Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Bakalářská práce

Zabezpečení Linuxového serveru

Autor práce: **Tomáš Mudruňka**

Vedoucí práce: **doc. Ing. Arnošt Veselý, CSc.**

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Katedra informačního inženýrství

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Mudruňka Tomáš

Informatika

Název práce

Zabezpečení Linuxového serveru

Anglický název

Security of Linux server

Cíle práce

Cílem práce je stručně zmapovat současné bezpečnostní hrozby týkající se serverů využívajících operační systém Linux a možnou obranu proti nim. Poté si práce klade za cíl popsat bezpečnostní modul AppArmor a v poslední části předvést a otestovat jeho nasazení v praxi.

Metodika

V první třetině práce bude vypracován přehled základů zabezpečení serverů s OS Linux. Druhá třetina se zaměří na bezpečnostní modul AppArmor. Poslední třetině práce bude případová studie nasazení AppArmoru za účelem vyřešit konkrétní praktický problém.

Harmonogram zpracování

5/2014 - 12/2014 - studium a analýza zdrojů
1/2015 - kontrola postupu
1/2015 - 3/2015 - praktická část
3/2015 - odevzdání práce

+ praxe 120 hodin

Rozsah textové části
30-40 stran

Klíčová slova
Linux, server, bezpečnost, zabezpečení, jádro, modul, LSM, AppArmor

Doporučené zdroje informací

Jelínek L.: Jádro systému Linux, Computer Press, Brno, 2008
Nemeth E., Snyder G., Hein T.R.: Linux - kompletní příručka administrátora, Computer Press, Brno, 2004
Toxen B.: Bezpečnost v Linuxu, Computer Press, Brno, 2003
AppArmor. [online]. [cit. 2013-04-25]. Dostupné z: http://en.wikipedia.org/wiki/AppArmor

Vedoucí práce
Veselý Arnošt, doc. Ing., CSc.

Termín odevzdání
březen 2014

Ing. Martin Pelikán, Ph.D.
Vedoucí katedry

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE
Provozně ekonomická fakulta

prof. Ing. Jan Hron, DrSc., dr. h. c.
Děkan fakulty

V Praze dne 25.9.2013

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Zabezpečení Linuxového serveru" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

Podepsán Tomáš Mudruňka dne 23.11. 2014

Poděkování

Rád bych poděkoval vedoucímu bakalářské práce panu doc. Ing. Arnoštu Veselému, CSc. za konzultace a odborné vedení práce. Poděkování patří též rodině a všem ostatním, kteří mě při psaní této práce podporovali. Taktéž děkuji autorům svobodného softwaru, který je v práci popisován, stejně jako autorům softwaru, který byl při psaní práce použit (sémantický editor LyX a verzovací systém Git).

Zabezpečení Linuxového serveru
Security of Linux server

Abstrakt

V práci jsou stručně zmapovány současné bezpečnostní hrozby týkající se serverů využívajících operační systém Linux a možné obrany proti nim. Dále je v práci popsán bezpečnostní modul AppArmor a je demonstrováno jeho použití na praktickém příkladu.

Abstract

This thesis briefly informs about contemporary security threats related to computer servers running Linux operating system and possible defense against them. Thesis also describes AppArmor security module and demonstrates it's use on practical example.

Klíčová slova

Linux, server, bezpečnost, zabezpečení, jádro, modul, LSM, AppArmor

Keywords

Linux, server, security, kernel, module, LSM, AppArmor

Obsah

Seznam obrázků

Práce neobsahuje žádné obrázky.

Seznam tabulek

Práce neobsahuje žádné tabulky.

1 Úvod

Vzhledem k tomu, že operační systém Linux je velmi často používán na síťových serverech dostupných přes mezinárodní síť internet a je tak potenciálně vystaven případným útokům, je zabezpečení Linuxových systémů vždy vděčné téma. Proto se tato práce zabývá základními aspekty zabezpečení Linuxových systémů.

Kromě toho také existuje mnoho technologií, které umožňují zabezpečení Linuxového systému posílit nad rámec základní konfigurace. Jednou z těchto technologií je bezpečnostní modul AppArmor, který je v práci rovněž popsán včetně ukázky jeho praktického nasazení.

2 Cíl práce a metodika

Cílem práce je stručně seznámit čtenáře s problematikou zabezpečení Linuxových systémů. V práci budou popsány nejvýznamnější slabiny, se kterými se na Linuxových serverech můžeme běžně setkat, a hrozby, kterým tyto slabiny podléhají. Součástí práce bude i popis nějčastějších opatření, která se používají k odvrácení těchto hrozeb. Poté se práce zaměří na bezpečnostní modul AppArmor, zmapuje jeho oficiální dokumentaci a další prameny, které se po teoretické stránce zabývají jeho funkcionalitou. Následně bude demonstrováno nasazení tohoto modulu v praxi na modelovém případu. Zhodnocena bude také funkčnost celého řešení.

Tato práce je zpracována na základě analýzy odborné literatury a praktických zkušeností s nasazením popisovaných technologií. Vzhledem k povaze zvoleného tématu je velký důraz kladen na online zdroje a technickou dokumentaci popisovaných technologií.

3 Zabezpečení OS Linux (Přehled řešené problematiky)

V této kapitole je shrnuta problematika Linuxu a jeho zabezpečení.

3.1 Seznámení s OS Linux

Linux je jádro operačního systému, které pak spolu s dalšími programy v uživatelském prostoru[1] tvoří operační systém jako celek užitečný pro uživatele. Ačkoliv je Linux nezávislý na konkrétních userspace programech, je téměř výhradně provozován spolu s programy vytvořenými pod záštitou projektu GNU[2]. Tato kombinace softwaru se potom označuje jako GNU/Linux. GNU/Linux je potom dodáván a šířen formou tzv. "distribucí", což je většinou konkrétní kombinace Linuxového jádra a uživatelských programů předpřipravená a odladěná k použití tak, jak to koncový uživatel očekává. [1]

3.2 Zabezpečení počítačových systémů

O aspektech zabezpečení počítačových systémů toho bylo a ještě bude napsáno velmi mnoho. Ve své práci bych proto rád rozebral pouze společné jmenovatele všech bezpečnostních rizik, kterým Linux a podobné systémy podléhají.

Pro začátek lze bezpečnostní hrozby rozdělit do následujících skupin:

1. Bezpečnostní nedostatky hardwaru
2. Bezpečnostní chyby v jádře OS
3. Bezpečností chyby v uživatelském prostoru

Rozdělení je částečně dáno i samotnou architekturou použitého počítače. Veškeré nedostatky hardwaru a jejich aspekty jsou nad rámec této práce, protože není možné je ovlivnit pomocí žádného operačního systému. Naopak pokud by to možné bylo, už se nejedná o chybu HW, ale o chybu OS, který ji neošetřil[3]. Pro běžné uživatele a správce systémů je problematika HW bezpečnosti prakticky nedostupná a většinou nezbývá, než se spolehnout na kvalitu HW poskytnutého dodavatelem. Pochopitelně by bylo velmi náročné

[1] Uživatelský prostor (neboli "user space") je oddělená část paměti vyhrazená pro všechny procesy, které nejsou součástí jádra OS. Opakem tohoto termínu je termín jaderný prostor (také "kernel space").

[2] Rekurzivní akronym "GNU's Not Unix!", tedy "GNU není Unix!"

