiž mohou oprávnění programu často odvozovat.

2.2 Java

Programy v jazyce Java bývají překládány do platformě nezávislého a efektivně interpretovatelného mezikódu, takzvaného bajtkódu. Bajtkód (bytecode) bývá zpravidla interpretován virtuálním strojem Javy (JVM - Java Virtual Machine). [9]

JVM je abstraktní výpočetní stroj. Podobně jako reálné výpočetní stroje má svoji instrukční sadu a paměť, se kterou může manipulovat, ale na rozdíl od nich pro něj neexistuje jeho fyzická implementace, pouze emulovaná implementace softwarová. To znamená, že jeho kód není prováděn nativně hardwarem. Je interpretován speciálním programem, interpretem, který je sám zkompilovaný do nativního kódu dané platformy a prováděn jejím hardwarem. To mimo nezávislosti na platformě přináší také vyšší stupeň abstrakce. [9]

Jeho efekt je také bezpečnostní – protože programy v JVM mohou k fyzickým zdrojům počítače (např. k souborům nebo k síti) přistupovat jen skrze JVM, zablokování takového přístupu ze strany JVM není nikterak složité – JVM stačí odmítnout takový požadavek a interpretovaný program nemá možnost JVM obejít.

2.3 Správce bezpečnosti v Javě

Pro maximální nastavitelnost omezení kladených na programy běžící ve virtuálních strojích Javy nerozhoduje o povolení nebo zablokování operace nad zdrojem JVM samotná JVM. Dotazuje se namísto toho speciálního objektu, tzv. správce bezpečnosti. [27]

Správce bezpečnosti je objektem třídy `java.lang.SecurityManager` nebo její podtřídy. Třídu, jejíž objekt JVM použije při svém startu jako objekt správce bezpečnosti, lze nastavit skrze konfigurační proměnnou JVM `java.security.manager`. [6]

Aplikace může referenci na aktuálně používaný objekt správce bezpečnosti získat voláním `System.getSecurityManager()` a vypnout nebo jej vyměnit voláním `System.setSecurityManager()`. Volání těchto metod je samo chráněno správcem bezpečnosti. Nehrozí tedy, že by se program neoprávněně zbavil omezení, která na něj správce bezpečnosti uvalil, jeho vypnutím nebo výměnou. [27]

Hodnotu konfigurační proměnné `java.security.manager` při startu JVM je možné nastavit za pomoci k tomu určeného parametru `-D` příkazu `java`. Je-li hodnota této konfigurační proměnné stanovena na `default` nebo je-li definována bez uvedení hodnoty (jak

```
java -Djava.security.manager HelloWorld
java -Djava.security.manager="" HelloWorld
java -Djava.security.manager=default HelloWorld
java -Djava.security.manager=java.lang.SecurityManager HelloWorld
java -Djava.security.manager=cz.test.TestovaciSM HelloWorld
```

Ukázka kódu 2.1: Příkazy spouštějící program s výchozím správcem bezpečnosti

demonstrují první tři řádky ukázky kódu 2.1), je JVM spuštěna s výchozím správcem bezpečnosti. [6]

Jinou možnou hodnotou této konfigurační proměnné je celý název třídy, jenž má být jako správce bezpečnosti použita. Tento případ demonstrují poslední dva řádky ukázky kódu 2.1. Je tak možné nechat o oprávněních kódu běžícího v JVM rozhodovat vlastní podtřídu výchozího správce bezpečnosti. V rámci inicializace JVM pak bude vytvořena instance třídy určená zmíněnou konfigurační proměnnou. Tato instance je vytvářena voláním `Class.newInstance()`, tj. za pomoci jeho neparametrického konstruktoru.

Jestliže se vytvoření objektu správce bezpečnosti nezdaří, skončí celá inicializace JVM výjimkou `java.lang.InternalError` [7] a ke spuštění uživatelského programu tak vůbec nedojde. Je tedy uplatněn bezpečnostní princip, kdy systém zůstane bezpečný i v případě poruchového stavu, třebaže na úkor funkčnosti.

2.3.1 Implementace vlastního správce bezpečnosti

V této podkapitole je ukázán způsob stanovení bezpečnostní politiky vytvořením vlastního správce bezpečnosti. Jak již bylo uvedeno v kapitole 2.3, správce bezpečnosti je objekt (pod)třídy `java.lang.SecurityManager`, na který JVM, při požadavku na Java API (Application Programming Interface – rozhraní pro programování aplikací), deleguje rozhodnutí o povolení nebo nepovolení operace nad zdrojem.

Jakmile se tedy uživatelská aplikace pokusí provést potenciálně nežádoucí operaci (například se pokusí ukončit JVM voláním `System.exit(0)`), Java API se dotáže správce bezpečnosti na oprávněnost této operace tím, že zavolá jeho k tomu určenou metodu (v tomto případě `checkExit(0)`). [21]

Správce bezpečnosti pak na základě parametrů tohoto volání rozhodne o tom, zda operaci zakáže a vyvolá k tomu určenou výjimku `SecurityException`. Není-li žádná výjimka vyvolána, považuje se operace za povolenou a je provedena. [21]

Bezpečnostní politiku tak můžeme definovat vytvořením vlastní podtřídy třídy `SecurityManager`, kde přepíšeme metody rozhodující o povolení operací, jejichž oprávněnost chceme stanovit.

Od Javy verze Java 2 (SDK v1.2) je možné používat univerzálnější řešení než přepisovat jednotlivé kontrolní metody. Od této verze totiž všechny kontrolní metody (jakou je `checkExit()`) delegují své rozhodnutí na metodu `checkPermission()` správce bezpečnosti. Pro stanovení bezpečnostní politiky tak stačí přepsat tuto jedinou metodu. [30]

Příklad jednoduchého správce bezpečnosti ukazuje ukázka kódu 2.2. Tento jednoduchý správce bezpečnosti implementuje výše zmíněnou metodu `checkPermission()`. Ta je vo-

```
public class TestovaciSM extends SecurityManager {
  @Override
  public void checkPermission(Permission permission) {
    if(permission instanceof FilePermission){
      throw new SecurityException("Prace se soubory je zakazana!");
    }
  }
}
```

Ukázka kódu 2.2: Jednoduchý správce bezpečnosti

lána při každém pokusu o provedení potenciálně nebezpečné operace. Je-li oprávnění, které uživatelský kód vyžaduje, třídy `FilePermission` nebo její podtřídy, je vyvolána výjimka, která provedení operace zabrání.

V rámci vlastní aplikace je možné používání vlastního správce bezpečnosti vyvolat za pomoci dříve zmíněného příkazu `System.setSecurityManager()`, jak ukazuje ukázka kódu 2.3.

```
System.setSecurityManager(new TestovaciSM());
```

Ukázka kódu 2.3: Nastavení vlastního správce bezpečnosti zevnitř JVM

Z vně aplikace pak je možné použití tohoto správce bezpečnosti nastavit při startu JVM obdobně, jak je to popsáno v kapitole 2.3. Příkaz spouštějící aplikaci s tímto správcem bezpečnosti ukazuje ukázka kódu 2.4.

```
java -Djava.security.manager=TestovaciSM HelloWorld
```

Ukázka kódu 2.4: Spuštění aplikace s vlastním správcem bezpečnosti

Vedlejším efektem existence metod volaných při každém pokusu o provedení potenciálně nebezpečné operace je také možnost vedení statistik, jaká oprávnění aplikace nejčastěji využívá. Ty by pak teoreticky mohly být základem pro inteligentního správce bezpečnosti, přidělujícího programu oprávnění na základě heuristické analýzy.

Z těchto statistik můžeme zjistit například to, že i v rámci vykonávání prázdného programu v Javě (třídy, jejíž metoda `main()` neobsahuje žádný příkaz) je prováděno celkem 29 potenciálně nebezpečných operací. V 18 případech je příčinou dotazu čtení obsahu konfiguračních proměnných (Property), v 8 případech jde o manipulaci s vlákny (Thread) a ve zbývajících 3 případech jde o čtení souboru spouštěné třídy (`Program.class`). Ve všech případech jde o operace související se samotnou inicializací JVM.

Pro stanovení bezpečnostní politiky se však tento způsob (vytvoření vlastního správce bezpečnosti) v současnosti již nepoužívá, ačkoli stále možný je. Důvodem je, že vyžaduje od správce serveru, na němž aplikace běží, znalosti programování. Neumožňuje navíc jednoduchým způsobem přidělovat oprávnění v závislosti na třídě, která se potenciálně nebezpečnou operaci snaží provést.

2.4 Zavaděče tříd

Než se dostaneme k v současnosti využívanému způsobu stanovování bezpečnostní politiky, pozastavíme se nad tématem zavádění tříd. Ačkoli se na první pohled může zdát, že toto téma s bezpečností příliš nesouvisí, opak je pravdou. Jen a pouze zavaděč tříd (classloader), který třídu do paměti zavádí, totiž může určit původ třídy, od kterého se oprávnění kódu zpravidla odvozuje.

Zavaděč je objekt odpovědný za načítání tříd a rozhraní v Javě. Na základně binárního názvu třídy (binary name of class) [5], tedy označení třídy používaného v bajtkódu (např. `java.lang.String` nebo `java.security.KeyStore$Builder$FileBuilder$1`), se tuto třídu pokusí vyhledat a načíst do paměti. [22]

Třída každého zavaděče tříd musí být podtřídou třídy `java.lang.ClassLoader` a musí implementovat metodu `findClass()`. Ta je jádrem celého zavaděče – provádí samotné vy-

hledání třídy podle názvu. Jejím výstupem je objekt třídy `java.lang.Class`, představující samotnou třídu. [22]

```
Class classOfUnknown = unknown.getClass();
String nameOfClass = classOfUnknown.getName(); // "java.lang.String"
```

Ukázka kódu 2.5: Získávání názvu třídy neznámého objektu

Při programování běžných aplikací můžeme objekty tříd využít například při snaze získat název třídy neznámého objektu jako řetězec za běhu aplikace, jak ukazuje ukázka kódu 2.5.

První řádek ukazuje získání objektu třídy (objektu třídy `Class`) neznámého objektu. To je možné díky tomu, že každý objekt v Javě si uchovává referenci na svoji třídu. Druhý řádek pak ze získaného objektu třídy získává název této třídy. Objekt třídy dále obsahuje například seznam atributů (`java.lang.reflect.Field`), metod (`java.lang.reflect.Method`) nebo anotací (`java.lang.annotation.Annotation`). [1]

2.4.1 Zdroje kódu a ochranné domény

Zdroj kódu (`java.security.CodeSource`) je souhrnem informací určujících původ třídy: [19]

- `location`: adresa v jednotném formátu (URL – Uniform Resource Locator), ze které byla třída načtena.
- `signers[]` / `certs[]`: Pole podepsaných autorů kódu nebo pole jejich certifikátů. Tyto dvě informace jsou vzájemně odvoditelné. Pro vytvoření objektu zdroje kódu postačuje jedna z nich a druhá je z ní v případě potřeby odvozena.

Ochranná doména (`java.security.ProtectionDomain`) je objektem spojujícím objekt třídy (`Class`) se zdrojem kódu (`CodeSource`) a oprávněními (`Permission`). Konkrétně se skládá z následujících informací: [17]

- `codesource`: Zdroj kódu třídy reprezentovaný výše rozebranou třídou.
- `classloader`: Zavaděč tříd, kterým byla ochranná doména přidělena.
- `principals`: Role uživatele přihlášeného k programu. (viz kapitola 2.5.6)
- `permissions`: Statická oprávnění kódu třídy. (viz kapitola 2.5.6)

Každá třída spadá vždy do jedné ochranné domény, která může být společná více třídám. Za nastavení ochranné domény i zdroje kódu odpovídá zavaděč třídy. Jen ten totiž může původ třídy znát, protože právě on získání jejího kódu a ověření jejího podpisu zajišťuje.

2.4.2 Systémový zavaděč tříd

Systémový zavaděč tříd je zavaděč používaný pro načítání tříd samotného Java API a zavaděčů uživatelských tříd. Je napsán převážně v nativním kódu a je součástí JVM. K načítání tříd tedy využívá nativní metody pro přístup k souborovému systému operačního systému. Třídy načítá ze souborů, jejíž adresu odvozuje z názvu třídy, balíčku a proměnné prostředí `CLASSPATH`. [21]

```
java -classpath /srv/classes my.program.Test
```

Ukázka kódu 2.6: Spuštění JVM s hodnotou proměnné `CLASSPATH` rovnou `/srv/classes`

Je-li tedy JVM spuštěna příkazem uvedeným v ukázce kódu 2.6, bude spuštěna třídní metoda `main()` třídy `Test` ze souboru `/srv/classes/my/program/Test.class`.

Systémový zavaděč je používán k načítání jen nejzákladnějších tříd, zejména tříd Java API a dalších tříd potřebných k provozu jiných zavaděčů tříd. Ostatní třídy z `CLASSPATH` jsou pak načítány jinými zavaděči, typicky zavaděčem tříd z URL, jež bude popsán dále, v kapitole 2.4.3.

O třídách načtených systémovým zavaděčem se často mluví jako o třídách bez zavaděče, protože jejich objekt `Class` má referenci na zavaděč nastavenu na `null`. [21] Zároveň je rovna `null` také ochranná doména takto načtených tříd. Třídy načtené systémovým zavaděčem tak nemají svá oprávnění omezena. [21]

2.4.3 Zavaděč tříd z URL

Zavaděč tříd z URL (`URLClassLoader`) hledá třídy na URL adresách předaných jeho konstruktoru. Je jedním z nejpoužívanějších zavaděčů tříd v Javě, protože je implicitním zavaděčem používaným k načítání tříd z adres určených proměnnou `CLASSPATH`. Dokáže třídy načítat nejen z adresářů, ale i z JAR archivů, je-li jako adresa uvedena adresa JAR archivu. [21]

Zavaděč tříd z URL je, tak jako většina zavaděčů, jedním z bezpečných zavaděčů (`SecureClassLoader`). To znamená, že nastavuje ochrannou doménu třídám, které načítá. Třídy pocházející ze stejného zdroje (s odpovídajícím `CodeSource`) patří do stejné ochranné domény a mají tak stejná statická oprávnění. (viz kapitola 2.5.6) Navrácení vždy stejného objektu ochranné domény přitom zavaděč zajišťuje pomocí asociativního pole `SecureClassLoader.pdcache`, jehož klíčem je objekt zdroje kódu a hodnotou objekt ochranné domény. [18]

2.5 Systém řízení přístupu a soubory bezpečnostní politiky

Systém řízení přístupu (access controller) je mechanismus využívající ochranné domény tříd na zásobníku volání k rozhodnutí, zda operaci požadovanou programem povolit či nikoliv. Na oprávněnost volajícího kódu provést vybranou akci se jej může dotazovat uživatelský kód a veškeré rozhodování na něj zároveň deleguje standardní správce bezpečnosti v Javě verze 2 a novějších. [21]

Systém řízení přístupu tedy, stejně jako správce bezpečnosti, umožňuje omezit, zdali může být operace nad zdrojem provedena nebo ne. Zdroji zde však již nejsou jen zdroje samotné JVM. Uživatelské programy mohou systém řízení přístupu využívat také k omezení přístupu k vlastním zdrojům, které dále poskytují. [21]

Máme-li tedy například knihovnu zprostředkovávající přístup k databázi v souboru po záznamech, při použití oprávnění jen na úrovni Java API by bylo možné programu přidělovat oprávnění jen k celému souboru najednou. Jednotlivé záznamy databáze v souboru je však možné považovat za samostatné zdroje knihovny. Knihovna pak může využít systém řízení přístupu k jejich ochraně.