[3] Toto tvrzení je nutné brát s rezervou, protože není správné vždy řešit špatný návrh HW pomocí obezliček v OS.

a nákladné například ověřovat funkcionalitu mikrokódu procesoru, nebo dokonce samotného integrovaného obvodu, na kterém existuje[4]. Nicméně je přinejmenším dobré řídit se zásadami fyzické bezpečnosti a omezit fyzický přístup nepovolaných osob k serverům a okolní infrastruktuře, protože to je často jedno z největších bezpečnostních rizik. [2][6]

Stejně tak bezpečnostní chyby v jádře OS v této práci nebudou příliš rozebírány, protože v okamžiku, kdy je z bezpečnostního hlediska kompromitován samotný operační systém, nemá smysl se o dalším zabezpečení bavit. Proto je nutné udržovat v systému nejnovější otestovanou verzi jádra, u které byly opraveny všechny (veřejně známé) bezpečnostní chyby. Je velmi nepravděpodobné, že by správce serveru měl na odhalení a opravu případných chyb v jádře větší kapacitu, než všichni aktivní vývojáři dohromady. Z tohoto důvodu většina správců spoléhá se zabezpečním jádra právě na jeho vývojáře.

Poslední skupinou jsou bezpečnostní rizika v uživatelském prostoru. Právě těmi se bude tato práce zaobírat. Pokud vyjdeme z předpokladu, že máme k dispozici ideálně bezpečný HW a ideálně bezpečnou verzi jádra Linux, můžeme uvažovat o bezpečnosti programů, které pod tímto jádrem běží. Těmto programům v uživatelském prostoru budeme v praxi přirozeně chtít omezit pravomoce. Budeme chtít, aby každý uživatel (nebo služba), měl přístup výhradně k datům a prostředkům systému, které potřebuje a neohrozil tak soukromí, nebo funkčnost dalších uživatelů či služeb.

Největší potřebu hlídat tyto bezpečnostní aspekty máme u podsystémů, které jsou ve styku s nedůvěryhodnými osobami a dalšími systémy. Typicky jsou to služby využívané třetími osobami, služby veřejně dostupné přes počítačovou síť, nebo dokonce přes internet.

3.3 Základní návrh Linuxu z hlediska zabezpečení

Jak již bylo naznačeno, tak OS Linux umožňuje spouštět pod svojí kontrolou další programy a jako každý OS umožňuje jejich vzájemnou koexistenci a interakci, sdílení prostředků mezi těmito programy a řízení přístupu k jednotlivým částem systému. Právě toto řízení přístupu je pro tuto práci důležité.

[4]Přesto se v poslední době (po tzv. "aféře Snowden") tyto aspekty velmi řeší alespoň na úrovni národní bezpečnosti a šíří se například nepodložené zvěsti, že se tajné služby v Kremlu rozhodly nejcitlivější informace zpracovávat výhradně na psacích strojích až do doby, než bude Rusko schopné si pro státní účely vyrábět vlastní procesory založené na architektuře ARM.

3.3.1 Uživatelské účty a skupiny

Základním kamenem řízení přístupu na Linuxu (nikoliv však samospásným) jsou tzv. "uživatelské účty". Nemusí se jednat o účty skutečných uživatelů ve smyslu konkrétní osoby přihlášené do systému a využívající jeho prostředky. Velmi často jde také o účty určené pro běh více nebo méně interaktivní služby (tzv. "démona"). Podstatné je to, že proces spuštěný pod uživatelským účtem s omezenými pravomocemi nemůže tyto pravomoce získat zpět, pokud nejsou splněny určité okolnosti - např. získání práv od jiného programu, který těmito právy disponuje (např. na základě zadání hesla nebo zneužití bezpečnostní chyby). [3]

3.3.2 SuperUživatel (root)

Na typickém Linuxovém systému existuje uživatel pojmenovaný "root", který má maximální možná přístupová práva k celému systému a pokud je něčím omezen, tak jsou to spíše omezení rázu technického, než administrativního. Tak jako je každý uživatel označen svým uživatelským číslem, tak je i uživatel "root" označen číslem 0.

3.3.3 Přístupová práva k souborům

Samotné rozdělení spuštěných procesů pod uživatelské účty by nestačilo, také je nutné, aby systém daným procesům na základě těchto uživatelských účtů selektivně odmítal přístup k jednotlivým prostředkům. Typickým případem jsou soubory. Každý soubor v systému nese kromě svého názvu a obsahu také informaci o svém vlastníku a přístupových právech. Tato práva definují, kdo s tímto souborem smí manipulovat a jak. V základě lze oprávnění rozdělit na tato tři práva:

1. čtení (značí se "r" jako "Read")
2. zápis (značí se "w" jako "Write")
3. spouštění (značí se "x" jako "eXecute")

Tyto tři druhy práv je potom možné nezávisle přidělovat těmto třem skupinám uživatelů:

1. vlastník souboru (“user”)
2. skupina souboru (“group”)
3. všichni ostatní (“other”)

Vlastník souboru je ten, kdo soubor vytvořil, nebo komu byl přiřazen do vlastnictví správcem systému. [3][4]

Ke skupinám je nutné ještě doplnit, že každý uživatelský účet na Linuxu může být ještě navíc členem různých uživatelských skupin. Účel těchto skupin je dán potřebou sdílet některé soubory mezi více uživateli, ale zároveň nepovolit přístup úplně všem. [3][4]

3.3.4 Přístupová práva k dalším zdrojům

Kromě přístupu k souborům je samozřejmě také nutné řídit i přístup k dalším zdrojům. K tomu Linux používá dva dále popsané přístupy.

Jednak některé zdroje prezentuje jako soubory pomocí tzv. VFS[5], což umožňuje používat pro tyto zdroje stejný systém přístupových práv jako pro obyčejné soubory (např. právo na zápis do virtuálního souboru, který představuje tiskárnu, odpovídá právu na tisk na této tiskárně). [7]

A dále práva pro některé pokročilejší a více specifické úkony definuje jádro naprosto nezávisle na právech k souborům. Jako zástupce tohoto individuálně definovaného privilegia lze uvést možnost otevřít naslouchající síťový socket TCP na čísle portu nižším než 1024, což může výhradně uživatel root[6]. [8]

[5] Virtuální systém souborů (Virtual File System) je technika, kdy OS “předstírá” existenci souborů, které ve skutečnosti neexistují v klasické podobě na datovém úložišti. Místo toho jsou definovány čistě softwarově. Manipulace s nimi (čtení nebo zápis) nemá význam ve smyslu uchování dat, ale místo toho zprostředkuje komunikaci uživatelského prostoru s některým z podsystémů jádra, nebo hardwarovým vybavením. To umožňuje programům v uživatelském prostředí komunikovat s jádrem pomocí dobře známých metod pro práci se soubory.

[6] nebo proces s capabilitou `CAP_NET_BIND_SERVICE`

3.3.5 Capabilities

Ačkoliv uživatel root má v systému všechna myslitelná práva, je někdy vhodné jeho pravomoce omezit a někdy je naopak vhodné, aby proces měl některou z pravomocí tradičně patřící rootovi, ale jinak práva roota neměl. Proto Linux od verze 2.2 využívá navíc ještě tzv. "Capabilities". To jsou příznaky, které se nastavují jednotlivým běžícím vláknům a které umožňují dále omezovat práva i procesům běžícím pod uživatelským účtem root a naopak dávat speciální oprávnění procesům, které pod rootem neběží. Lze tedy nastavovat přístupová práva s větší granularitou, než tomu bylo před zavedením capabilities.

Různé funkce kernelu, u kterých vývojáři získali pocit, že by bylo vhodné jejich užití omezit jen na procesy se speciálními privilegii, byly opatřeny testem, který odmítne jejich provedení, pokud volající proces nemá nastavené patřičné capabilities. Bohužel mezi vývojáři kernelu panují nejasnosti ohledně toho, jakou konkrétní "capabilitu" přiřadit konkrétní vlastnosti jádra. A tak někteří vytvářejí pro každou funkci novou capabilitu, což ovšem může při nasazení nové verze kernelu způsobit znefunkčnění systémů se stávající konfigurací. Jiní přiřazují novým funkcím existující capability, což zase potencionálně způsobí nečekané zvýšení privilegií. Poslední skupina vývojářů nastavuje většině funkcí požadavek na capabilitu `CAP_SYS_ADMIN`. To bohužel popírá důvody nasazení tohoto systému, protože to vůbec neumožní jemné nastavení práv k jednotlivým akcím v systému, ale vrací nás to do stavu, kdy buď uživatel může všechno, nebo naopak nemůže skoro nic. [5]

Capability mohou být procesům nastaveny například pomocí PAM modulu `pam_cap.so` při přihlášení do systému. Také je možné příkazem `setcap` nastavit capability přímo spustitelným souborům podobně jako SUID[7] bit a umožnit tak i neprivilegovaným uživatelům využívat programy vyžadující tato privilegia[8].