Systém řízení přístupu je schopný pravidla politiky načítat z objektů bezpečnostní politiky. Implicitní implementace objektu politiky pak pravidla politiky načítá z textového konfiguračního souboru – tzv. souboru bezpečnostní politiky. Vzniká tak mnohem snazší způsob stanovení bezpečnostní politiky – namísto tvorby vlastní třídy správce bezpečnosti stačí pro stanovení politiky poupravit textový soubor bezpečnostní politiky. [21]

Systém řízení přístupu rozhoduje o povolení potenciálně nežádoucí operace na základě oprávnění třídy, jejíž metoda tuto operaci vyvolala i oprávnění tříd, jenž vyvolaly provádění této metody. Množina oprávnění, které prováděný kód v danou chvíli má, tak odpovídá průniku oprávnění všech těchto tříd – tříd na zásobníku volání.

Jestliže tedy metoda `a()` třídy `A` volá metodu `b()` třídy `B` a metoda `b()` se pokusí provést potenciálně nebezpečnou operaci, aby systém řízení přístupu operaci povolil, musí obě třídy, `A` i `B` mít přiděleno potřebné oprávnění.

Na legálnost operace se systému řízení přístupu může dotazovat samotná volaná metoda (voláním metody `AccessController.checkPermission()`), nebo správce bezpečnosti na základě dotazu na legálnost operace od Java API. V obou případech bude operace povolena jen pokud budou potřebným oprávněním disponovat všechny třídy na zásobníku volání.

2.5.1 Implementace a použití systému řízení přístupu

Systém řízení přístupu je implementován třídou `AccessController`. Voláním jeho statických metod na něj standardní správce bezpečnosti deleguje rozhodnutí o povolení provedení potenciálně nežádoucí operace uživatelským programem. Samotnou kontrolu provádí jeho metoda `checkPermission(Permission)`, která v případě neoprávněnosti požadavku na provedení operace vyvolá výjimku `AccessControlException`. Vstupem metody je objekt třídy `Permission`, který reprezentuje oprávnění, jehož vlastnictví je testováno. [21]

Ukázka kódu 2.7 ukazuje způsob, jakým se může kód dotázat systému řízení přístupu, zdali má zadané oprávnění. Dotaz je položen voláním metody `AccessController.checkPermission()`, přičemž je jako parametr připojen objekt testovaného oprávnění. Má-li kód toto oprávnění, provádění pokračuje a na standardní výstup je tak vypsáno, že operace byla povolena. V opačném případě je vyvolána výjimka `AccessControlException`, která je v této ukázce odchycena a na standardní výstup je v takovém případě vypsáno, že operace byla zamítnuta.

```
try {
    // ma tento program opravneni pripojit se na lokalni port 80?
    AccessController.checkPermission(
        new SocketPermission("localhost:80", "connect")
    );
    System.out.println("Pripojeni bylo povoleno");
} catch (AccessControlException e) {
    System.out.println("Pripojeni bylo zamitnuto");
}
```

Ukázka kódu 2.7: Příklad položení dotazu systému řízení přístupu [21]

2.5.2 Třídy oprávnění

Třídy oprávnění představují typ zdroje, ke kterému lze přidělovat oprávnění. Musí být podtřídami třídy `Permission`, ale pro jednodušší implementaci jsou často odvozovány od její podtřídy `BasicPermission`.

Aby bylo možné třídu oprávnění použít, musí být definován její konstruktor, který musí být veřejně přístupný – opatřený specifikátorem přístupu `public`. Jeho parametrům pak musejí odpovídat definice oprávnění v souboru bezpečnostní politiky.

Příklad třídy oprávnění

Ukázka kódu 2.8 ukazuje nejjednodušší způsob vytvoření vlastní třídy oprávnění. Jako příklad je zde použita třída oprávnění z příkladu s knihovnou pro přístup k záznamům v databázovém souboru. (Později bude využita v příkladu v kapitole 2.5.3).

```
public class ZaznamPermission extends java.security.BasicPermission {
    public ZaznamPermission(String name, String actions) {
        super(name, actions);
    }
}
```

Ukázka kódu 2.8: Demonstrační třída oprávnění

2.5.3 Privilegované bloky kódu

Vraťme se nyní k našemu příkladu s knihovnou pro přístup k databázi v souboru. Různým programům chceme přidělovat oprávnění k jednotlivým databázovým záznamům jako k samostatným zdrojům systému, ale zároveň jim nechceme povolit přístup ke zbytku databázového souboru.

Ze způsobu fungování systému řízení přístupu, jak jsme si jej zatím popsali vyplývá, že aby mohla knihovna zprostředkovávající přístup k databázi přistupovat k databázovému souboru, musí mít oprávnění pracovat s tímto souborem jak samotná databázová knihovna, tak i každá třída podílející se na jejím volání.

Tento stav je samozřejmě nepřípustný, protože oprávnění na úrovni uživatelského kódu by tímto ztratila význam. Pokud by uživatelský kód neměl oprávnění přímo přistupovat k databázovému souboru, nemohla by mu přístup k němu zprostředkovat ani tato knihovna. Pokud by naopak uživatelský kód oprávnění k databázovému souboru dostal, mohl by obejít kód knihovny zajišťující kontrolu oprávnění k jednotlivým záznamům a přistupovat tak k datům v celé databázi libovolně bez ohledu na oprávnění k jednotlivým záznamům.

Právě tento problém řeší privilegované bloky kódu. Kód v privilegovaném bloku je spuštěn se samostatným zásobníkem volání. Při ověřování oprávnění při přístupu k jakémukoli zdroji tak ověřování skončí u třídy s tímto blokem. [24] Část kódu knihovny, jež vložíme do privilegovaného bloku, bude tedy spuštěna s oprávněními této knihovny, bez ohledu na oprávnění kódu jež metodu knihovny zavolal.

Příklad použití privilegovaného bloku

```
class DatabazeVSouboru {
 public void provedOperaciNadDatabazi(){
  // Zde bude kod omezen opravnenimi volajici tridy
  String s = AccessController.doPrivileged(new PrivilegedAction<String>()
      {
   public String run(){
    // Zde bude kod spusten nezavisle na opravnenich volajici tridy
    return "privileged";
   }
  }
 }
}
```

Ukázka kódu 2.9: Příklad použití privilegovaného bloku [24]

Ukázka kódu 2.9 ukazuje jednoduchý příklad použití privilegovaného bloku. Metoda `provedOperaciNadDatabazi()` je spuštěna omezená oprávněními kódu, který ji zavolal. Volá metodu `doPrivileged()` systému řízení přístupu, čímž privilegovaně spouští kód metody `run()` objektu, který metodě `doPrivileged()` předal. Kód metody `run()` je tedy spuštěn bez ohledu na oprávnění tříd, jejichž metody volaly metodu `provedOperaciNadDatabazi()`.

Datový typ návratové hodnoty metody `run()` je určen parametrem datového typu `PrivilegedAction`. Tato návratová hodnota je pak vrácena také metodou `doPrivileged()`. Tímto datovým typem však mohou být jen třídy objektů, ne primitivní datové typy. Má-li tedy být výstupem privilegovaně spuštěného kódu primitivní datový typ, je nutné použít obalové třídy, jakou je např. `java.lang.Integer`.

```
class DatabazeVSouboru {
 public Zaznam nactiZaznam(String klic){
  // volajici kod musi mit opravneni k nacteni zaznamu
  AccessController.checkPermission(new ZaznamPermission(klic,"nacteni"));

  return AccessController.doPrivileged(new PrivilegedAction<Zaznam>() {
   public Zaznam run() {
    // provedeni operace nad samotnymi daty (s opravnenimi knihovny)
   }
  });
 }
}
```

Ukázka kódu 2.10: Demonstrační knihovna pro přístup k databázi

Příklad s knihovnou pro přístup k databázi

Ukázka kódu 2.10 ukazuje možnou implementaci knihovny pro přístup k databázi navržené v kapitole 2.5.3. Kód jež nemá oprávnění pro přístup k souboru zde volá metodu `nactiZaznam()`. Tato metoda nejprve (na 4. řádku) voláním `checkPermission()` zkontroluje, zdali má tento kód oprávnění pro přístup k danému záznamu. Pokud by oprávnění neměl, bylo by vykonávání metody již nyní přerušeno výjimkou systému řízení přístupu (`java.security.AccessControlException`). Následně uvnitř privilegovaného bloku (na 8. řádku) provede načtení záznamu z databázového souboru. Protože toto bude provedeno v rámci privilegovaného bloku, nezabrání tomuto čtení absence oprávnění ke čtení databázového souboru kódu, jež volal metodu `nactiZaznam()`. Hodnota vrácená z privilegovaného bloku je nakonec vrácena také metodou `nactiZaznam()` (na 6. řádku).

2.5.4 Soubory bezpečnostní politiky

Tato kapitola popisuje implementaci bezpečnostních politik za pomoci souborů bezpečnostních politik.

Soubor bezpečnostní politiky se skládá z deklarací původů tříd a jim přidělovaných oprávnění. Původ třídy je zde stanoven na základě podpisu JAR archivu, z kterého byla třída získána (direktiva `signedBy`), URL kořene systému balíčků tříd (packages), ze kterého byla třída získána (direktiva `codeBase`) nebo obou těchto kritérií najednou. [21]

Syntaxe souboru bezpečnostní politiky, používané standardní implementací bezpečnostní politiky `PolicyFile`, je ukázána v ukázce kódu 2.11.

```
grant signedBy "autor", codeBase "/koren", principal Trida "jmeno@realm"
    {
    permission TridaOpravneni "parametry","konstruktoru";
    ...
};
```

Ukázka kódu 2.11: Syntaxe souboru bezpečnostní politiky [21][28]

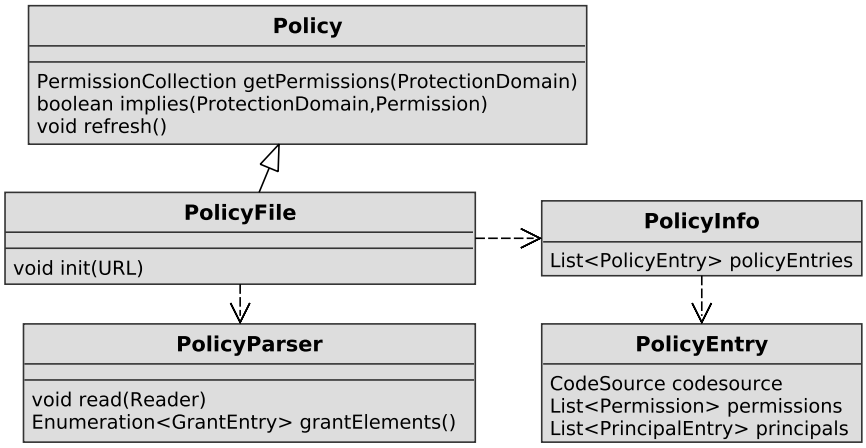
Jak je vidět, oprávnění jsou přidělována kombinacím podepsaného autora (`signedBy`), kořene systému balíčků (`codeBase`) a identifikátoru uživatele nebo role (`principal`). Samotná oprávnění pak jsou definována názvem třídy oprávnění a parametry jeho konstruktoru oddělenými čárkami.

Jestliže třída oprávnění v době načítání souboru bezpečnostní politiky není dostupná, je oprávnění uloženo do dočasného objektu speciální třídy `java.security.UnresolvedPermission`. Stane-li se tato třída později dostupnou, je dočasný objekt nahrazen instancí této třídy při dotazu na toto oprávnění. [29]

Třídy podílející se na bezpečnostní politice postavené na souborech bezpečnostní politiky ukazuje diagram tříd na obrázku 2.1. Tuto politiku realizuje třída `PolicyFile`, podtřída třídy `Policy`, která představuje obecnou bezpečnostní politiku.

Objekt politiky `PolicyFile` provádí rozhodnutí o oprávněnosti operací na základě aktuálního objektu `PolicyInfo`. Tento objekt obsahuje množinu položek bezpečnostní politiky, `PolicyEntry`.

Protože tato množina je v průběhu načítání souborů bezpečnostní politiky tvořena postupně, je umístěna v tomto samostatném objektu. Díky tomu je bezpečnostní politika vyměňována atomicky – jeden objekt `PolicyInfo` je v rámci atomické reference nahrazen



Obrázek 2.1: Třídy podílející se na politice postavené na souborech politiky.

novým. V průběhu výměny bezpečnostní politiky tak nemůže dojít k okamžiku, kdy by byla uplatňována jen z části načtená politika.

K inicializaci a tedy i k načítání politiky ze souborů dochází při vytváření objektu `PolicyFile` a při volání jeho metody `refresh()`. V obou případech jsou soubory politiky určeny konfiguračními proměnnými (property) popsanými v kapitole 2.5.5.

Příklad souboru bezpečnostní politiky

Tento příklad ukazuje jednoduchou bezpečnostní politiku, opět v kontextu příkladu knihovny pro přístup k databázi z kapitoly 2.5.3.

Program zde má přiděleno oprávnění k načtení záznamu „Lucka“. Třída tohoto oprávnění byla definována v kapitole 2.5.2. Nemá ale přiděleno oprávnění k přístupu k samotným databázovým souborům. K těm má oprávnění přistupovat jen knihovna zpřístupňující tuto databázi, popsaná v kapitole 2.5.3. Kvůli způsobu implementace kontroly oprávnění systémem řízení přístupu (popsaným v kapitole 2.5.1) je nezbytné přidělit knihovně oprávnění ke všem záznamům v databázi, jinak by žádná kontrola oprávněnosti požadavku nemohla skončit pozitivně.

```
grant codeBase "file:/srv/program/" {
  permission ZaznamPermission "Lucka", "nacteni";
};
grant codeBase "file:/srv/knihovna/" {
  permission ZaznamPermission "*", "nacteni";
  permission java.io.FilePermission "/srv/knihovna/data/-","read,write";
};
```

Ukázka kódu 2.12: Příklad souboru bezpečnostní politiky

Zástupné znaky

URL kořene systému balíčků může být v souboru bezpečnostní politiky ukončeno zástupnými znaky pomlčka (`-`) a hvězdička (`*`).

Zatímco `/*` odpovídá všem souborům v daném adresáři, `/-` odpovídá všem souborům v daném adresáři i jeho podadresářích, rekurzivně. V obou případech jsou zahrnuty jak soubory tříd (`.class`), tak i celé archivy tříd (`.jar`/`.war`/`.ear`). Archivy vnořené v těchto archivech však již prohledávány nejsou. [23]

2.5.5 Nastavení třídy a souboru bezpečnostní politiky

Tato kapitola popisuje nastavení implicitní třídy a souborů bezpečnostní politiky. Změna těchto nastavení dovoluje správci počítače stanovit bezpečnostní politiku aplikovanou implicitně na všechny programy v Javě běžící na daném počítači.

Používaná třída a soubory bezpečnostní politiky jsou určeny globálně, pro celé běhové prostředí Javy (JRE – Java Runtime Environment), v konfiguračním souboru `java.security` umístěném v podadresáři `lib/security` adresáře JRE. [28]

Třída bezpečnostní politiky je určena svým celým jménem (např. `sun.security.provider.PolicyFile`) v konfigurační proměnné `policy.provider`. Soubory bezpečnostní politiky jsou zde určeny svou absolutní adresou, uvedenou včetně protokolu (pro lokální soubory `file:`), v konfiguračních proměnných `policy.url.n`, kde `n` je pořadové číslo souboru. Načítání souborů bezpečnostní politiky začíná od `policy.url.1` a postupně se `n` inkrementuje a načítají se jednotlivé soubory bezpečnostní politiky, dokud `policy.url.n` existuje. [28]

Příklady nastavení konfiguračních proměnných v tomto souboru jsou uvedeny v ukázce kódu 2.13, která je rozšířením příkladu z knihy pana Oakse. [21]

```
policy.provider=sun.security.provider.PolicyFile
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
policy.url.3=file:/my-policies/my.policy
```

Ukázka kódu 2.13: Význačnější proměnné konfiguračního souboru `java.security`

Nastavenou třídou bezpečnostní politiky je zde třída `PolicyFile` a jako soubory bezpečnostní politiky jsou použity soubory `${java.home}/lib/security/java.policy`, `${user.home}/.java.policy` a `/my-policies/my.policy` v uvedeném pořadí.

Implicitně jsou používány právě první dvě uvedená umístění – `${java.home}/lib/security/java.policy` pro celý systém a `${user.home}/.java.policy` jako jeho rozšíření pro jednotlivé uživatele. Poslední umístění je přidáno pro demonstraci, jak je možné přidat další umístění. [25]

Bezpečnostní politiku je dále možné rozšířit při startu JVM o další soubor bezpečnostní politiky. Stačí nastavit konfigurační proměnnou `java.security.policy` při startu JVM [21], jak je ukázáno v ukázce kódu 2.14.

```
java -Djava.security.policy=dalsi_soubor_politiky.policy ProgramABC
```

Ukázka kódu 2.14: Spuštění JVM s vlastním souborem bezpečnostní politiky

```
java -Djava.security.policy==jediny_soubor_politiky.policy ProgramABC
```

Ukázka kódu 2.15: Spuštění JVM jen s vlastním souborem bezpečnostní politiky

V tomto případě bude uvedený soubor bezpečnostní politiky použit zároveň s výše uvedeným standardním souborem bezpečnostní politiky. Chceme-li standardní soubory bezpečnostní politiky určené souborem `java.security` zcela nahradit, stačí namísto jednoho rovnítka v definici konfigurační proměnné použít rovnítka dvě, jak je ukázáno v ukázce kódu 2.15. [21]

Obsah této proměnné je možné změnit i za běhu programu, má-li daná část programu oprávnění k zápisu do této proměnné. (viz ukázka kódu 2.16)

```
System.setProperty("java.security.policy", "jiny_soubor_politky.policy");
```

Ukázka kódu 2.16: Nastavení souboru bezpečnostní politiky zevnitř JVM

Jak již bylo zmíněno v kapitole 2.5.4, nové načtení politiky ze souborů politiky je možné provést skrze metodu `refresh()` objektu politiky. Změnu politiky tak je možné nechat projevit zavoláním této metody, jak ukazuje ukázka kódu 2.17.

```
Policy.getPolicy().refresh();
```

Ukázka kódu 2.17: Znovunačtení souboru bezpečnostní politiky

Problémem je, že tato metoda je označena jako nedoporučovaná (deprecated), protože je implementačně závislá. [26] Zatímco u třídy bezpečnostní politiky `PolicyFile` správně způsobí projevení změn v souboru bezpečnostní politiky, u jiných implementací bezpečnostní politiky může být implementována jako prázdná operace. Tato práce se proto bude dále zabývat jen bezpečnostními politikami poskytovanými standardní implementací třídy `Policy`, `sun.security.provider.PolicyFile`, a na ní postavenými nebo na ni delegujícími implementacemi.

2.5.6 Statická a dynamická oprávnění ochranných domén

Oprávnění ochranných domén (`ProtectionDomain`) popsaných v kapitole 2.4.1 jsou ukládány přímo v objektech těchto ochranných domén, konkrétně v atributu `permissions`. [17] Protože jsou tato oprávnění do ochranné domény ukládány při jejím vytváření, tedy při zavádění třídy a následně nemohou být změněna, označujeme je jako statická.

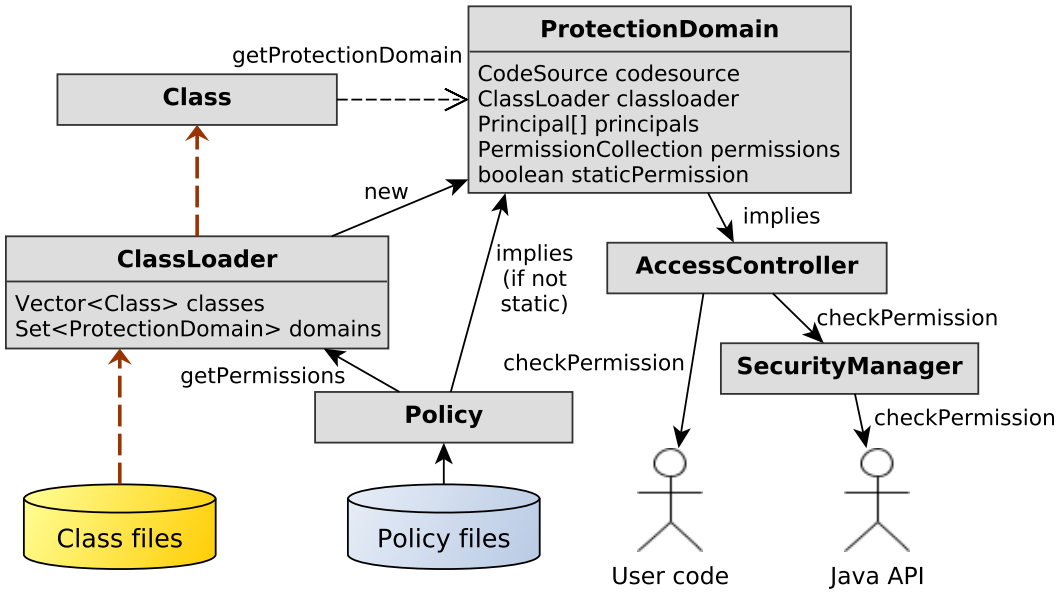
Ochranné domény však souběžně mohou používat ještě oprávnění dynamická. Není-li ochranná doména označena jako používající výhradně statická oprávnění, dotazuje se ochranná doména na oprávnění objektu bezpečnostní politiky při každém volání její metody `implies()`. Takto získaná oprávnění pak označujeme jako dynamická. Jsou-li dynamická oprávnění použita, efektivní oprávnění uplatňovaná metodou `implies()` budou sjednocením dynamických a statických oprávnění třídy. [17]

V současnosti jsou obvykle preferována oprávnění dynamická, jež poskytují vyšší variabilitu tím, že je možné bezpečnostní politiku vyměňovat za běhu programu – změny v ní se projeví, jakmile začne být uplatňována metodou `implies()` objektu bezpečnostní politiky, tedy zpravidla po volání metody `refresh()` používaného objektu bezpečnostní politiky.

Dynamická oprávnění jsou používána na všechny třídy zaváděné zavaděčem tříd z URL (viz kapitola 2.4.3) a jsou tedy využívána ve většině programů v Javě. [11] Umožňují navíc přidělovat oprávnění nejen na základě zdroje kódu, ale také na základě rolí (`principal`) uživatele spouštějícího daný kód. Tato dvě kritéria se však obvykle používají pouze samostatně – buď systém řízení přístupu ověřuje oprávnění kódu provést požadovanou operaci, nebo uživatelský kód ověřuje oprávnění uživatele, který jej spouští, k provedení požadované operace. V obou případech se však stále jedná o dynamická oprávnění.

Statická oprávnění byla v Javě donedávna implementována jen v zájmu zachování kompatibility s verzemi Javy staršími než Java 2 SE 1.4. [17] V současnosti se však statická oprávnění opět začaly používat a to za účelem optimalizace. Získáváním oprávnění třídy z objektu bezpečnostní politiky pouze při zavádění třídy a ne při každém požadavku na ověření oprávněnosti k provedení operace je totiž možné proces ověřování oprávněnosti urychlit.

2.6 Celkový pohled na bezpečnost v Javě



Obrázek 2.2: Diagram datových toků tříd (přerušované tlusté šipky) a oprávnění tříd (souvislé šipky)

Diagram datových toků na obrázku 2.2 ukazuje putování informace o oprávněních třídy ze souborů bezpečnostní politiky (Policy files) až do systému řízení přístupu (`AccessController`) a správce bezpečnosti (`SecurityManager`). Souvislé šipky představují přenos informace o oprávněních, zatímco přerušované tlusté šipky přenos informace o třídách. Tenké přerušované šipky představují reference. Směr šipky pak vede z třídy objektu s referenční proměnnou do třídy referencovaného objektu.

Ochranné domény (`ProtectionDomain`) a tím i statická oprávnění třídy jsou třídám přidělovány při jejich zavádění. Objekty ochranných domén (`ProtectionDomain`) vytváří zavaděč tříd a to na základě jejich zdroje kódu (`CodeSource`). Zavaděč tříd (`ClassLoader`) tak informace o oprávněních předává objektu ochranné domény (`ProtectionDomain`) při jeho vytváření. Na statická oprávnění, která má zdroji kódu (a tedy i vytvořené ochranné

doméně) zavaděč přidělit, se zavaděč dotazuje aktuálně používaného objektu bezpečnostní politiky (`Policy`).

Primárním výstupem zavaděče tříd je ale objekt třídy (`java.lang.Class`). Až ten nese referenci na takto vytvořenou ochrannou doménu, kterou má společnou se všemi třídami se stejným zdrojem kódu. Zavádění tříd podrobněji viz kapitola 2.4.

Systém řízení přístupu (`java.security.AccessController`) pak při dotazu `checkPermission()` ze strany uživatele nebo ze strany správce bezpečnosti rozhoduje o povolení operace na základě odpovědí na jeho dotaz `implies()` položený ochranným doménám tříd na zásobníku volání. Informace o oprávněních tak putují z objektu ochranné domény do systému řízení přístupu, jak je znázorněno na obrázku.

Metoda `implies()` ochranné domény (`ProtectionDomain`) rozhoduje v závislosti na tom, zda je nastavena jako používající výhradně statická oprávnění. Pokud ano, rozhoduje jen na základě statických oprávnění uložených do ochranné domény při jejím vytváření. V opačném případě se zároveň dotazuje také objektu bezpečnostní politiky (`Policy`). Cesta informace o oprávněních z objektu bezpečnostních politiky do ochranné domény je také znázorněna na obrázku.

Na systém řízení přístupu pak dotaz `checkPermission()` deleguje také standardní správce bezpečnosti (`java.lang.SecurityManager`), což je znázorněno na obrázku jako putování informace ze systému řízení přístupu do objektu správce bezpečnosti.

Kapitola 3

Distribuované prostředí WildFly

Tato kapitola popisuje prostředí aplikačního serveru WildFly, dříve známého jako JBoss Application Server. Podrobněji pak rozebírá možnosti používání bezpečnostních politik Javy v prostředí tohoto aplikačního serveru.

3.1 Úvod do WildFly

WildFly, dříve známý jako JBoss Application Server, je aplikační server standardu Java EE (Enterprise Edition). Poskytuje běhové prostředí (nejen) Java EE aplikacím, které jsou na tomto aplikačním serveru nainstalovány. [20]

Jednotlivé instalace WildFly nazýváme hostiteli (host). Hostitel typicky odpovídá jednomu fyzickému serveru. V rámci každého hostitele může běžet více serverů (server). Hostitelé mohou být spojováni do domén (domain), přičemž jeden z nich v takovém případě zastává roli tzv. doménového řadiče (domain controller). Ostatní hostitelé domény se k němu pak registrují a je na ně aplikována doménová konfigurace. [13]

Doménová konfigurace je uložena v konfigurační souboru domény, `domain.xml`, uložené právě na doménovém řadiči. Tento soubor obsahuje konfiguraci skupin serverů a profilů serverů. Konfigurace profilů pak zahrnuje také konfiguraci subsystémů používaných na serverech domény, které používají daný profil. [13]

Každý hostitel má zároveň vlastní konfigurační soubor `host.xml`. Tento soubor určuje název hostitele, zda-li je tento hostitel doménovým řadičem a, pokud není, také adresu doménového řadiče. Zde jsou definovány servery, které na daném hostiteli poběží, porty, na kterých budou poskytovat své služby, a skupiny do kterých budou patřit. [13]

Distribuovanosti prostředí je docíleno nasazováním aplikací (deployment) na celé skupiny serverů najednou. Požadavky směřované na aplikace pak jsou rozprostírány mezi všechny servery domény osazené danou aplikací pomocí tzv. vyvažovačů zátěže (load balancer). [10]

Vyvažovač zátěže umožňuje skrýt celou doménu serverů za jedinou IP adresu. Na této adrese pak naslouchá požadavkům a předává je jednotlivým serverům této domény – zpravidla tomu, který je v daný okamžik nejméně vytížený. [10]

3.2 Rozhraní pro správu WildFly

Tato podkapitola popisuje rozhraní umožňující správu aplikačních serverů WildFly, **JBoss management API**. Toto rozhraní umožňuje spravovat doménu WildFly a provádět změny

v konfiguraci domény i jednotlivých serverů. [31]

Rozhraní zpřístupňuje konfiguraci ve formě stromové struktury, skládající se z konfiguračních uzlů a konfiguračních vlastností. Zatímco konfigurační uzly se mohou libovolně zanořovat, vlastnosti mohou být jen listy této stromové struktury. [31]

Pro reprezentaci dotazů nad touto stromovou strukturou i odpovídajících odpovědí je využívána dynamická reprezentace modelu (DMR – Dynamic Model Representation). DMR všechny informace (např. dotaz nebo odpověď Management API) ukládá jako strom uzlů – objektů třídy `ModelNode`. [31]

Na rozdíl od uzlů konfiguračního stromu může být obsahem uzlu `ModelNode` hodnota jednoho z definovaných datových typů (viz [8]) nebo seznam jiných uzlů. Ne však obojí zároveň. [8]

Klienti JBoss Management API mohou aplikačnímu serveru pokládat dotazy ve formě uzlů DMR a žádat tak o provedení operace nad konfiguračním stromem. Výsledek této operace je pak opět ve formě uzlu DMR předán dotazujícímu se klientovi. [31]

Operace mohou sloužit jak ke čtení dat z konfiguračního stromu a k jejich zápisu, tak i k provádějí jiných akcí souvisejících se správou serverů nebo domény. Příklady posledního uvedeného typu operace mohou být operace startu (`start`) nebo znovunačtení (`reload`) serveru. Každý uzel konfiguračního stromu může mít jinou množinu operací, které nad ním lze provádět. Na tuto množinu operací je možné se serveru dotázat. [31]

3.2.1 Rozšíření WildFly

Funkce poskytované aplikačním serverem aplikacím i jeho správcům je možné různými způsoby rozšiřovat za pomoci rozšíření. Typickým obsahem rozšíření je subsystém. Subsystémy jsou konfigurovány pro jednotlivé konfigurační profily, tedy pro všechny servery používající daný profil.