Na této funkcionalitě taktéž staví mnoho dalších systémů. O capabilities tato práce bude ještě pojednávat v další části v souvislosti s modulem AppArmor.

[7] viz. níže

[8] Učebnicovým příkladem je program `ping`, viz níže.

3.4 Slabá místa a rizika

3.4.1 Síťové služby a pravděpodobnost jejich napadení

Síťové služby jsou programy, které vystavují své rozhraní do počítačové sítě a umožňují tak vzdálený přístup k prostředkům systému. To bohužel často znamená, že pokud je v takovém programu bezpečnostní nedostatek, je také možné ho zneužít na dálku.

Mnoho začínajících správců serverů má pocit, že je jejich server příliš nevýznamný na to, aby ho někdo napadal a nevěnují proto zabezpečení dostatečnou pozornost. To je však omyl, protože drtivá většina napadení je prováděna automaticky z různých zavirovaných, nebo jinak napadených počítačů, které dle více či méně náhodných vzorů prohledávají internet a hledají zranitelné síťové služby.

3.4.2 SUID programy

Typickým příkladem pontenciální bezpečnostní zranitelnosti v uživatelském prostoru jsou programy s nastaveným tzv. SetUID bitem. SetUID bit je další z příznaků, které je možné nastavit spustitelnému souboru v rámci dalších rozšíření k výše uvedeným přístupovým právům pro čtení, zápis a spouštění. Programy, které mají nastavený SetUID jsou systémem spuštěny pod uživatelským účtem jejich vlastníka, nikoliv pod uživatelským účtem, který dal pokyn k jejich spuštění. [3]

Takové programy pak často umožňují neprivilegovaným uživatelům provádět privilegované operace. Typickým příkladem může být příkaz `ping`, který ač je dostupný všem uživatelům, tak umožňuje odesílání ICMP Echo Requestů, což vyžaduje přímý a neomezený přístup k síťovému rozhraní. Dalším příkladem je program `passwd`, který umožňuje uživatelům změnit svoje heslo, které je přitom uložené v souboru, kam běžný uživatel jinak nemá přístup. [3]

Bezpečnostní riziko potom spočívá v tom, že pokud se uživateli podaří takový program donutit, aby udělal něco, co jeho autor nezamýšlel. Představme si například situaci, kdy by výše zmíněný program na změnu hesla obsahoval chybu, která by způsobila, že by za určitých okolností změnil heslo jinému uživateli než tomu, který ho spustil. A nebo ještě horší případ, že by bylo možné tento program donutit spustit jiný příkaz, dle volby

útočníka, právě ve chvíli, kdy běží s právy superuživatele. K tomu by mohlo dojít buď chybou v logice progamu, nebo například tzv. "přetečením bufferu". To si pro zjednodušení můžeme zatím představit jako situaci, kdy by uživatel takovému programu např. zadal heslo delší, než na jaké je stavěný a tento program obsahoval chybu, která by v takové situaci způsobila jeho pád nebo další nezvyklé nebo nepředvídatelné chování.

3.4.3 Přetečení bufferu

Jednou z nejznámějších zranitelností, se kterou se hojně setkáváme v programech nejen na Linuxu je přetečení bufferu. Jak již bylo naznačeno výše, tak typickým příkladem je situace, kdy programátor očekával vstupní data o určité délce, ale program pak za běhu dostal dat více. Problém nastává ve chvíli, kdy autor programu pevně stanovil velikost paměti, která je určená pro uložení vstupních dat před dalším zpracováním, nicméně kód samotný umožňuje načtení většího množství dat do této oblasti v paměti, protože délku dat nekontroluje nebo nelimituje.

Vstupní data se pak postupně zapisují na vyhrazené místo v paměti (tzv. buffer) a ve chvíli, kdy už se tam nevejdou, tak se zapisují stále dál do následujících oblastí v paměti, které však již byly vyhrazeny pro jiné účely. Takové situace lze potenciálně zneužít k ovlivnění chování programu. Například takovému programu lze podstrčit data do dalších bufferů uložených za bufferem přeteklým, a dokonce je možné mu podstrčit i úplně jiný spustitelný kód. Buffer se totiž nachází ve stejné oblasti (v tzv. stacku), kde se také nachází návratová adresa aktuálně běžící funkce, což je zjednodušeně adresa další části programu, která se má provést po dokončení aktuálně prováděné funkce. Útočníkovi pak stačí tuto adresu přepsat adresou přeteklého bufferu, do kterého podstrčil svůj vlastní spustitelný kód. [9]

Problematika tohoto druhu útoků proti spustitelnému kódu je ovšem složitější a její detailní popis je nad rámec této práce. Pro tuto práci je důležitý pouze fakt, že existuje možnost, jak donutit některé chybně napsané programy, ke spuštění kódu dle volby útočníka. Další část práce se totiž bude zabývat tím, jak zamezit takovým útokům a především nežádoucím důsledkům, které z nich mohou plynout. To vše bude rozebráno tak, aby nezáleželo na druhu útoku nebo jeho konkrétním provedení, ale pouze na tom, jaká aplikace bude napadena.

3.4.4 Typický scénář kompromitace systému

Pokud dojde ke kompromitaci zabezpečení serveru, typicky k tomu dojde ve dvou krocích. Nejdříve útočník získá nějakou minimální kontrolu nad systémem, což může být například omezený shellový přístup k neprivilegovanému uživatelskému účtu. K tomu dojde nejčastěji kompromitací některé síťové služby. V takovém případě má pak útočník stejné pravomoce, jako měla služba, kterou se mu podařilo kompromitovat. Následně se útočník buď pokusí tento omezený přístup k některým prostředkům serveru využít ve svůj prospěch (např. k rozesílání spamu a podobným aktivitám) nebo má ještě vyšší ambice a pokusí se prolomit další vrstvu zabezpečení a získat přístup k uživatelskému účtu s vyššími pravomocemi. Druhou možností se vyznačují především cílené útoky, při kterých jde útočníkovi o získání větší výhody, než jen možnosti zneužít cizích prostředků. Může jít například o osobní zájem na získání, poškození nebo zfalšování dat uložených na serveru.

Je nutné poznamenat, že jednou kompromitovaný server je nutné už vždy považovat za kompromitovaný. Existuje velmi mnoho technik, pomocí kterých si útočník může v systému ukrýt zadní vrátka. Některé z nich může být neúměrně složité odhalit a proto je často lepší a snadnější celý systém přeinstalovat.

3.5 Prostředky pro posílení zabezpečení kódu v uživatelském prostoru

Již byly zmíněny možnosti napadení spustitelného kódu pomocí přetečení bufferu ve stacku. Proto zde budou stručně zmíněny také možnosti obrany proti tomuto typu útoků přímo v kódu programů samotných. Na tomto místě budou zmíněny pouze stručně, protože později ještě uvedu komplexnější způsoby obrany.

3.5.1 Umístění bufferů a chráněných dat ve stacku

Jednou z prvních věcí, která se nabízí pro ochranu citlivých částí stacku (ukazatele, parametry funkcí,...) před přetečením bufferu, je jejich strategické umístění vůči bufferu. Může pomoci jejich umístění před buffer, nebo do náhodné vzdálenosti za buffer. Pokud je totiž pointer pokaždé v jiné vzdálenosti za koncem bufferu, je pro útočníka mnohem těžší správně umístit vlastní data na jeho místo.