Pro potřeby této práce je důležité přijmout hlavně to, že rozšíření může vytvářet vlastní typy uzlů konfiguračního stromu. Těmto uzlům může definovat vlastnosti, které mohou mít. Uzlům i vlastnostem pak lze přidělovat třídy ošetřující operace nad nimi. [32]

Je tak například možné definovat typ uzlu, který při svém vytvoření nebo smazání spustí speciální metodu pro tento účel vytvořeného objektu. Metody podobných objektů mohou být spouštěny také při nastavení hodnoty nebo nastavení nedefinované hodnoty konfigurační vlastnosti tohoto uzlu.

V podstatě nic co běží v rámci WildFly, ať již součásti WildFly nebo uživatelské aplikace, neběží celou dobu běhu aplikačního serveru. Pouze si zaregistrují události, na které později, až se vyskytnou, reagují – říkáme, že vše co běží v rámci WildFly je služba. [8]

Běžnějším příkladem registrované události, než jakou je úprava konfiguračního uzlu, je příchod HTTP požadavku se specifickou URL. Takové události jsou typicky registrovány Servlety v rámci své definice v konfiguračním souboru `web.xml` své aplikace. [8]

3.2.2 Webová konzola WildFly

Webová konzola WildFly je standardní součástí aplikačního serveru WildFly. Je GWT (Google Web Toolkit) aplikací běžící na straně klienta, ve webovém prohlížeči. Aplikační server tato aplikace spravuje prostřednictvím JBoss Management API. Konkrétně využívá jeho variantu běžící nad protokolem HTTP, která dotazy na API zasílá ve formě HTTP POST požadavků, jejichž tělem je JSON reprezentace uzlu DMR, `ModelNode`. Obdobným způsobem pak server zasílá odpověď. [3]

3.3 Běh WildFly se správcem bezpečnosti

Tato podkapitola se zabývá problémem používání správce bezpečnosti a bezpečnostních politik v prostředí WildFly.

Na první pohled se může zdát, že nakonfigurovat použití správce bezpečnosti na aplikační server WildFly je možné stejně jako na jakoukoli jinou aplikaci v Javě – nastavením patřičných konfiguračních vlastností (popsaných v kapitolách 2.3 a 2.5.5) při startu JVM za pomoci k tomu určeného parametru `-D` příkazu `java`.

Ve spouštěcím skriptu WildFly (`standalone.sh` / `domain.sh` / `standalone.bat` / `domain.bat`) je pro potřeby přidávání vlastních parametrů příkazu `java` používána proměnná `JAVA_OPTS`. Připojením potřebných parametrů na konec hodnoty této proměnné bychom tedy měli dosáhnout použití správce bezpečnosti a stanovené bezpečnostní politiky ve všech JVM aplikačního serveru. Kód který by k tomu do spouštěcího skriptu musel být přidán ukazuje (pro `*.sh` variantu) ukázka kódu 3.1. Každý její řádek přidá na konec proměnné `$JAVA_OPTS` jeden parametr příkazu `java` potřebný ke spouštění serveru s bezpečnostní politikou. [2]

```
JAVA_OPTS="$JAVA_OPTS -Djava.security.manager"
JAVA_OPTS="$JAVA_OPTS -Djava.security.policy==wildfly.policy"
```

Ukázka kódu 3.1: Doplnění spouštěcího skriptu o použití správce bezpečnosti

Uvedený soubor bezpečnostní politiky (`wildfly.policy`) prozatím ponecháme prázdný. Zkusíme-li nyní aplikační server spustit, spuštění skončí neúspěchem kvůli chybějícím oprávněním, nezbytných k běhu aplikačního serveru. Tento výsledek je ale pro nás pozitivní, neboť prokazuje, že aplikační server byl bezpečnostní politikou skutečně omezen.

Do souboru bezpečnostní politiky dále přidáme přidělení všech oprávnění veškerému kódu běžícímu v rámci aplikačního serveru. Vzniklý soubor bezpečnostní politiky ukazuje ukázka kódu 3.2.

```
grant {
    permission java.security.AllPermission;
};
```

Ukázka kódu 3.2: První testovací soubor bezpečnostní politiky

Zopakujeme-li experiment, zjistíme že s takto nastavenou bezpečnostní politikou se aplikační server bez problémů spustil a bez problému funguje. Podrobnějším zkoumáním však zjistíme, že na uživatelské aplikace se tato benevolentní bezpečnostní politika neuplatňuje. Zatímco zapnutí správce bezpečnosti tedy možnosti uživatelských aplikací skutečně omezilo, nastavení vše povolujících bezpečnostních politik uživatelským aplikacím žádné z odebraných schopností nenavrátilo.

3.3.1 Zavaděč tříd modulů WildFly

Tato podkapitola popisuje jádro problému s bezpečnostními politikami ve WildFly – zavaděč tříd modulů WildFly (`org.jboss.modules.ModuleClassLoader`).

Zavaděč tříd se podařilo jako viníka odhalit postupným krokováním programu. To potvrdilo, že objekty bezpečnostní politiky produkují správná oprávnění. Zároveň ale ukázalo,

že do ochranných domén tříd jsou ukládány oprávnění jiná. Jak vyplývá z popisu cesty oprávnění ze souborů bezpečnostních politik do ochranných domén v kapitole 2.6, jediným mezičlánkem, který mohl s oprávněními v této době manipulovat, je právě zavaděč tříd.

Tento předpoklad pohled do zdrojového kódu zavaděče tříd modulů potvrdil – tento zavaděč oprávnění poskytovaná objektem bezpečnostní politiky implicitně ignoruje. Namísto toho třídám do ochranných domén ukládá oprávnění určená specifikací Java EE. [12] (Ta jsou v případě uživatelských aplikací upravitelná souborem `permission.xml`.)

Zároveň však byla nalezena konfigurační proměnná (property), která toto chování umožňuje změnit. Bude-li konfigurační proměnná `jboss.modules.policy-permissions` nastavena na hodnotu `true`, bude zavaděč tříd modulů do ochranných domén ukládat jak oprávnění určená specifikací Java EE, tak i oprávnění z objektů bezpečnostní politiky. Množina efektivních oprávnění třídy pak bude odpovídat sjednocení těchto dvou množin oprávnění. [4]

Ukázka kódu 3.3 ukazuje všechny definice konfiguračních proměnných, které musí být do spouštěcího skriptu aplikačního serveru WildFly přidány, aby byl server a aplikace které v něm běží omezeny oprávněními stanovenými uvedeným souborem bezpečnostní politiky (v příkladu `directory/wildfly.policy`).

```
JAVA_OPTS="$JAVA_OPTS -Djboss.modules.policy-permissions=true"
JAVA_OPTS="$JAVA_OPTS -Djava.security.manager"
JAVA_OPTS="$JAVA_OPTS -Djava.security.policy==directory/wildfly.policy"
```

Ukázka kódu 3.3: Úprava spouštěcího skriptu pro spuštění s bezpečnostní politikou

3.3.2 Úpravy zavaděče tříd modulů WildFly

Tato podkapitola popisuje úpravy zavaděče tříd modulů WildFly, které bylo nutné provést, aby bylo možné vyměňovat bezpečnostní politiky za běhu aplikačního serveru.

Jakékoli snahy o výměnu bezpečnostní politiky za běhu aplikačního serveru se doposud jevily jako marné. Oprávnění totiž byly nastavována jako statická. Jsou tedy z objektu bezpečnostní politiky načítána pouze při načítání třídy. Změny bezpečnostní politiky se tak projevovaly pouze na nově načítaných třídách.

Protože je cílem výměna bezpečnostní politiky za běhu aplikačního serveru, je tento stav nedostačující. Aby byla možná změna bezpečnostní politiky za běhu aplikačního serveru, je nezbytné, aby byla oprávnění používána jako dynamická, jak je popsáno v kapitole 2.5.6. V rámci této práce byla tedy vytvořena záplata zavaděče tříd modulů WildFly, která umožňuje používání dynamických oprávnění ve WildFly. Ukázána je v ukázce kódu 3.4.

Zatímco řešení popsané v předchozí kapitole ukládalo do ochranných domén množinu oprávnění, která byla sjednocením oprávnění udělených dle specifikace Java EE a oprávnění z objektu bezpečnostní politiky, v této variantě jsou do ochranné domény ukládány opět jen oprávnění udělená dle specifikace Java EE. Při vytváření této ochranné domény je ale použit konstruktor se všemi čtyřmi parametry, čímž je tato doména označena jako používající dynamická oprávnění. Při každém dotazu na oprávnění k provedení potenciálně nežádoucí operace tak je tento dotaz předán objektu bezpečnostní politiky. Efektivní oprávnění třídy pak jsou sjednocením dynamických oprávnění z objektu bezpečnostní politiky a statických oprávnění přidělených dle specifikace Java EE při načítání třídy.

```
+  if (POLICY_REFRESHABLE) {
+     protectionDomain = new ProtectionDomain(codeSource, permissions,
+                                this, null); // staticPermission=false
+  } else if (POLICY_PERMISSIONS && POLICY_READY.get()) {
-  if (POLICY_PERMISSIONS && POLICY_READY.get()) {
```

Ukázka kódu 3.4: Hlavní část záplaty umožňující nastavit používání dynamických oprávnění

Použití této nové varianty bylo přitom záměrně omezeno, aby bylo možné začlenění této úpravy do vývojové větve WildFly. Byla vytvořena nová konfigurační vlastnost `jboss.modules.policy-refreshable`, jejíž hodnota určuje, zda-li mají být dynamická oprávnění použita. Je-li její hodnota `true`, budou použita dynamická oprávnění, jinak budou používána statická oprávnění jako doposud. O začlenění této záplaty do vývojové větve bylo požádáno [14].

3.3.3 Výměna bezpečnostní politiky za běhu

Tato podkapitola popisuje postup, který je nutné použít, aby bylo možné z aplikace nebo z rozšíření WildFly provést výměnu bezpečnostní politiky.

Jak již bylo popsáno v kapitole 2.5.5, pro výměnu politiky je nutné pozměnit hodnotu konfigurační proměnné `java.security.policy` a vyžádat od objektu politiky znovunačtení souboru bezpečnostní politiky. V prostředí WildFly však, přestože již je zajištěno používání dynamických oprávnění (viz kapitola 3.3.2), bohužel nestačí zavolat metodu `refresh()` aktuálně používaného objektu bezpečnostní politiky. Tímto objektem je totiž instance třídy `org.jboss.security.jacc.DelegatingPolicy`. Tento objekt sice všechny volání metody `implies()` nesouvisející s JACC (Java Authorization Contract for Containers) deleguje na původní objekt bezpečnostní politiky, metodu `refresh()` na něj však nedeleguje.

Toto chování jsem nahlásil jako chybu implementace třídy `DelegatingPolicy`. Deleguje-li třída dotazování, měla by dle mého přesvědčení delegovat také žádost o znovunačtení dat, na základě kterých jsou tato rozhodnutí prováděna. [15]

Pro potřeby této bakalářské práce však bylo znovunačtení bezpečnostní politiky prozatím zajištěno za pomoci reflexe, jak je ukázáno v ukázce kódu 3.5. Reflexí je zde získána hodnota atributu `delegate` aktuálně používaného objektu bezpečnostní politiky třídy `DelegatingPolicy`. Hodnotou `delegate` je další objekt bezpečnostní politiky a zavolání jeho metody `refresh()` již postačuje k dosažení znovunačtení bezpečnostní politiky ve WildFly.

```
Class<?> delegatingPolicy = Class.forName("org.jboss.security.jacc.
    DelegatingPolicy");
Field f = delegatingPolicy.getDeclaredField("delegate");
f.setAccessible(true);
Policy in = (Policy) f.get(Policy.getPolicy());
in.refresh();
```

Ukázka kódu 3.5: Znovunačtení bezpečnostní politiky při nastavené `DelegatingPolicy`

Kapitola 4

Návrh systému

V této kapitole je navržen způsob implementace systému pro centralizovanou správu a distribuci bezpečnostních politik.

4.1 Specifikace problému

Tato práce řeší problém, jakým způsobem používat bezpečnostní politiky Javy v distribuovaných prostředích aplikačních serverů WildFly. Tato prostředí jsou typicky tvořena skupinou několika (nebo i mnoha) aplikační serverů seskupených do domény WildFly.

Pravděpodobně nejprimitivnějším možným řešením, jak bezpečnostní politiky na tyto domény serverů nasazovat, je použít stejné řešení, jaké bylo v kapitole 3.3 použito pro nasazení politiky na samostatný server. Tento způsob je ale poměrně pracný, neboť pro každou změnu bezpečnostní politiky vyžaduje zkopírování tohoto souboru na jednotlivé servery nebo úpravu jeho adresy ve spouštěcím skriptu WildFly. Následně je navíc nutné server WildFly restartovat.

Cílem této práce proto je, implementovat řešení pro správu bezpečnostních politik, splňující následující požadavky:

- Mělo by umožnit nastavovat použití bezpečnostní politiky na serverech domény.
- Mělo by umožňovat provádět toto nastavení **centrálně** – z jediného místa.
- Nemělo by vyžadovat ruční **přenos souborů** bezpečnostní politiky mezi servery, na kterých mají být použity.
- Nemělo by vyžadovat **restart** serverů domény pro projevení změn v nastavení politiky.