3.5.2 Canaries

Využití tzv. canaries (doslova "kanárků"[9]) je technika, kdy se do stacku za buffer vloží náhodná hodnota, nebo kontrolní součet chráněných dat. Přetečení se potom kontroluje tak, že se testuje, jestli "kanárek" nebyl přepsán. [10]

3.5.3 GCC Stack Protector

Výše uvedené metody jsou většinou implementovány v rámci kompilátorů. Lze uvést alespoň jeden typický příklad.

GCC[10] je kompilátor programovacího jazyka C (případně C++), který je velmi hojně využíván ke kompilaci programů v rámci projektu GNU. Pomocí něj je sestavována i značná část programů využívaných v rámci GNU/Linuxu. Tento kompilátor má možnost aktivovat tzv. stack protector, což je soubor technik, které pomáhají zlepšit odolnost výsledného binárního kódu vůči přetečení ve stacku. Stack protector lze aktivovat nastavením překladače bez potřeby upravovat samotné zdrojové kódy programu. Přesto je i tak více než vhodné dodržet při programování v jazycích tohoto typu zásady bezpečného kódu a zabránit tak přetečení dokonale. [10]

3.5.4 Seccomp

Seccomp (zkráceně "secure computing mode") je funkce kernelu, která umožňuje kódu v uživatelském prostoru uzavřít se z hlediska jádra do sandboxu. Tento režim se aktivuje pro proces, který si ho vyžádal pomocí volání `prctl()` s argumentem `PR_SET_SECCOMP`.

Pokud si některý proces vyžádá vstup do seccomp režimu, kernel mu odmítne veškerá systémová volání až na výjimky. Tyto výjimky mohou být určeny pomocí dvou režimů seccompu.

První z těchto režimů se nastavuje argumentem `SECCOMP_MODE_STRICT` a povoluje pouze volání `read()`, `write()`, `_exit()`, a `sigreturn()`. Což zjednodušeně znamená, že proces může pouze zapisovat nebo číst z předem otevřených souborů a socketů, potom se

[9] Jde o narážku na využití živých kanárků horníky pro testování dýchatelnosti vzduchu v dolech.
[10] GNU C Compiler

může také ukončit. V případě, že se kód pokusí zavolat jiné systémové volání, je kernelem ukončen pomocí signálu `SIGKILL`.

Druhý režim, který seccomp nabízí je nastaven argumentem `SECCOMP_MODE_FILTER` a umožňuje programátorovi nastavit, která systémová volání mají být povolena a která ne. K tomu se používá systém Berkeley Packet Filter, což je systém původně vyvinutý na filtrování síťového provozu, ale nalezl svoje využití i v této poněkud odlišné oblasti. Pokud je v tomto filtru povoleno procesu spouštět další procesy pomocí `fork()` nebo `execve()`, přenáší se nastavení seccompu i na takto spuštěné procesy, aby nebylo možné se z jeho vlivu takto vymanit.

Ve výsledku umožňuje seccomp programátorům zajistit procesy potencionálně vystavené bezpečnostním hrozbám před zneužitím případným útočníkem. Příkladem může být proces, který pouze přijme data ze sítě, provede nad nimi výpočet a výsledek opět vrátí po síti. U takového procesu se nepočítá s tím, že by měl například zapisovat do souborů na disku, nebo spouštět další programy. Proto může být žádoucí se takové možnosti vzdát úplně. V případě, že by se pak útočníkovi podařilo proces kompromitovat nalezením chyby ve zpracování vstupu, bude tento proces zabit při první snaze takto získané nadvlády zneužít. [11]

Základní ukázku použití seccompu při programování v jazyce C naleznete v příloze[11].

3.6 Prostředky pro posílení zabezpečení ze strany jádra

V této části budou rozebrány metody zabezpečení, které poskytuje Linuxové jádro a které mohou fungovat bez ohledu na typ útoku a to dokonce nejen v případě, že aplikace obsahuje bezpečnostní chybu, ale také v případě, že sama aplikace je (potenciálně) záměrně škodlivá. To však vyžaduje dostatečné pochopení dané problematiky, volbu správné metody a její korektní nasazení.

[11] Při studiu základů Seccompu jsem naprogramoval tento testovací program, abych si Seccomp prakticky vyzkoušel a zároveň si ověřil jeho funkčnost. I přesto, že se tato práce Seccompem primárně nezabývá, přišlo mi zajímavé ho k práci přiložit pro případné zájemce.

3.6.1 Firewall

Jeden ze základních prostředků, jak může Linuxové jádro omezit chování programů v uživatelském prostoru je firewall. Firewall slouží výhradně k omezení síťového provozu a ostatním aktivitám aplikace zabránit nemůže. Přesto bývá často vítaným doplňkem, protože umožňuje například omezit dostupnost některých síťových služeb na vybrané klienty a zamezit tak potenciálním rizikům ze zbytku internetu.

Na Linuxu se firewall nalézá přímo v jádru pod názvem "netfilter" a není ho potřeba instalovat. Je potřeba pouze userspace software k jeho nastavení. Základní nastavení firewallu se provádí pomocí programu `iptables` (případně `ip6tables` pro IPv6, `arptables` pro ARP a `ebtables` pro nastavení firewallu na bridgích). To jsou také názvy jaderných rozhraní pro konfiguraci netfilteru z uživatelského prostoru. V současné době se však zvažuje nahrazení tohoto rozhraní pomocí rozhraní `nftables`, které má sjednotit všechny rozhraní firewallu.

Zajímavostí netfilteru je také to, že umí síťový provoz jednotlivých aplikací zpracovávat i na základě toho, pod jakým uživatelským účtem aplikace běží. Netfilter neslouží jen k omezování procesů běžících v userspacu, ale taktéž samozřejmě zvládá řídit provoz například routovaný nebo bridgeovaný přes Linuxový systém. To však není pro tuto práci příliš zajímavé, protože takový provoz jen zřídka znamená bezpečnostní trhlinu a pokud ano, tak způsobí většinou spíše přetížení systému, než aby umožnil útočníkovi získat skutečnou kontrolu nad systémem. [12]

3.6.2 Chroot

Další z vlastností jádra, která je často zmiňovaná v souvislosti se zabezpečením, je tzv. "chroot", což je zkratka, která znamená "change root". Pro pochopení toho co chroot dělá, je nutné nejříve pochopit, že výraz "root" není v souvislosti s Linuxem používán pouze pro označení superuživatele. Je také používán k označení kořenu systému souborů[12], ve kterém se pak do podsložek připojují další systémy souborů. Chroot je funkce Linuxového jádra, která umožňuje změnit tento adresář pro nově spouštěný proces[13]. To například umožňuje s jistými omezeními v rámci jednoho Linuxového systému spustit druhý. Oba poté běží

[12] tj. adresář "/"

[13] Z toho tedy vyplívá, že kořenový adresář nemusí být pro všechny procesy stejný.

pod stejným jádrem a sdílí stejné prostředky. Programová výbava uživatelského prostoru takto spuštěného systému je pak naprosto nezávislá, používá vlastní knihovny[14], atd... [13]

Problém s chrootem však je, že víceméně nepředstavuje žádné skutečné zabezpečení navíc a je možné se z omezení systému ohraničeného chrootem vymanit a získat zpět kontrolu na nadřazeným systémem, ze kterého byl tento chroot spuštěn. Obzvlášť, pokud uvnitř chrootu existuje možnost získat práva uživatele root. Taktéž je nutné poznamenat, že chroot upravuje pouze přístup k systému souborů a nikoliv k ostatním prostředkům systému. [13][16]

Chroot je tedy spíše praktickým nástrojem v rukou správce systému. Osobně nedoporučuji využívat ho jako základní kámen zabezpečení systému. Přesto však může být v některých případech chroot žádaným doplňkem k zabezpečení či izolaci některých procesů na serverech.