Návrhy řešení uvedených požadavků zpracovávají následující kapitoly.

4.2 Možnosti řešení

Možnosti řešení problému popsaného v kapitole 4.1 jsou následující:

- Používat jako soubor bezpečnostní politiky soubor uložený v jednom společném umístění.
- Zajistit přenos souborů bezpečnostní politiky samostatným programem nebo rozšířením WildFly.

Obě varianty jsou podrobněji popsány v následujících podkapitolách.

4.2.1 Společné umístění souboru bezpečnostní politiky

Poměrně jednoduchým řešením problému by mohlo být nastavit na serverech domény ručně soubor bezpečnostní politiky, ležící v umístění společném pro všechny servery domény. Pro jakoukoli změnu bezpečnostní politiky by pak stačilo upravit tento jeden společný soubor bezpečnostní politiky.

Sdíleným umístěním by mohl být adresář sdílený například prostřednictvím protokolu FTP (File Transfer Protocol), NFS (Network File System) nebo HTTP (Hypertext Transfer Protocol). Toto umístění by bylo příhodné umístit na doménový řadič, který zajišťuje šíření i ostatní konfigurace mezi servery domény.

Ačkoli toto řešení poměrně elegantně řeší problém změny obsahu souboru bezpečnostní politiky, problematičtější už je používání více souborů bezpečnostní politiky na různých serverech a přepínání mezi nimi. Tento problém je možné vyřešit za pomoci symbolických odkazů – soubor používaný JVM by byl symbolickým odkazem daného serveru. Namísto výměny adresy souboru bezpečnostní politiky na straně serveru by tak byl vyměňován cíl odkazu tohoto symbolického odkazu.

Problémem, který přetrvává, je ale způsob, jak zajistit znovunačtení souboru bezpečnostní politiky a zajistit tak nové načtení bezpečnostní politiky. To může zajistit jen aplikace běžící v rámci JVM jednotlivých serverů nebo restart celého aplikačního serveru.

Chceme-li tedy zajistit výměnu bezpečnostní politiky bez potřeby restartu aplikačního serveru, je nutné vytvořit program, který bude nasazen na všech serverech domény a bude naslouchat požadavkům na znovunačtení bezpečnostní politiky. Tento program je možné implementovat ve formě Java EE aplikace nebo ve formě rozšíření aplikačního serveru WildFly. V tomto případě však ke druhé, složitější, variantě není žádný racionální důvod.

4.2.2 Přenos souborů bezpečnostní politiky samostatným programem

V předchozí variantě byl program nasazený na serverech domény využíván jen ke znovunačítání bezpečnostní politiky. Tato varianta počítá s jeho využitím také k nastavení a přenosu samotného souboru bezpečnostní politiky. Soubor bezpečnostní politiky by přitom stačilo na server, na který má být nasazen, uložit až těsně před tím, než by měl být nastaven. K jeho uložení by přitom postačoval dočasný soubor, který by následně, po vyvolání znovunačtení bezpečnostní politiky, mohl být bezpečně smazán. Soubor bezpečnostní politiky totiž musí na serveru existovat jen po dobu, kdy je jeho obsah načítán, k čemuž v tomto případě bude docházet jen v souvislosti se znovunačítáním bezpečnostní politiky.

Byla-li by tato aplikace implementována jako rozšíření aplikačního serveru WildFly, jakým je například subsystém, mohla by k ukládání a distribuci souborů bezpečnostních politik využívat konfigurační strom WildFly. Nebylo by tak třeba žádného speciálního úložiště souborů bezpečnostní politiky – soubory bezpečnostní politiky by tak byly uloženy společně s konfigurací většiny součástí WildFly v konfiguračním stromu domény. (O konfiguračním stromu WildFly více v kapitole 3.1.)

Tím, že by soubory bezpečnostní politiky byly uloženy v konfiguračním stromě WildFly by navíc bylo možné provádět znovunačítání bezpečnostní politiky v reakci na událost změny používané bezpečnostní politiky. Na změnu hodnoty v konfiguračním stromu WildFly je totiž při její definici možné navázat akci. (viz kapitola 3.2.1)

4.3 Výběr řešení

Jako lepší řešení bylo zvoleno druhé uvedené řešení – řešení ve formě programu provádějícího jak znovunačítání, tak také nastavování a distribuci souborů bezpečnostní politiky. Zvoleno bylo kvůli menším nárokům na instalaci a správu takového řešení. Zatímco první řešení by vyžadovalo jak přípravu sdíleného umístění souborů politiky, tak i instalaci programu umožňujícího provést znovunačtení politiky, zvolené řešení vystačí jen s instalací implementovaného programu. Případné nižší nároky na implementaci tohoto programu nemohou v tomto případě omluvit vyšší nároky na správu výsledného řešení.

Jako vhodný způsob implementace tohoto programu, zajišťujícího distribuci a znovunačítání souborů bezpečnostní politiky, byla dále zvolena varianta ve formě rozšíření aplikačního serveru WildFly. Výhody spojené s využíváním konfiguračního stromu WildFly pro ukládání souborů bezpečnostní politiky převyšují zvýšené nároky na implementaci programu ve formě rozšíření WildFly namísto implementace ve formě Java EE aplikace.

4.4 Návrh uživatelského rozhraní

Součástí zadání práce bylo také vytvoření webového rozhraní pro komunikaci implementovaného systému s uživatelem. Toto webové rozhraní by mělo umožňovat následující:

- Sledovat bezpečnostní politiky nasazené na jednotlivých serverech domény.
- Upravovat bezpečnostní politiky uložené v konfiguračním stromu WildFly.
- Nasazovat tyto bezpečnostní politiky na jednotlivé servery domény.
- Nasazovat tyto bezpečnostní politiky na celé skupiny serverů domény.
- Vypínat používání bezpečnostní politiky na serverech nebo jejich skupinách.

Jelikož je samotné jádro řešení implementováno jako subsystém WildFly, jako logické řešení se nabízí implementovat uživatelské rozhraní tohoto subsystému jako rozšíření webové konzoly WildFly. Toto rozhraní bylo navrženo jako tři nové konfigurační obrazovky webové konzoly WildFly:

- Obrazovka **Servers** (Servery) umožňuje zvolit, které bezpečnostní politiky mají být nasazeny na jednotlivých serverech a skupinách domény. Nastavením prázdné (None/`null`) bezpečnostní politiky navíc umožňuje používání bezpečnostní politiky na serveru nebo skupině vypnout.
- Obrazovka **Policies** (Politiky) umožňuje vytvářet, upravovat a odstraňovat samotné bezpečnostní politiky. Ty pak mohou být jednotlivým serverům přidělovány na obrazovce **Servers**.
- Obrazovka **JSM Policy** (Politika správce bezpečnosti Javy) umožňuje sledovat politiky nasazené na jednotlivých serverech – jejich název a obsah.

Grafický návrh první uvedené konfigurační obrazovky, umožňující zvolit, které bezpečnostní politiky mají být nasazeny na jednotlivých serverech a skupinách, je vyobrazen na obrázku 4.1. Zároveň je pro srovnání na obrázku 4.2 ukázán snímek obrazovky ukazující výslednou podobu této konfigurační obrazovky, jak byla později implementována.

Jako způsob výběru bezpečnostní politiky serveru i skupiny byla zvolena omezená nabídka. Tato nabídka umožňuje výběr jedné bezpečnostní politiky ze seznamu politik vytvořených prostřednictvím druhé konfigurační obrazovky. Umožňuje zároveň vypnout používání bezpečnostní politiky výběrem speciální hodnoty **none** (žádná). Další speciální hodnota **ambiguous** (neurčitá) má signalizovat, že servery skupiny používají různé bezpečnostní politiky.

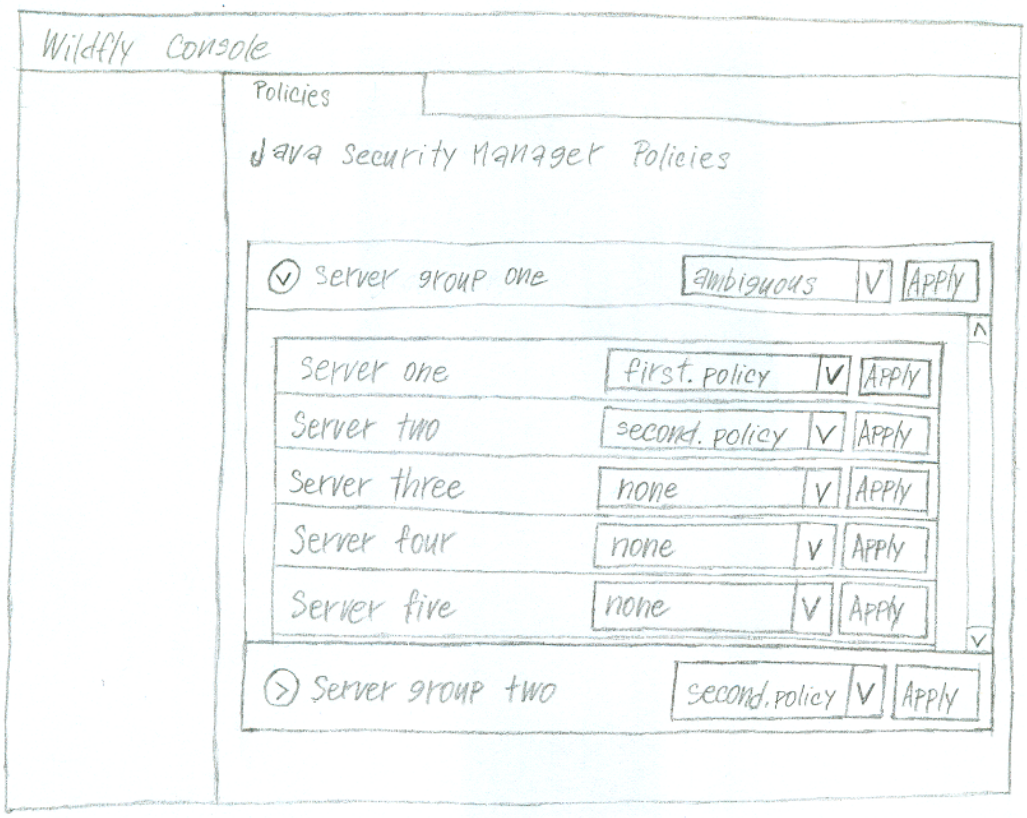
Celou konfigurační obrazovku mimo nadpisu tvoří seznam skupin v doméně. Každou položku tohoto seznamu přitom tvoří ikona pro zobrazení seznamu serverů obsažených v této skupině, název skupiny, výše popsaná nabídka pro výběr politiky pro celou skupinu najednou a tlačítko pro uložení změny výběru.

Použitím ikony je pod položkou skupiny zobrazen seznam serverů této skupiny podobný seznamu skupin. Jeho položky jsou tvořeny názvem serveru, nabídkou pro výběr politiky a tlačítkem pro uložení změny.

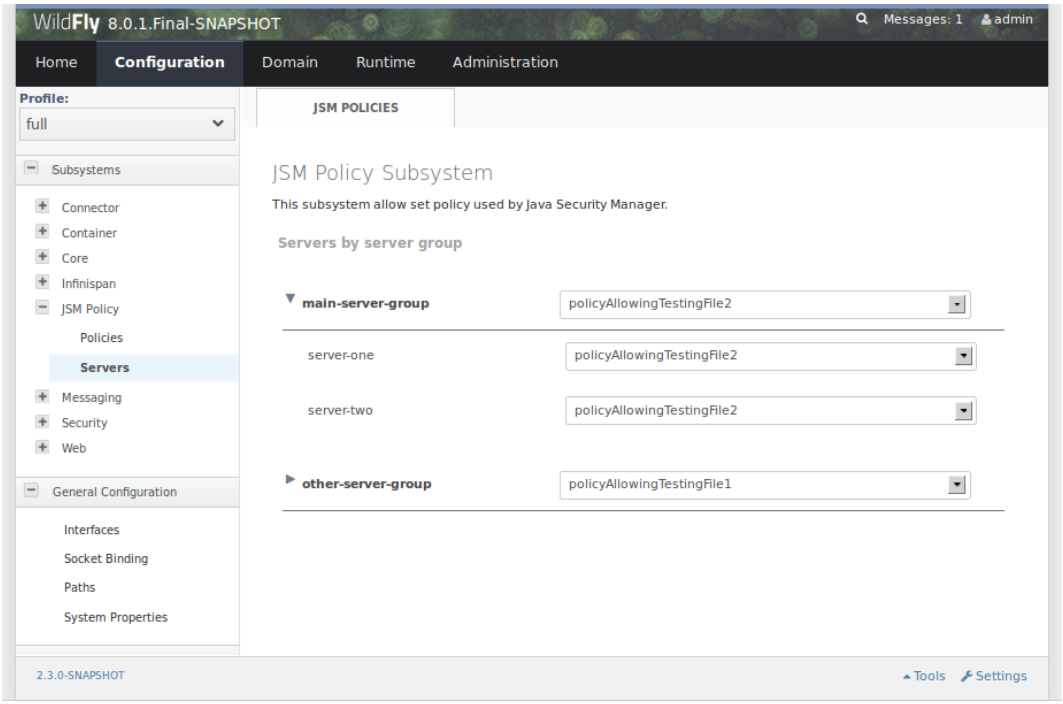
Na obrázku 4.2 je navíc vidět také základní ovládací prvky webové konzoly WildFly, které jsou v návrhu vyobrazeny jen schématicky. V levé části obrazovky se nachází menu subsystémů – umožňuje navigaci mezi konfiguračními obrazovkami jednotlivých subsystémů WildFly ve zvoleném profilu. Vrchní menu zobrazuje informace o systému, přihlášeném uživateli a umožňuje navigovat mezi jednotlivými částmi WildFly konzoly.

Odkaz na konfigurační obrazovky **Servers** a **Policies** se nachází v části **Configuration** (Konfigurace), je-li přepnuta na profil, v kterém je nainstalován implementovaný subsystém. Konfigurační obrazovka **JSM Policy**, umožňující sledovat politiky nasazené na jednotlivých serverech, je v části **Runtime** (Za běhu).

Webová konzola umožňuje také rozdělit jednu konfigurační obrazovku na více panelů. (Příklad panelu je na obrázku 4.1 popsán jako **Policies**.) Možnosti rozdělení konfigurační obrazovky na více panelů v tomto rozšíření webové konzoly není využíváno.