3.6.3 Nice, limits

Tyto dvě vlastnosti jádra zmíním jen stručně, protože samy o sobě neposkytují opravdové zabezpečení. Umožňují zabezpečení posílit a nastavit určité mantinely pro jednotlivé uživatele a procesy. Toho často využívají další komplexnější bezpečnostní technologie k doplnění svojí funkcionality. Samostatné využití těchto funkcí je však také velmi časté, nehraje však zásadní roli pro zabezpečení.

Nice určuje převrácenou prioritu procesu z hlediska plánovače procesů. Čím větší hodnotu nice má proces, tím nižší má prioritu[15]. Tato hodnota může nabývat hodnot od -20 do 19, přičemž krajní hodnoty může procesu nastavit jen uživatel root. Vysoké hodnoty nice znamenají, že jádro provádění takového procesu naplánuje pouze pokud nemá vůbec nic jiného na práci. Záporné hodnoty naopak procesu umožňují na vytíženém systému získat přednost. To se hodí například, pokud uživatel chce provádět náročné výpočty a zároveň poslouchat hudbu nebo sledovat film. Pokud se provádění hodinu trvajícího výpočtu občas zastaví, tak to nepředstavuje problém, ale přerušovaná reprodukce hudby může být pro uživatele nepříjemná. Stejně tak na serverových systémech může být požadováno v rámci

[14]Můžeme například spustit pod již běžícím systémem jinou distribuci GNU/Linuxu, nebo dokonce v 64b systému spustit 32b chroot a používat tak programy nedostupné pro 64b systémy.

[15]a tím je tedy "hodnější" (odtud "nice") při dělení se o prostředky s ostatními procesy

údržby provádět nějakou náročnější operaci, ale zároveň nesmí být narušena dostupnost poskytované služby. Také je nutné dát velký pozor na procesy s extrémně nízkou hodnotou nice. Pokud proces s nice -20 (tzv. real-time režim) skončí v nekonečné smyččce, je velmi obtížné až nemožné ho ukončit. Tento proces totiž vytlačí z provádění ostatní procesy, přes které by ho bylo možné ukončit. Většina procesů v systému má po startu hodnotu nice 0. Výjimky jsou nastaveny spíše dle potřeb uživatelů pro pohodlnou práci se systémem. Hodnota nice se nastavuje procesům nejčastěji příkazy `nice` (pro nově spouštěné procesy) a `renice` (pro procesy již spuštěné). [14]

Limity jsou funkce Linuxu umožňující nastavit procesům a uživatelům omezení ve využívání snadno počitatelných prostředků systému. Na typickém Linuxovém systému se nastavují v souboru `/etc/security/limits.conf` a v platnost je nejčastěji zavádí pam modul `pam_limits.so`. Současně umožňují nastavit asi 30 různých parametrů, z nichž lze jako zástupce zvolit maximální počet otevřených souborů, využití paměti, maximální počet využitých procesů, maximální počet současných přihlášení uživatele do systému a krajní hodnoty nice, jaké může proces nebo uživatel nastavit. [3]

3.6.4 Cgroups

Cgroups je systém, který umožňuje sdružovat procesy do skupin, kterým je následně možné nastavovat další pravidla. Základní funkcionalita umožňuje procesy omezovat podobně jako to dělají limity nebo nice. Nastavení však probíhá poněkud komplexněji.

Tím však možnosti cgroups zdaleka nekončí. Velice zajímavá je například izolace těchto skupin pomocí jmenných prostorů[16]. Ta umožňuje různé pohledy na sdílené prostředky systému. Například je možné zajistit, že procesy uvnitř takové skupiny vidí jen ostatní procesy v této skupině, ale zbytek procesů v systému už nevidí. Nebo je možné přesměrovat adresáře na disku, takže například každá služba, která přistupuje k dočasnému adresáři `/tmp`, uvidí ve skutečnosti jiný adresář vyhrazený jen pro ni a tudíž neuvidí dočasné soubory jiných služeb zařazených do jiných skupin. [15]

Ačkoli je takto snadné zvýšit zabezpečení z hlediska oddělení jednotlivých služeb, tak toto nastavení lze provést zcela transparentně bez nutnosti úpravy jejich spustitelného kódu. Podobně se dá odstínit například i přístup k síťovým rozhraním, atd...

[16] tzv. namespaců

Další zajímavou možností je tzv. checkpoint/restore celé cgroupy. Novější verze jádra umožňují uložit stav běžícího procesu na disk a později se k němu vrátit. To lze použít buď lokálně, nebo je dokonce možné takto přenést běžící proces na úplně jiný systém. Případně je takto možné provést restart systému bez nutnosti ukončit proces, který se pak dá načíst do paměti, aby mohl pokračovat tam, kde předtím přestal. Cgroups toto rozšiřují o možnost udělat checkpoint celé skupiny procesů.

To vše je často využito jako základ různých kontejnerových technologií, které budu rozebírat dále.

3.6.5 Kontejnery

Existuje několik technologií, které vzaly některé myšlenky chrootu a dotáhly je téměř k dokonalosti v tom smyslu, že jsou vhodné i pro bezpečné oddělení dvou a více systémů. Většinou jde o sady patchů do jádra, které umožňují pod Linuxem spouštět další skutečně nezávislé Linuxové userspace systémy. V takovém případě tyto sice sdílí jádro, ale už mají lépe oddělený i přístup dalším prostředkům jádra. Jsou navrženy tak, aby z nich nebylo možné vystoupit ani v případě, že má uživatel uvnitř práva uživatele root a aby se nemohly vzájemně ovlivňovat. Takové kontejnery lze proto použít k oddělení aplikací, které zajistí, že v případě bezpečnostní chyby v jedné aplikaci nezíská útočník přístup mimo kontejner a k aplikacím ostatním. To ale vyžaduje, aby každá aplikace měla kompletní vlastní systém, což nemusí být vždy žádoucí a efektivní. Jindy to naopak může být velmi výhodné a je běžné, že se takové kontejnery pronajímají k libovolnému použití zákazníkům, kteří si nechtějí kupovat celý server.

Mezi tyto kontejnerové technologie patří dnes již zastaralý Linux vServer, momentálně nejrozšířenější OpenVZ a nakonec projekt LXC (Linux Containers), který je přímo součástí oficiální verze jádra. Poslední jmenovaný má ambice nahradit dva předchozí jmenované projekty, ale v době psaní této práce zatím bohužel není připraven k produkčnímu použití a obsahuje ještě jisté nedostatky.

3.6.6 Bezpečnostní sady patchů do jádra

Existují projekty, které poskytují sady patchů, které si kladou za cíl zlepšit zabezpečení Linuxu pomocí různých druhů úprav zdrojových kódů a nastavení. Jde především o úpravy,

které jinak vývojáři nechtějí do jádra zařazovat. Důvodem je například, že narušují některou jinak žádanou funkcionalitu pro některé krajní případy, nebo jsou jinak v rozporu se záměry vývojářů jádra. Jejich nasazení často vyžaduje, aby si správce jádro sám sestavil ze zdrojových kódů, na které předtím tyto patche aplikoval. [21]

Je však nutno podotknout, že samotné nasazení takových patchů samo o sobě není dostatečné zabezpečení. Jde spíše o doplněk, protože použití takto upraveného jádra nijak nedefinuje žádná omezení pro jednotlivé procesy. Stejně jako použití zimních pneumatik na vozidle nijak nezaručí, že nedojde k havárii.

Některé z těchto projektů však poskytují poměrně sofistikované nástroje nad rámec pouhého vyčištění jádra od potenciálních hrozeb.