Obrázek 4.1: Grafický návrh uživatelského rozhraní plánovaného rozšíření WildFly



Obrázek 4.2: Výsledná podoba webového uživatelského rozhraní

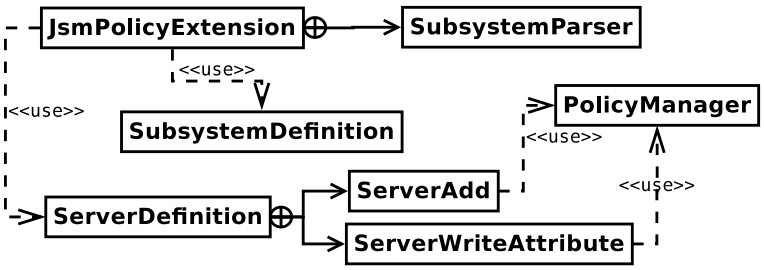
Kapitola 5

Popis implementace systému

Implementace řešení byla rozdělena do několika etap. V první etapě byl vytvořen první prototyp systému umožňující nasazení bezpečnostní politiky na základě hodnoty konfigurační vlastnosti, v které byla prozatím jen URL adresa tohoto souboru. Tento prototyp prokázal, že nasazování bezpečnostních politik implementovaným způsobem je možné.

Teprve následně byla implementována konečná verze řešení, jež do konfiguračního stromu WildFly ukládala soubory politik ve formě konfiguračních uzlů a uzly jednotlivých serverů jako odkazů na ně.

5.1 Prototyp subsystému



Obrázek 5.1: Diagram tříd prototypu

Prototyp subsystému byl založen na kostře subsystému z Maven repozitáře, podle dokumentace WildFly 8 věnující se právě vytváření rozšíření WildFly. [32] Z této kostry byl vytvořen subsystém, jehož jedinou funkcí bylo rozšíření konfiguračního stromu o uzel subsystému (`subsystem=jsmpolicy`) a jeho uzly `server=*`, uchovávající informaci o URL adrese souboru bezpečnostní politiky, jež má být použita na serveru daného jména. Diagram tříd celého prototypu ukazuje obrázek 5.1.

Byla tedy vytvořena třída definující rozšíření, `JsmPolicyExtension`. Její vnitřní třídou byla třída, zajišťující načítání a ukládání podstromu konfiguračního stromu patřící subsystému, `SubsystemParser`. Uzel `subsystem=jsmpolicy` byl definován třídou `SubsystemDefinition` a uzel serveru (`server=*`) třídou `ServerDefinition`. Oba uzly byly zaregistrovány jako součásti rozšíření `JsmPolicyExtension`.

V rámci konstruktoru třídy definující server bylo provedeno navázání akcí vytvoření serveru a odstranění serveru na třídy ošetřující tyto akce – `ServerAdd` a `ServerRemove`. V rámci metody `registerAttributes()` třídy definující uzel serveru byla zaregistrována

```
public void setPolicy(String policy) throws OperationFailedException {
    if (policy == null) {
        System.setSecurityManager(null);
    } else {
        System.setProperty("java.security.policy", policy);
        refreshDelegatingPolicy(); // DelegatingPolicy.delegate.refresh()
        System.setSecurityManager(new SecurityManager());
    }
}
```

Ukázka kódu 5.1: Metoda `PolicyManager.setPolicy()` zajišťující nasazení souboru bezpečnostní politiky

konfigurační vlastnost `policy`. Zápis do této konfigurační vlastnosti byl ošetřen třídou `ServerWriteAttribute`. Těmito třídami bylo na operace nad uzly serveru a jejich vlastnostmi navázána akce nastavení dané bezpečnostní politiky. Ta byla realizována třídní metodou `setPolicy()` třídy `PolicyManager`, zodpovídající za veškeré nastavování bezpečnostní politiky. Bezpečnostní politika tak byla na server nasazována při vytvoření odpovídajícího uzlu serveru, při změně obsahu jeho vlastnosti stanovující URL souboru bezpečnostní politiky, a odebírána při odstranění tohoto uzlu.

Samotnou metodu `setPolicy()`, zajišťující nasazení souboru bezpečnostní politiky, ukazuje ukázka kódu 5.1. Je-li jako parametr předána hodnota `null`, je správce bezpečnosti vypnut. Je-li jako parametr předán platný řetězec, je použit jako URL adresa souboru bezpečnostní politiky – je uložen do konfigurační proměnné `java.security.policy`, je vyžádáno znovunačtení bezpečnostní politiky a jako správce bezpečnosti je nastaven standardní správce bezpečnosti Javy.

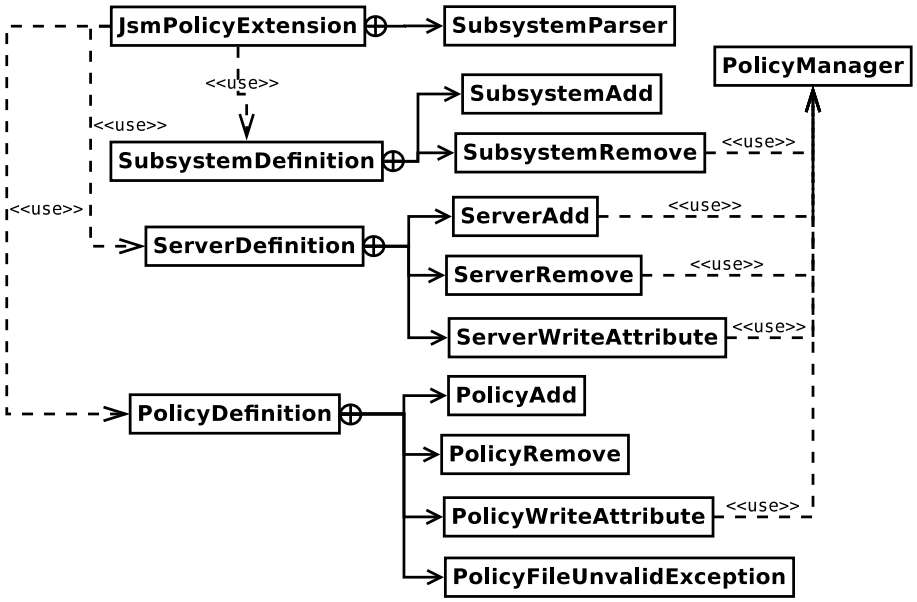
Samotné znovunačtení politiky přitom nemůže být provedeno pouhým voláním metody `refresh()` používaného objektu bezpečnostní politiky. Protože je jako objekt bezpečnostní politiky využíváno `DelegatingPolicy`, je nutné tuto metodu volat nad jeho privátní vlastností `delegate`, jak je již popsáno v kapitole 3.3.3. Metoda `refreshDelegatingPolicy()` zde svojí funkčností odpovídá ukázce kódu 3.5.

Subsystém v této verzi žádným způsobem nezajišťoval distribuci souborů bezpečnostní politiky – soubor musel být skrze uvedenou URL adresu přístupný ze serveru, na kterém měla být politika nasazena. V opačném případě se nasazení bezpečnostní politiky nezdařilo.

5.2 Konečná verze subsystému

Konečná verze subsystému se od prototypu liší zejména tím, že v konfiguračním stromu již nejsou ukládány jen URL adresy souborů politiky, ale také samotné soubory bezpečnostní politiky.

Byl tedy vytvořen další poduzel uzlu subsystému, `policy=*`. Tento nový uzel představuje bezpečnostní politiku a jeho vlastností `file` je obsah této politiky ve formě řetězce, tak jak by byl uložen v souboru bezpečnostní politiky. Zároveň byl změněn význam konfigurační vlastnosti `policy` uzlu `server=*`. Zatímco doposud obsahoval URL adresu souboru bezpečnostní politiky nasazené na daném serveru, nově obsahuje název uzlu `policy=*`, jehož hodnota atributu `file` má být použita jako obsah souboru bezpečnostní politiky pro daný server.



Obrázek 5.2: Diagram tříd konečné verze implementovaného rozšíření WildFly

Nadále tak nestačilo provádět nasazení bezpečnostní politiky při změně atributu `policy` uzlu `server=*`, ale bylo nutné bezpečnostní politiku znovu nasadit také při změně obsahu vlastnosti `file` uzlu `policy=*`.

Nasazování bezpečnostní politiky je řešeno analogicky jako v případě akce navázané na změnu vlastnosti serveru – třídou `PolicyWriteAttribute` ošetřující zápis do konfigurační vlastnosti `file`. Při definici této konfigurační vlastnosti byl navíc definován také objekt validátoru – objektu zajišťujícího kontrolu ukládané hodnoty konfigurační vlastnosti `file`. Není-li hodnota platným souborem bezpečnostní politiky, je uložení odmítnuto. Tato kontrola je prováděna za pomoci třídní metody `validatePolicyFile()` třídy `PolicyManager`. Aby bylo možné stejnou kontrolu provádět také při vytváření nového uzlu bezpečnostní politiky, musela být vytvořena třída `PolicyAdd` ošetřující vytváření těchto uzlů. Zároveň byla definována také třída `PolicyRemove` ošetřující operaci odstranění uzlu bezpečnostní politiky. Ta znemožňuje odstranění uzlu bezpečnostní politiky, která je aktuálně používána alespoň jedním serverem. (Existuje-li uzel serveru, jehož vlastnost `policy` obsahuje název odstraňovaného uzlu `policy=*`, není odstranění provedeno.)

Samotné nasazování bezpečnostní politiky zajišťuje metoda `setPolicyFile()`. Její hlavní část zachycuje ukázka kódu 5.2. Jak je vidět, k samotnému nasazení bezpečnostní politiky využívá metodu `setPolicy()` popsanou v kapitole 5.1. Jejím vstupem však není adresa souboru bezpečnostní politiky, ale jeho obsah. Není-li roven `null`, je vytvořen dočasný soubor `jsm-*.policy` a obsah souboru bezpečnostní politiky (`fileContent`) je uložen do něj. Dočasný soubor je jako soubor politiky nastaven metodou `setPolicy()` popsanou v kapitole 5.1 a smazán. V opačném případě, kdy je obsah bezpečnostní politiky roven `null`, je rovněž volána metoda `setPolicy()`, ale s parametrem `null`, čímž je používání bezpečnostní politiky a správce bezpečnosti vypnuto.

Jak je tedy vidět, používání souborů bezpečnostních politik uložených v konfiguračním stromu WildFly tak je ve své podstatě triviální.

```
public void setPolicyFile(String fileContent) {
 if (fileContent == null) {
  setPolicy(null);
 } else {
  String tempDirString=System.getProperty("jboss.server.temp.dir",null);
  File tempDir = tempDirString == null ? null : new File(tempDirString);
  File temp = File.createTempFile("jsm-", ".policy", tempDir);
  FileOutputStream out = new FileOutputStream(temp);
  out.write(fileContent.getBytes());
  out.close();
  setPolicy(temp.getAbsolutePath()); // use temp file as policy file
  temp.delete();
 }
}
```

Ukázka kódu 5.2: Klíčová část metody `setPolicyFile()` třídy `PolicyManager`

5.3 Webové uživatelské rozhraní

Webové uživatelské rozhraní umožňující správu implementovaného subsystému bylo implementováno tak, jak bylo navrženo v kapitole 4.4. Je implementováno jako záplata administrační konzoly WildFly, do které přidává konfigurační obrazovky popsané v kapitole 4.4.

5.3.1 Konfigurační obrazovka bezpečnostních politik

Obrazovka **Policies** (Politiky), umožňující spravovat bezpečnostní politiky nasaditelné na servery domény, byla vytvořena jednoduchou úpravou existující konfigurační obrazovky WildFly, **Mail**. Tato obrazovka umožňovala spravovat uzly subsystému `mail`. Jednoduchou úpravou tak byla vytvořena konfigurační obrazovka pro správu uzlů politiky.

Hlavní třídou konfigurační obrazovky je `PoliciesPresenter`. Tato třída zajišťuje komunikaci s konfiguračním stromem. Formou vnitřní třídy zároveň definuje rozhraní třídy zobrazující konfigurační obrazovku, `MyView`. Toto rozhraní implementuje třída `PoliciesView`, která z jednotlivých grafických komponent (`Widget`) skládá grafickou komponentu celé obrazovky. Hlavní z nich přitom je `PolicyEditor`. Tato grafická komponenta zahrnuje seznam bezpečnostních politik i formulář pro úpravu vybrané položky tohoto seznamu. Jeho součástí je také tlačítko pro přidání nové bezpečnostní politiky, které zobrazuje okno průvodce vytvořením. To je implementováno třídou `NewPolicyWizard`.

Jediná komplikace souvisela s grafickou komponentou textového pole, `TextAreaItem`. Neumožňovala uložit text obsahující znak nového řádku nebo uvozovek. Tento problém byl vyřešen vlastní podtřídou této komponenty, `NewlinedTextAreaItem`, která před tyto znaky vkládala znak zpětného lomítka. Stejná úprava na samotné komponentě `TextAreaItem` byla souběžně předána k začlenění a začleněna [16] do vývojové větve knihovny Ballroom, z které pochází.

5.3.2 Konfigurační obrazovka serverů domény

Obrazovka **Servers** (Servery), umožňuje nasazovat bezpečností politiky na jednotlivé servery a jejich skupiny. Vznikla úpravou konfigurační obrazovky **Runtime / JNDI View**.

Ta byla vybrána kvůli stromové struktuře, kterou dokázala zobrazit. Ta byla základním předpokladem pro naplnění grafického návrhu plánované konfigurační obrazovky.

Hlavní třídou konfigurační obrazovky je `ServersPresenter`. Vykreslení konfigurační obrazovky je zajištěno třídou `ServersView`, která do obsahu generované obrazovky vkládá komponentu `com.google.gwt.user.cellview.client.CellTree` z knihovny GWT. Tato komponenta zajišťuje samotné zobrazení stromové struktury. Její datový model je definován třídou `ServersTreeViewModel`, která komponentě `CellTree` zpřístupňuje informace o skupinách a serverech domény ve formě seznamu uzlů `Node`.

Vykreslení jednotlivých uzlů zajišťuje grafická komponenta `NodeCell`, jež implementuje rozhraní `AbstractInputCell`. To je nezbytné k tomu, aby mohla být použita k vykreslování položek `CellTree`. Uzel se vykresluje jako popisek s názvem serveru/skupiny a jako omezená nabídka umožňující nastavit politiku výběrem z politik dostupných v subsystému. Na změnu hodnoty této omezené nabídky je potom navázáno samotné nasazení bezpečnostní politiky.

5.3.3 Konfigurační obrazovka sledování nasazených politik

Obrazovka **JSM Policy** (Politika správce bezpečnosti Javy) umožňuje sledovat nasazené bezpečnostní politiky. Je velice jednoduchá – zobrazuje pouze název a obsah politiky aktuálně nasazené na vybraném serveru.

Hlavní třídou konfigurační obrazovky je `DetailPresenter`. Ta zajišťuje získání informace o aktuálně nasazené politice na zvoleném serveru z konfiguračního stromu WildFly. Vykreslení konfigurační obrazovky zajišťuje třída `DetailView`. Grafická komponenta konfigurační obrazovky se skládá z komponenty `TextBox`, textového pole používaného k zobrazení názvu nasazené politiky, a z komponenty `TextArea`, textového pole používaného k zobrazení obsahu nasazené politiky.