Mezi nejznámější projekty tohoto typu patří grsecurity a PaX. [21]

3.6.7 LSM - Linuxové bezpečnostní moduly

Poslední a asi nejzajímavější možností, jak zlepšit zabezpečení aplikace v uživatelském prostoru, jsou takzvané LSM (Linux Security Modules). Jde o framework, který je dostupný v Linuxovém jádře od verze 2.6 a který umožňuje implementaci různých bezpečnostních politik typu MAC (Mandatory Access Control - Mandatorní řízení přístupu). To umožňuje správci systému jednotlivým procesům ještě přesněji a závazněji nadefinovat přístupová práva a privilegia, ve srovnání například se souborovými právy. [17]

Výhoda tohoto systému spočívá kromě možnosti jemněji definovat množinu povolených operací a přístupných zdrojů také v tom, že uživatel nemůže sám (ať už záměrně nebo omylem[17]) dát práva jiným uživatelům. [17]

Mezi nejznámější LSM patří AppArmor, SELinux[18], Smack a TOMOYO Linux. V další části této práce bude podrobněji popisováno fungování a užití pouze systému AppArmor. Ten sice není tak komplexní jako SELinux a některé politiky nedokáže definovat tak přesně, ale je nesrovnatelně snažší ho začít používat bez potřeby hlubokých znalostí. To ho možná v některých případech činí i bezpečnějším. Bezpečnost systému totiž není daná jen tím,

[17]Typicky se stává, že neznalí uživatelé ve snaze vyřešit problém s přístupovými právy jednoduše nastaví souboru plná práva pro všechny uživatele. To téměř vždy jejich problém vyřeší, ale může to mít i nepříjemné následky.

[18]Ten byl mimochodem vyvinut Národní bezpečnostní agenturou (NSA) Spojených států amerických.

jak silný LSM správce použije, ale také tím, jak se ho naučí používat a jak dobře mu porozumí.

4 AppArmor (úvod)

Tato část se zabývá teoretickým rozborem fungování AppArmoru.

4.1 Co je AppArmor

Jak jsem již uvedl, AppArmor je Linuxový bezpečnostní modul (LSM), který implementuje mandatorní řízení přístupu (MAC) k prostředkům systému. [18]

4.2 Instalace modulu AppArmor

Některé distribuce Linuxu již mají kernel sestavený s alespoň základní podporou AppArmoru. Pokud ovšem potřebujete kernel sestavit sami, nebo se jen chcete dozvědět více, přečtěte si tuto část.

4.2.1 Sestavení jádra s podporou AppArmoru

AppArmor je sice již nyní součástí hlavní větve kernelu, nicméně pouze částečně. Ne všechny jeho součásti se do jádra dostaly a pokud budeme chtít využít AppArmor naplno, bude nutné aplikovat patche přímo od jeho vývojářů. Pokud se rozhodneme využít pouze "vanilkový kernel"[19], musíme se připravit na to, že přijdeme o novější a pokročilejší funkce, jako je například limitování síťového provozu pomocí AppArmoru. Linux se vyvíjí velmi rychle a je pravděpodobné, že tato informace nebude již krátce po publikaci této práce

[19] tzv. "vanilla kernel" je označení pro jádro z hlavní větve, která je momentálně spravovaná původním autorem Linuxu (Linusem Torvaldsem) a celosvětovou komunitou vývojářů, kteří jsou s ním v úzké spolupráci. Tato verze neobsahuje žádné úpravy, které neprošly jejich schvalovacím procesem. Není ovšem vyjímkou, že uživatelé používají jádra s neoficiálními úpravami od třetích stran. [19] Označení má svůj původ ve vanilkové příchuti zmrzliny, která tvoří jakýsi základ, ke kterému se pak mohou přidávat další polevy.

platit. Pravděpodobně však také vzniknou nové verze AppArmoru, které budou na zařazení čekat. [19]

Při kompilaci je nutné kernel správně nastavit[20], aby se AppArmor do kernelu zakompiloval. To provedeme pomocí následujících parametrů:

```
CONFIG_SECURITY_APPARMOR=y
CONFIG_SECURITY_APPARMOR_BOOTPARAM_VALUE=0
#CONFIG_DEFAULT_SECURITY_APPARMOR is not set
```

První parametr způsobí, že se AppArmor do kernelu vůbec zakompiluje. Zbylé parametry způsobí, že bude AppArmor ve výchozím stavu vypnutý, dokud ho nepovolíme v zavaděči pomocí parametru kernelu[21] `apparmor=1 security=apparmor`, ale lze také použít následující nastavení:

```
CONFIG_SECURITY_APPARMOR=y
CONFIG_SECURITY_APPARMOR_BOOTPARAM_VALUE=1
CONFIG_DEFAULT_SECURITY_APPARMOR=y
CONFIG_DEFAULT_SECURITY="apparmor"
```

Kernel sestavený s takovým nastavením naopak bude mít AppArmor zapnutý i bez dodatečných parametrů zadaných při bootu, nicméně bude pak možné AppArmor zakázat ze zavaděče pomocí parametrů `apparmor=0 security=""`.

Po nabootování do takto sestaveného kernelu použijeme následující příkaz pro ověření podpory AppArmoru:

[20] Při sestavování binárního obrazu z jádra se nabízí opravdu velké množství voleb, kterými můžeme obraz přizpůsobit pro svoje potřeby. Volit můžeme jak zásadní věci jako je např. cílová architektura procesoru, pro kterou obraz sestavujeme, tak i jednotlivé vlastnosti, které chceme do kernelu zahrnout ať už přímo, nebo formou dynamicky zaveditelných modulů. Toto nastavení se bude pochopitelně lišit například pro osobní počítač a nebo pro mobilní telefon, ve kterém nebudeme chtít zbytečně zabírat paměť zařízení velkým kernelem se spoustou funkcí a ovladačů na hardware, který pravděpodobně k takovému zařízení nikdy nepřipojíme. Toto nastavení samozřejmě poté už nelze měnit bez rekompilace celého kernelu. Konfiguraci aktuálně nabootovaného systému můžeme přečíst ze souboru `/proc/config` nebo `/proc/config.gz`, pokud to ovšem není v configu zakázáno. V takovém případě záleží na distribuci systému, jakým způsobem její správci o konfiguraci uvědomí koncové uživatele.

[21] Narozdíl od konfigurace kernelu, kterou provedeme při kompilaci, je kernelu navíc možné předat množství dalších parametrů během bootu. Tyto parametry předává kernelu typicky zavaděč a aktuální parametry, se kterými je systém nabootován můžeme většinou přečíst z virtuálního souboru `/proc/cmdline`.

```
cat /sys/module/apparmor/parameters/enabled
```

Mohou nastat tři situace:

1. Je vypsána chybová hláška o neexistujícím souboru. Kernel nepodporuje AppArmor, nebo nebyl zkompilován s nastavením `CONFIG_SECURITY_APPARMOR=y`.
2. Je vypsáno písmeno "N", AppArmor nebyl při bootu povolen, nebo byl naopak zakázán (viz. výše).
3. Je vypsáno písmeno "Y", AppArmor je zaveden v jádře a připraven k použití.

Při dodržení tohoto postupu bude mít uživatel nabootovaný systém s podporou AppArmoru a připravený na jeho nastavení.

4.3 Konfigurace AppArmoru

Nastavení pravidel AppArmoru se provádí pomocí tzv. profilů, které se vždy vztahují k jednotlivým spustitelným souborům. Tyto spustitelné soubory jsou v profilech odkazovány pomocí cesty v souborovém systému. Tyto profily je nutné zavést do kernelu, který je uvede v účinnost ve chvíli spuštění daného programu. [20]

K práci s profily slouží sada user-space nástrojů, jejichž názvy většinou začínají "aa-" nebo "apparmor". Tyto nástroje často komunikují se samotným jaderným modulem AppArmoru skrz rozhraní zpřístupněná do uživatelského prostoru. Z hlediska uživatele pak tvoří nepostradatelnou součást AppArmoru. Spolu s těmito nástroji jsou většinou taktéž instalovány příslušné manuálové stránky dostupné přes příkaz `man`. [23]

4.3.1 Zavádění profilů do jádra

Jednotlivé profily jsou uložené na disku v textové podobě a následně předzpracovány do binární podoby vyžadované jádrem pomocí programu `apparmor_parse`. Tento program zároveň zajistí jejich zavedení do (paměti) jádra a nastavení požadovaného režimu vynucování (viz. dále). [23]

V produkčním prostředí je samozřejmě žádoucí profily zavést do jádra co nejdříve během spouštění systému. Typicky je to prováděno spouštěcím (init) skriptem, který automaticky při startu systému zavede všechny profily uložené v adresáři `/etc/apparmor.d/`. Je důležité, aby tento skript proběhl co nejdříve v pořadí, protože všechny procesy spuštěné ještě před zavedením příslušného profilu nejsou tímto profilem omezeny (tedy ani pokud je profil zaveden později, kdy proces již běží a je tedy vhodné takový proces restartovat). Tyto procesy bývají ve výpisech označovány jako "unconfined", opakem je označení "confined". Kritické jsou logicky pouze "unconfined" procesy, které mají zavedený profil. To je možné vyčíst například z přehledu stavu zavedených profilů a spuštěných procesů, který poskytuje program `aa-status`. Zajímavý je také výpis programu `aa-unconfined`, který obsahuje seznam unconfined procesů, které komunikují po síti.

Jedna z poměrně častých kritik AppArmoru se zaměřuje právě na to, že (např. narozdíl od SELinuxu) nevynucuje definovanou bezpečnostní politiku ihned po zavedení jádra, ale až po zavedení všech profilů patřičnými init skripty. Navíc načtení profilů je možné až po připojení souborového systému, který je obsahuje. Tudíž je velmi běžné, že se spouští množství procesů ještě předtím než AppArmor začne vůbec fungovat. Nicméně při korektním nastavení, které s touto vlastností počítá, to nelze považovat za příliš velkou nevýhodu. Stačí profily zavést co nejdříve před spuštěním všech služeb a ideálně i předtím než je nastavena síť nebo zobrazena přihlašovací obrazovka. Běžně to není problém, protože přesně takto je to ve většině linuxových distribucí také implementováno již od instalace.

4.3.2 Režimy vynucování profilů

Každý AppArmor profil zavedený do jádra může být v jednom ze dvou režimů. [20]

Režim "enforce" udává, že je profil vynucován a procesy, kterých se profil týká (v ideálním případě) nemají možnost tato pravidla porušit. Neúspěšné pokusy o přístup k prostředkům systému jsou zapsány do logu[22]. Do tohoto režimu přepneme profil příkazem `aa-enforce`. [20][23]

Druhý režim je tzv. "complain". V tomto režimu jsou pravidla sice zavedena, ale nejsou vynucována. Při porušení pravidel v "complain" režimu je tato událost pouze zalogována,

[22]Log je obecný název pro záznam systémových událostí. Výraz má pravděpodobně svůj původ v primitivním způsobu záznamu, kdy si lidé dělali zářezy do dřevěné klády (anglicky "log"), za účelem zaznamenat například počet uplynulých dní nebo ulovených zvířat.

což může pomoci při vytváření profilů. Z takového logu lze zjistit k jakým prostředkům daný proces přistupuje a lze je tedy přidat do profilu, který následně bude vynucován. Pokud se pak takový proces pokusí přistupovat k prostředkům, ke kterým během toho "učení" nepřistupoval, bude mu to zamítnuto a samozřejmě opět zalogováno. Tento režim lze aktivovat pro jednotlivé profily příkazem `aa-complain`. [20][23]

Zároveň s libovolným z těchto dvou režimů je také možné aktivovat tzv. režim "audit". Ten způsobí, že se do logu nezapisují jen neúspěšné pokusy o přístup k prostředkům, ale také úspěšné. To umožňuje sledovat, jak jsou přidělená práva využívána jednotlivými procesy až do důsledků. K zapínání a vypínání auditovacího režimu slouží program `aa-audit`. [20][23]

Navíc je samozřejmě možné profil nastavit, aby se nezaváděl vůbec. V takovém případě se nic neomezuje ani neloguje. Toho lze dosáhnout příkazem `aa-disable`. [23]

4.3.3 Textová reprezentace profilu

Již bylo zmíněno, že nastavení jednotlivých profilů apparmoru je uloženo v textové podobě v souboru na disku. Jako příklad lze uvést profil pro velmi běžný program `/bin/ping`, který v nějaké podobě obsahuje téměř každý operační systém s připojením do sítě. Tento program je ideální jako příklad, protože není příliš komplexní. Navíc je to jeden z nejrozšířenějších programů, které mají na Linuxu typicky nastavený příznak SUID. Proto i při spuštění neprivilegovaným uživatelem má tento program práva uživatele root, aby mohl po síti posílat speciální druh packetů. [22]

Vzhledem k běžným zvyklostem se textové soubory s profily jmenují tak, že např. profil pro program `/bin/ping` je uložen v souboru `/etc/apparmor.d/bin.ping`, jehož typický obsah je uveden zde:

```
#include <tunables/global>

/bin/ping flags=(complain) {
        #include <abstractions/base>
        #include <abstractions/consoles>
        #include <abstractions/nameservice>
```

```
        capability net_raw,
        capability setuid,

        network inet raw,

        /bin/ping mixr,
        /etc/modules.conf r,
}
```

Jeho obsah je možno vysvětlit takto:

- Řádek `/bin/ping flags=(complain)` uvozuje sekci pravidel pro spustitelný soubor `/bin/ping` (která následuje uzavřená ve složených závorkách) a navíc určuje, že tato pravidla mají být zavedena pouze v režimu "complain". V případě, že by šlo o režim "enforce", je možné sekci `flags=(...)` vynechat úplně.
- Řádky začínající `#include` způsobí načtení doplňujících nastavení ze souborů, jejichž název následuje. To je výhodné, protože tak lze na jednom místě spravovat pravidla společná pro více profilů bez potřeby tato nastavení po každé změně kopírovat do všech profilů (typicky v adresáři "abstractions"). Také tam mohou být definovány různé proměnné, které pak autor konkrétního profilu může využít, aniž by je musel ručně definovat (typicky v adresáři "tunables"). Například je možné prostě "nainkludovat" soubor, který zajistí, že daná aplikace bude smět přehrávat zvuk, aniž by bylo nutné u každé takové aplikace ručně vypisovat všechna konkrétní práva, která jsou k tomu potřeba (komunikace se zvukovou kartou, čtení a zápis systémového nastavení hlasitosti zvuku, spouštění zvukových daemonů, a mnoho dalšího).
- Řádky začínající `capability` umožní programu využít vyjmenované capability. V uvedeném případě capabilita `setuid` umožňuje skutečně nastavit práva roota a capabilita `net_raw` umožňuje pracovat se síťovou kartou v nízkoúrovňovém "raw" režimu, který umožní posílat a přijímat prakticky cokoliv po IP síti (včetně speciálních ICMP packetů vyžadovaných pingem).
- Řádka `network inet raw` opět povoluje komunikaci v IP sítích prostřednictvím síťového socketu typu raw. Není to ale na úrovni capabilit (které fungují přímo v jádře

nezávisle na AppArmoru), ale na úrovni omezování síťového provozu samotným apparmorem pomocí jeho interních mechanismů. Je nutné dodat, že tím není povolen žádný jiný druh síťové komunikace, který není tímto druhem komunikace vyžadován.

- Zbylé řádky umožňují programu čtení ("r") souboru `/etc/modules.conf` a čtení ("r"), spuštění ("x") souboru `/bin/ping` s tím, že nově spuštěný proces zdědí ("i") definovaná omezení a bude možné ho namapovat do paměti přes `mmap()` ("m").