Zatímco předchozí konfigurační obrazovky jsou rozšířeními administrační konzoly typu `SubsystemExtension`, díky čemuž je lze najít pod záložkou **Configuration** v podmenu příslušného subsystému, obrazovka sledování je rozšířením typu `RuntimeExtension`. Zobrazuje se proto pod záložkou **Runtime**, v níž jsou nastavení specifická pro jednotlivé servery domény. Na tento globální výběr serveru je potom navázán také výběr sledovaného serveru domény.

Kapitola 6

Testování systému

Protože cílem této bakalářské práce nebylo jen vytvoření systému umožňujícího správu bezpečnostních politik nasazených na jednotlivých serverech distribuovaného systému, ale také prokázání jako kvality prostřednictvím testů, tato kapitola se zabývá tvorbou testů, umožňujících otestovat implementované řešení.

6.1 Analýza

Cílem implementace řešení bylo umožnění nasazování bezpečnostních politik na servery v doméně aplikačních serverů WildFly. Implementované řešení by mělo být schopné nasadit bezpečnostní politiku na specifikovaný server na základě požadavku skrze JBoss Management API. Testování by se tak jednoznačně mělo soustředit právě na schopnost systému nastavit na základě požadavku na stanoveném serveru stanovenou politiku. Testování této klíčové vlastnosti je možné provádět jedině v rámci testu integračního, tedy testu testujícího systém jako celek. Jednotkové testování v této oblasti sice rovněž může odhalit některé chyby, s hledáním klíčových chyb, vyplývajících z manipulací s bezpečnostními politikami pod WildFly, však pomoci nemůže.

Integrační test bude založen na testování schopnosti uživatelské aplikace přistoupit k souboru, k němuž je přístup omezen bezpečnostní politikou a na vyměňování bezpečnostní politiky skrze testovaný subsystém v průběhu testu. Vzhledem k tomu že nejsložitější možnou situací, která může v rámci jednoho serveru nastat je výměna jedné bezpečnostní politiky za druhou, k úplnému otestování implementovaného řešení postačují dva testovací soubory a dvě testovací bezpečnostní politiky. První bezpečnostní politika přitom bude přidělovat oprávnění přístupu pouze k prvnímu souboru a druhá pouze k druhému.

6.2 Implementace testů

Implementované integrační testy jsou tvořeny dvěma částmi – agentem a managerem.

Agent je standardní uživatelskou Java EE aplikací nasazovanou na testovaný aplikační server. Na základě požadavku na rozhraní REST (Representational State Transfer) se pokusí přečíst požadovaný testovací soubor. Následně vrátí řetězec `true` v případě že se mu to podaří. V případě že mu v tom bezpečnostní politiky zabrání, vrátí řetězec `false`.

Manager je běžným JUnit testem. Na základě komunikace s agentem prostřednictvím rozhraní REST zjišťuje, ke kterým testovacím souborům má agent oprávnění přistupovat.

Skrze rozhraní JBoss Management API pak je schopný požádat subsystém nainstalovaný na serveru o použití vybrané politiky na vybraný server.

Samotný test řídí Manager. Postupně žádá o nasazení jednotlivých souborů bezpečnostní politiky a testuje, zdali má Agent oprávnění přistupovat ke správné podmnožině z nich.

6.3 Testovací prostředí

Implementované testy předpokládají doménu aplikačních serverů o alespoň dvou WildFly serverech. Otestovanou verzí WildFly je 8.0.0.Final. Názvy těchto serverů a URL adresy na kterých poskytují své služby je možné nastavit v souboru `manager/src/test/java/org/picketbox/jsmpolicy/test/Constants.java`. Na stejném místě je možné nastavit také adresu doménového řadiče a port na kterém poskytuje HTTP Management API.

Tyto servery mohou ale také nemusí být servery jediného hostitele. Na všech hostitelích ale již musí být nainstalováno implementované rozšíření a záplata zavaděče tříd modulů WildFly nastavená na používání dynamických bezpečnostních politik. Samotný subsystém nemusí být na jednotlivých serverech nasazen, bude nasazen automaticky v rámci inicializace testu.

6.4 Výsledky testování

Všechny testy proběhly v rámci závěrečného testování implementovaného řešení v pořádku. Byla-li na server nasazena bezpečnostní politika, byla účinná ihned po skončení provádění příkazu provádějícího její nasazení. Byly-li na různé servery souběžně nasazovány dvě různé politiky, vzájemně se neovlivňovaly – na první server je vždy aplikována jen politika určená pro první server a na druhý server jen politika určená druhému serveru.

6.5 Zhodnocení

Testování prokázalo, že implementovaný subsystém dokáže bez problémů nastavovat bezpečnostní politiky serverů domény, aniž by se přitom bezpečnostní politiky nasazené na různých serverech domény vzájemně ovlivňovaly. Implementované řešení tedy dosáhlo cílů, které byly při jeho návrhu stanoveny.

Kapitola 7

Závěr

Cílem práce bylo navrhnout, vytvořit a otestovat systém pro centralizovanou správu a distribuci bezpečnostních politik Javy v distribuovaném prostředí aplikačního serveru WildFly. Ačkoli jen samotné používání bezpečnostní politiky ve WildFly bylo v době tvorby této práce bez znalosti zdrojového kódu nemožné (viz kapitola 3), byl v rámci této bakalářské práce nejen nalezen způsob jak bezpečnostní politiky ve WildFly používat, ale také vytvořen nový způsob, umožňující také její výměnu za běhu aplikačního serveru bez potřeby jeho restartu nebo znovunačtení. Dosaženo toho bylo za pomoci vytvořené záplaty (patch) WildFly (viz kapitola 3.3.2), která byla zároveň předána k začlenění do hlavní vývojové větve. [14]

Bylo vytvořeno rozšíření aplikačního serveru, umožňující provádět výměnu používané politiky a zapínat/vypínat správce bezpečnosti v návaznosti na změnu své konfigurace pro příslušný server v konfiguračním stromu WildFly. Ten se zároveň využívá také pro ukládání jednotlivých souborů bezpečnostní politiky, které jsou tímto způsobem distribuovány napříč servery domény.

Nad tímto rozšířením pak bylo vytvořeno webové uživatelské rozhraní ve formě rozšíření administrační konzoly WildFly. To umožňuje nastavovat a sledovat použití správce bezpečnosti a bezpečnostní politiky nasazené na jednotlivých serverech nebo celých skupinách serverů domény.

Na implementované řešení byla závěrem vytvořena automatizovaná sada testů, umožňujících otestovat schopnost implementovaného systému měnit oprávnění aplikací nasazených na doméně aplikačních serverů. Její spuštění prokázalo plnou funkčnost implementovaného řešení.

Možným vylepšením do budoucna by mohlo být umožnění nasazování bezpečnostních politik na celé skupiny aplikačních serverů, ne na úrovni webové administrační konzoly WildFly, ale již na úrovni JBoss Management API. Uživatel by tak mohl provádět nasazování politik na skupiny serverů vlastním skriptem, aniž by sám musel procházet jednotlivé servery této skupiny.

Rozsáhlejším možným rozšířením by mohlo být zadávání bezpečnostních politik již ne ve formě textu v syntaxi typické pro bezpečnostní politiky uložené v textových souborech, ale prostřednictvím specializovaného uživatelského rozhraní. To by mohlo umožnit zadávat bezpečnostní politiky po jednotlivých oprávněních za pomoci omezených nabídek. Uživatel by tak byl vystaven mnohem nižšímu riziku zadání neplatné politiky oproti stávajícímu řešení, kde se dozvídá o chybě v zadání politiky až při neúspěšném pokusu o její uložení.

Literatura

[1] OpenJDK source: java.lang.Class [online]. http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/lang/Class.java, [cit. 2013-05-01].

[2] jboss.org: Running JBoss with a Java 2 security manager [online]. https://docs.jboss.org/jbossas/docs/Server_Configuration_Guide/4/html/Security_on_JBoss-Running_JBoss_with_a_Java_2_security_manager.html, [cit. 2014-02-15].

[3] WildFly 8 Admin Guide – API reference [online]. https://docs.jboss.org/author/display/WFLY8/Management+API+reference, [cit. 2014-03-07].

[4] GitHub: jboss-modules / jboss-modules / src / main / java / org / jboss / modules / ModuleClassLoader.java [online]. https://github.com/jboss-modules/jboss-modules/blob/453d89c8a9ddb226c9223f19274df71f3affbad6/src/main/java/org/jboss/modules/ModuleClassLoader.java, [cit. 2014-04-19].

[5] Java Virtual Machine Specification: Chapter 4. The class File Format: 4.2. The Internal Form of Names [online]. http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html, [cit. 2014-04-20].

[6] Java SE Documentation: 6 Security Management: 6.1 Managing Applets and Applications [online]. http://docs.oracle.com/javase/7/docs/technotes/guides/security/spec/security-spec.doc6.html, [cit. 2014-04-22].

[7] OpenJDK source: sun.misc.Launcher [online]. http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/sun/misc/Launcher.java, [cit. 2014-04-22].

[8] JBoss Application Server 7 [online]. http://www.slideshare.net/rayploski/jboss-application-server-7, [cit. 2014-04-27].

[9] Java Virtual Machine Specification: Chapter 1. Introduction [online]. http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-1.html, [cit. 2014-05-01].

[10] Achmatowicz, R.: WildFly 8: Load Balancing [online]. https://docs.jboss.org/author/display/WFLY8/Load+Balancing, [cit. 2014-04-20].

[11] Connelly, D.: OpenJDK source: java.net.URLClassLoader [online]. http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/7u40-b43/java/security/ProtectionDomain.java, [cit. 2014-04-19].

[12] DeMichiel, L.; Shannon, B.: Java™ Platform, Enterprise Edition (Java EE) Specification, v7 [online]. https://java.net/downloads/javaee-spec/JavaEE_Platform_Spec_EDR.pdf, [cit. 2014-03-15].

[13] Kabir Khan, B. S.: JBoss AS 7: Domain Setup [online]. https://docs.jboss.org/author/display/AS7/Domain+Setup, [cit. 2014-02-12].

[14] Kalina, J.: jboss-modules/jboss-modules: Allowing use of non-static permissions #48 [online]. https://github.com/jboss-modules/jboss-modules/pull/48, [cit. 2014-04-26].

[15] Kalina, J.: PicketBox/SECURITY-778: DelegatingPolicy should delegate refresh() [online]. https://issues.jboss.org/browse/SECURITY-778, [cit. 2014-05-05].

[16] Kalina, J.: hal/ballroom: TextAreaItem – allowing \n and ” in textarea #1 [online]. https://github.com/hal/ballroom/pull/1, [cit. 2014-05-11].

[17] Li Gong, G. E., Roland Schemers: OpenJDK source: java.security.CodeSource [online]. http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/7u40-b43/java/security/ProtectionDomain.java, [cit. 2014-04-01].

[18] Li Gong, R. S.: OpenJDK source: java.security.SecureClassLoader [online]. http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/7-b147/java/security/SecureClassLoader.java, [cit. 2013-12-11].

[19] Li Gong, R. S.: OpenJDK source: java.security.CodeSource [online]. http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/7u40-b43/java/security/CodeSource.java, [cit. 2014-04-01].

[20] McAllister, N.: Red Hat renames JBoss application server as WildFly [online]. http://www.theregister.co.uk/2013/04/22/jboss_as_becomes_wildfly/, [cit. 2014-04-20].

[21] Oaks, S.: *Java Security*. O'Reilly, 1998, iSBN 1-56592-403-7E.

[22] Oracle: Java™ Platform, Standard Edition 7 API Specification: Class ClassLoader [online]. http://docs.oracle.com/javase/7/docs/api/java/lang/ClassLoader.html, [cit. 2013-12-11].

[23] Oracle: Default Policy Implementation and Policy File Syntax [online]. http://download.java.net/jdk8/docs/technotes/guides/security/PolicyFiles.html, [cit. 2013-12-14].

[24] Oracle: Java™ Platform, Standard Edition 7 API Specification: Class AccessController [online]. http://docs.oracle.com/javase/7/docs/api/java/security/AccessController.html, [cit. 2013-12-14].

[25] Oracle: Java SE Documentation: Java Rich Internet Applications Guide – Security [online]. http://docs.oracle.com/javase/7/docs/technotes/guides/jweb/security/security.html, [cit. 2013-12-19].

[26] Oracle: Java™ Platform, Standard Edition 7 API Specification: Class Policy [online]. http://docs.oracle.com/javase/7/docs/api/java/security/Policy.html, [cit. 2013-12-19].

[27] Oracle: The Java™ Tutorials: The Security Manager [online]. http://docs.oracle.com/javase/tutorial/essential/environment/security.html, [cit. 2014-01-15].

[28] Oracle: Java SE Documentation: Default Policy Implementation and Policy File Syntax [online]. http://docs.oracle.com/javase/7/docs/technotes/guides/security/PolicyFiles.html, [cit. 2014-03-25].

[29] Oracle: Java™ Platform, Standard Edition 7 API Specification: Class UnresolvedPermission [online]. http://docs.oracle.com/javase/7/docs/api/java/security/UnresolvedPermission.html, [cit. 2014-03-25].

[30] Oracle: Java™ Platform, Standard Edition 7 API Specification: Class SecurityManager [online]. http://docs.oracle.com/javase/7/docs/api/java/lang/SecurityManager.html, [cit. 2014-04-22].

[31] Stansberry, B.: JBoss AS 7: Detyped management and the jboss-dmr library [online]. https://docs.jboss.org/author/display/AS7/Detyped+management+and+the+jboss-dmr+library, [cit. 2014-03-05].

[32] Stansberry, B.: WildFly 8: Extending WildFly 8 [online]. https://docs.jboss.org/author/display/WFLY8/Extending+WildFly+8, [cit. 2014-03-05].

Příloha A

Obsah CD

- `projekt.pdf` – Technická zpráva ve formátu PDF (Portable Document Format)
- `projekt-tisk.pdf` – Technická zpráva ve formátu PDF ve formě určené pro tisk
- `projekt.tar.gz` – Zdrojové soubory technické zprávy
- `wildfly-8.0.0.Final.tar.gz` – Živé demo – instalace WildFly s nainstalovanými implementovanými rošířeními, použitelná pro předvedení bakalářské práce (použití v příloze C)
- `jsm-policy-subsystem.tar.gz` – Subsystém WildFly implementovaný jako hlavní součást této práce
- `jsm-policy-console-hal.tar.gz` – Rozšíření webové konzole WildFly umožňující správu výše uvedeného rozšíření (popsané v kapitole 4.4)
- `jsm-policy-test.tar.gz` – Repozitář integračních testů popsaných v kapitole 6
- `jboss-modules.tar.gz` – Záplata popsaná v kapitole 3.3.2 v odpovídajícím Git repozitáři

Příloha B

Postup instalace

Uvedený postup předpokládá operační systém Linux s nainstalovaným běhovým prostředím Javy. Testován byl na distribuci Ubuntu 13.10 s běhovým prostředím Javy OpenJDK 7. Pro odzkoušení systému je možné tento postup přeskočit a využít již nainstalovaný systém podle postupu v příloze C.

1. Připravte standardní instalaci aplikačního serveru WildFly. Doporučenou verzí je 8.0.0.Final. Postup by měl fungovat i pro novější verze.

2. Proveďte instalaci záplaty `jboss-modules`: (Její účel je popsán v kapitole 3.3.3)

   (a) Nakopírujte do zapisovatelného adresáře repozitář `jboss-modules`. Získat jej můžete z archivu `jboss-modules.tar.gz` na přiloženém CD nebo z GitHub: https://github.com/honza889/jboss-modules

   (b) Proveďte kompilaci za pomoci nástroje Maven – v adresáři `jboss-modules` spusťte: `mvn install`

   (c) Výsledným archivem (`target/jboss-modules-*-SNAPSHOT.jar`) nahraďte archiv `jboss-modules.jar` v adresáři WildFly.[1]

   (d) Na začátek spouštěcího skriptu, kterým budete WildFly spouštět, (např. `bin/domain.sh`) přidejte:[1]

   ```
   JAVA_OPTS="$JAVA_OPTS -Djboss.modules.policy-refreshable=true"
   ```

   (e) Po dalším startu by již WildFly měl používat dynamická oprávnění. (viz kapitola 2.5.6)

3. Proveďte instalaci rozšíření WildFly `jsm-policy-subsystem`:

   (a) Nakopírujte do zapisovatelného adresáře repozitář `jsm-policy-subsystem`. Získat jej můžete z archivu `jsm-policy-subsystem.tar.gz` na přiloženém CD nebo z GitHub: https://github.com/honza889/jsm-policy-subsystem

   (b) Proveďte kompilaci za pomoci nástroje Maven: `mvn install`

   (c) Výsledný adresář `target/module/org` nahrajte do adresáře `modules/system/layers/base/` v adresáři WildFly.[1]

[1] Proveďte na všech hostitelích domény

(d) Spusťe konzolu WildFly příkazem `bin/jboss-cli.sh` a následujícími příkazy v ní nainstalujte a přidejte subsystém.[2]

i. Provozujete-li WildFly v režimu **standalone**, použijte:

```
/extension=org.picketbox.jsmpolicy.subsystem:add
/subsystem=jsmpolicy:add
```

ii. Pro režim **domain** a profil **full** použijte:

```
/extension=org.picketbox.jsmpolicy.subsystem:add
/profile=full/subsystem=jsmpolicy:add
```

(e) Jestliže příkazy skončí úspěšně, rozšíření se subsystémem i samotný subsystém **jsmpolicy** byly úspěšně nainstalovány.

4. Proveďte instalaci rozšíření webové konzoly WildFly `jsm-policy-console-hal`:

(a) Nakopírujte do zapisovatelného adresáře repozitář `jsm-policy-console-hal`. Získat jej můžete z archivu `jsm-policy-console-hal.tar.gz` na přiloženém CD nebo z GitHub:
https://github.com/honza889/jsm-policy-console-hal

(b) Proveďte kompilaci za pomoci nástroje Maven: `mvn install`

(c) Výsledný archiv (`build/app/target/jboss-as-console-*-resources.jar`) nahrajte do adresáře `modules/system/layers/base/org/jboss/as/console/main` v adresáři WildFly.[1]

(d) Upravte soubor `module.xml`, aby atribut `path` elementu `resource-root` odpovídal názvu souboru zkopírovaném v minulém kroku.[1]

(e) Proveďte restart aplikačního serveru WildFly[1] a následně znovunačtěte webovou stránku s webovou konzolou WildFly. (Např. stisknutím klávesy F5 nebo Ctrl+F5 ve webovém prohlížeči.)

(f) Jestliže se ve webové administrační konzole v nabídce subsystémů (**Subsystems**) ukáže nová položka **JSM Policy**, bylo toto rozšíření úspěšně nainstalováno.

B.1 Možné příčiny problémů

1. V nabídce subsystémů ve webové konzoli se nezobrazuje subsystém JSM Policy.

(a) Ve webové konzole je vybraný profil, v kterém není nainstalován subsystém **jsmpolicy** (dle bodu 3 postupu). (V případě režimu **standalone**: Není nainstalován subsystém **jsmpolicy**.)

(b) Zkompilované rozšíření webové konzoly nebylo (dle bodu 4c postupu) nakopírováno do správného adresáře.

(c) V tomto adresáři nebyl upraven soubor `module.xml` tak, aby odkazoval na soubor nakopírovaného rozšíření. (dle bodu 4d)

(d) Webová stránka s webovou konzolí WildFly nebyla znovunačtena nebo server WildFly restartován (dle bodu 4e postupu).

[2] Proveďte jen na doménovém řadiči (domain controller/master)

2. Nedaří se vytvořit bezpečnostní politiku.
   (a) Bezpečnostní politika se zadaným názvem (Policy name) již existuje.
3. Nedaří se odstranit bezpečnostní politiku.
   (a) Bezpečnostní politika je nasazena na některém ze serverů. Odstranit ji je možné jen když není používána. (Z kterého serveru je politiku možné odstranit je možné zjistit z podrobností chybové zprávy.)

B.2 Spuštění integračních testů

Jestliže jste úspěšně dokončili instalaci rozšíření aplikačního, můžete provést instalaci a spuštění integračních testů popsaných v kapitole 6.

1. Nakopírujte do zapisovatelného adresáře repozitář `jsm-policy-test`. Získat jej můžete z archivu `jsm-policy-test.tar.gz` na přiloženém CD nebo z GitHub: https://github.com/honza889/jsm-policy-test
2. Ujistěte se že testovací servery testovací domény WildFly s nainstalovanými rozšířeními jsou spuštěné. Integrační test využívá dvou testovacích serverů.
3. Upravte v konfiguračním souboru nástroje Maven `agent/pom.xml` údaje nezbytné pro přihlášení k testovací doméně.
4. Upravte v souboru `manager/src/test/java/org/picketbox/jsmpolicy/test/Constants.java` informace o testovacích serverech a testovací doméně.
5. Spusťte skript `./test.sh`
6. Výsledky testů budou uloženy ve formě HTML dokumentu do souboru `manager/target/site/surefire-report.html`, který by měl být po skončení testů automaticky zobrazen.

Příloha C

Postup spuštění a používání

Uvedený postup předpokládá operační systém Linux s nainstalovaným běhovým prostředím Javy. Testován byl na distribuci Ubuntu 13.10 s běhovým prostředím Javy OpenJDK 7. Postup používání je pro domény s více hostiteli shodný.

1. Nainstalujte systém podle přílohy B nebo použijte předpřipravenou instalaci z přiloženého CD – nakopírujte do zapisovatelného umístění adresář `wildfly-8.0.0.Final` z archivu `wildfly-8.0.0.Final.tar.gz` na přiloženém CD.

2. Proveďte spuštění aplikačního serveru WildFly spuštěním spouštěcího skriptu `bin/domain.sh` v adresáři aplikačního serveru (novém umístění `wildfly-8.0.0.Final`).

3. Otevřete webové administrační rozhraní WildFly – v případě instalace WildFly z přiloženého CD jej najdete na adrese `http://localhost:9990/` a přihlásit se můžete přihlašovacím jménem `admin` s heslem `admin`.

4. Otevřete část **Configuration** a v nabídce **Profile** vyberte **full**.

5. Vytvořte novou bezpečnostní politiku pro otestování:

   (a) V nabídce subsystému **JSM Policy** vyberte stránku **Policies**.

   (b) Klikněte na tlačítko **Add**, do pole **Policy name** uveďte jako název vytvářené politiky `passwd-only` a potvrďte tlačítkem **Save**.

   (c) V seznamu dostupných politik (**Available JSM Policies**) označte vytvořenou politiku (`passwd-only`) a ve spodní části stránky klikněte na odkaz **Edit**.

   (d) Do pole **File content** ve spodní části stránky vyplňte požadovanou bezpečnostní politiku v syntaxi obvyklé pro soubory bezpečnostní politiky Javy. Například tedy:

   ```
   grant codeBase "vfs:/content/JsmPolicyTestingAgent.war/-" {
       permission java.io.FilePermission "/etc/passwd","read";
   };
   ```

   Uvedená politika umožní testovací aplikaci (součást integračních testů popsaných v kapitole 6) přistupovat k souboru `/etc/passwd`.

   (e) Úpravu bezpečnostní politiky dokončete tlačítkem **Save**.

6. Aby bylo možné testovat také výměnu bezpečnostní politiky, vytvořte stejným postupem také druhou bezpečnostní politiku. Jako její název (**Policy name**) uveďte `group-only` a její obsah (`File content`) vyplňte:

```
grant codeBase "vfs:/content/JsmPolicyTestingAgent.war/-" {
    permission java.io.FilePermission "/etc/group","read";
};
```

7. Zajistěte, aby byla na aplikačním serveru nasazena testovací aplikace. V případě předpřipravené instalace je tento krok možné přeskočit. Byly-li na tomto serveru již spouštěny integrační testy, je tento krok rovněž možné přeskočit.

   (a) Nakopírujte do zapisovatelného adresáře adresář `agent` z `jsm-policy-test`, repozitáře který můžete získat ze stejně pojmenovaného adresáře na přiloženém CD nebo z GitHub:
   https://github.com/honza889/jsm-policy-test

   (b) Běží-li vaše instalace na jiném počítači než lokálním, nebo používáte-li jiné názvy skupin serverů než jsou ve WildFly implicitní, nastavte potřebné údaje v souboru `pom.xml`.

   (c) Proveďte kompilaci a nasazení testovací aplikace spuštěním následujícího příkazu v adresáři `agent`:
   `mvn install wildfly:deploy`

   (d) Skončí-li operace zprávou `BUILD SUCCESS`, byla aplikace úspěšně nasazena.

8. Nastavujte použití vytvořených bezpečnostních politiky a sledujte jejich účinky na testovací aplikaci:

   (a) Otevřete stránku testovací aplikace otevřením následující webové adresy:
   http://localhost:8080/JsmPolicyTestingAgent/
   Uvedený port (8080) implicitně odpovídá serveru `server-one` ze skupiny `main-server-group`. Pro jiný server nebo jinou konfiguraci je nutné číslo portu patřičně poupravit.

   (b) V nabídce subsystému **JSM Policy** vyberte stránku **Servers**.

   (c) Klikněte na ikonu šipky na řádku skupiny serveru, na kterém chceme nastavení politiky vyzkoušet – pro `server-one` klikněte na ikonu u `main-server-group`.

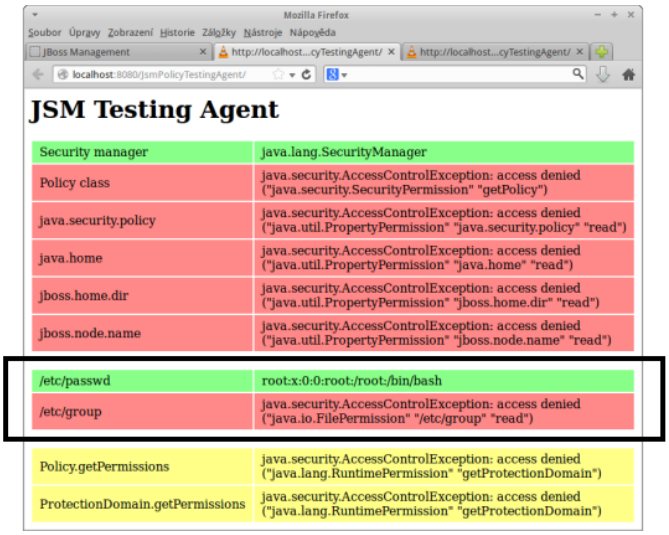
   (d) V zobrazeném seznamu serverů rozkliknuté skupiny změňte hodnotu v omezené nabídce testovacího serveru (`server-one`) na `passwd-only`.

   (e) Vraťte se ke stránce testovací aplikace a všimněte si řádků `/etc/passwd` a `/etc/group`, které ukazují, zda má testovací aplikace možnost číst dané soubory. Pokud ano, je zobrazen zeleně jejich první řádek, jako důkaz, že program soubor skutečně číst může. V opačném případě řádek obsahuje červeně popis výjimky, jež při pokusu o čtení z daného souboru nastala.

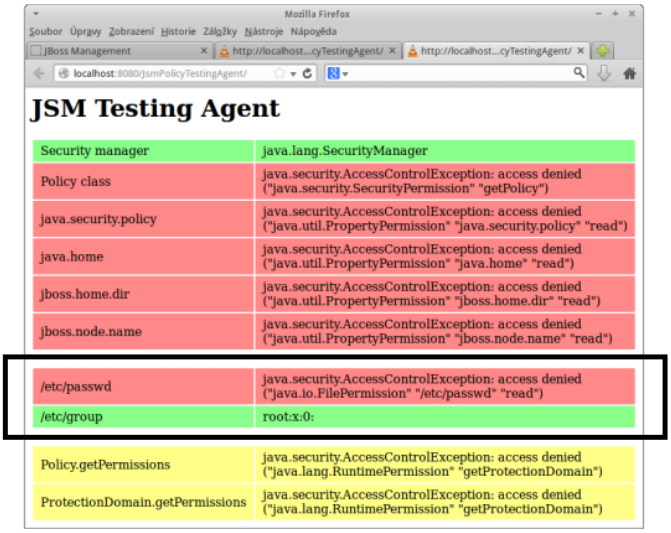
   (f) Stejným způsobem jako v předminulém bodu nastavte bezpečnostní politiku testovacího serveru na `group-only`.

   (g) Vraťte se opět ke stránce testovací aplikace, znovu ji načtěte (např. stiskem klávesy F5) a všimněte si, jakým způsobem se uvedené řádky změnili – měli byste si všimnout, že zatímco v prvním případě mohla testovací aplikace číst

soubor `/etc/passwd`, ale `/etc/group` ne (viz obrázek C.1), v druhém případě tomu bylo přesně naopak (viz obrázek C.2).



Obrázek C.1: Výstup testovací aplikace pro bezpečnostní politiku `passwd-only` – testovací aplikace může číst obsah souboru `/etc/passwd`, ale ne obsah souboru `/etc/group`



Obrázek C.2: Výstup testovací aplikace pro bezpečnostní politiku `group-only` – testovací aplikace může číst obsah souboru `/etc/group`, ale ne obsah souboru `/etc/passwd`