- Vše ostatní je zakázáno, protože to není výslovně povoleno.

Veškeré možnosti nastavení profilu, které AppArmor umožňuje, jsou nad rámec této práce. Je jich mnoho a k plnému pochopení jejich funkcionality by nestačil pouhý výčet. Kromě toho se AppArmor i samotný Linux stále vyvíjí. Proto se doporučuje, obrátit se na dokumentaci (např. [20]) nejnovější verze AA. [20][22]

Pro zajímavost uvádím, že AppArmor umožňuje i používat různé pokročilé funkce jako například použití různých profilů jedním programem v závislosti na tom, jaký program ho spustil, nebo dokonce nadefinování více podprofilů pro jeden program. Program pak může aktivně spolupracovat a podprofily si vybírat například napříč thready podle toho, co který thread zrovna obstarává pomocí knihovny `libapparmor`. Tak je možné, že program sice má právo používat některé prostředky systému, ale dobrovolně se jich vzdá ve chvílích, kdy provádí některé operace rizikové z bezpečnostního hlediska (typicky zpracování dat z nedůvěryhodného vstupu). [20]

4.3.4 Vytváření profilu

Kromě toho, že je možné AppArmor profil napsat ručně pomocí textového editoru, je také možné ho nechat automaticky vygenerovat. To se provádí příkazem `aa-genprof`, kterému se jako parametr předá jméno programu, který má být AppArmorem "obrněn"[23]. Ten potom zvolenému programu vytvoří (téměř) prázdný profil pomocí `aa-autodep` a zavede ho do jádra v "complain" režimu. Program `aa-autodep` zjistí jaké dynamické knihovny jsou k běhu programu potřeba pomocí programu `ldd` a přidá do profilu právo je načíst. Pokud program není spustitelný nativně, ale jde o skript, je do profilu přidáno právo spustit patřičný interpreter uvedený v hlavičce skriptu. Protože v AppArmoru platí, že co

[23]"AppArmor" přeci jen ve volném překladu znamená "brnění pro aplikace".

není povoleno, to je zakázáno, je tedy zakázáno naprosto vše ostatní, co AppArmor umí zakázat.

Nyní je vhodný čas, daný program skutečně spustit a pracovat s ním, tak jak je obvyklé. Toto nastavení logicky vyústí v to, že si AppArmor začne "stěžovat", že daný program porušuje svoje (zatím velmi nízké) pravomoce a tyto stížnosti zapíše do logu. Program `aa-genprof`, který stále ještě běží si tyto stížnosti v logu přečte pomocí programu `aa-logprof` a umožní je přetransformovat do podoby pravidel, která programu přidají potřebné pravomoce. Stejně tak lze v budoucnu pomocí programu `aa-logprof` projít log a doladit s jeho pomocí v profilu případné problémy, které budou zjištěny při pozdějším provozu programu.

Tyto postupy však vyžadují, aby správce jednotlivá práva vyžadovaná programem poučeně zhodnotil a následně se zodpovědně rozhodl, zda je programu přidělí, nebo ne. Naštěstí odmítnutí některých rizikových pravomocí, které se program snaží využít, se často obejde bez úplného pádu nebo znefunkčnění programu jako celku. Pokud však takový krajní případ nastane, pak je jen na správci, jak se rozhodne situaci řešit.

Poté, co je použit tento částečně automatizovaný přístup k vytvoření profilů, může být vhodné se nad vzniklým profilem zamyslet a zoptimalizovat ho pomocí ruční úpravy textovým editorem. Například pokud si program vyžádá přístup k jedné konkrétní zvukové kartě, dá se předpokládat, že by mohlo být žádoucí dát mu přístup ke všem zvukovým kartám jako k celku, ideálně pomocí nainkludování existujícího souboru "abstrakce" (viz. výše), atd... Stejně tak pokud program přistupuje k nějakému souboru, je pravděpodobné, že bude mít někdy potřebu přistupovat i k dalším souborům ve stejném adresáři. Zmíněné programy pro automatické generování profilů nám často takové optimalizace nabídnou samy. Při tom pomáhá program `aa-cleanprof`, který umí odstranit příliš konkrétní pravidla, která definují práva, která jsou již podchycena některým více obecným pravidlem. Taktéž může být nápomocný program `aa-mergeprof`, který umožňuje sloučit dva profily do jednoho. Nelze však vyloučit, že i poté budou žádoucí drobné ruční úpravy profilu. I zde opět platí, že pokud deset pravidel můžeme nahradit jedním univerzálnějším, znamená to přínos nejen pro výkon systému při zpracování těchto pravidel, ale i pro bezpečnost samotnou, protože kratší a tedy přehlednější profil můžeme snáze auditovat a odhalit v něm případné nedostatky.

5 AppArmor (případová studie)

V této části budu demonstrovat praktické využití AppArmoru na modelovém případu a proto v ní z velké části vycházím téměř výlučně z empirických poznatků o chování systému, který jsem v předchozí části popsal po stránce teoretické.

5.1 Popis modelového problému

Existuje server s OS Linux (konkrétně budu proces demonstrovat na distribuci Ubuntu Linux 14.04.1 LTS[24]) a webovým démonem Apache, na kterém běží interaktivní služba poskytovaná nedůvěryhodným skriptem v jazyce PHP. Skript považujeme za nedůvěryhodný, protože neznáme přesný záměr jeho autora a nevíme, zda se v jeho návrhu nedopustil bezpečnostní chyby. Tento program je také příliš rozsáhlý, než aby pro nás bylo výhodné si ho celý přečíst a manuálně zkontrolovat. Kromě této služby na serveru mají další uživatelé uloženy soubory s citlivými daty, u kterých si nemůžeme být jisti, že se při užívání nedopustí špatného nastavení přístupových práv[25]. Rádi bychom tedy zajistili, že ani při fatální chybě v nastavení uživatelských oprávnění nebude možné tato data získat skrze zmíněnou webovou službu.

Pro modelové účely jsem na serveru vytvořil uživatele "uzivatel" a jeho jménem jsem v jeho domácím adresáři vytvořil soubor `/home/uzivatel/tajemstvi.txt`, jehož obsah se budeme snažit získat. Aby šlo o model skutečně lehkovážného uživatele, tak jsem k souboru nastavil "plná" přístupová práva[26].

Kromě toho jsem do kořenového adresáře webserveru dal modelový skript `/var/www/test.php`, který bude naše zabezpečení pokoušet. Jeho zdrojový kód je prostý:

[24] Ubuntu má AppArmor funkční ihned po instalaci, patří mezi rozšířenější distribuce (i když ne nutně na serverech) a navíc jsem jeden takový server již před krátkým časem nainstaloval a proto se jeho využití při psaní této práce přímo samo nabízelo.

[25] Je nutné poznamenat, že poněkud tradičněji s k těmto účelům používalo přímo nastavení interpreteru PHP (tzv. safe mode, konkrétně direktiva `open_basedir`), které mělo za účel omezit skriptu přístup ke konkrétním adresářům. Praxe ukázala tento přístup jako nevyhovující vzhledem k velkému množství možností jak ho obejít. Dokonce i samotní vývojáři PHP od této možnosti postupně upouští a má být v budoucnu zrušena úplně. Proto bylo zvoleno zabezpečení přímo na úrovni jádra systému, které nejen že je nesrovnatelně účinnější, ale také je možné ho aplikovat v množství jiných případů, na aplikace, které nejsou psány v PHP, ale i úplně mimo kontext vzorového problému.

[26] V osmičkovém zápisu 777, tedy: s obsahem souboru může manipulovat libovolný uživatel systému

```
<?php
echo(file_get_contents('/home/uzivatel/tajemstvi.txt'));
system('cat␣/home/uzivatel/tajemstvi.txt␣2>&1');
```

První řádek je standartní uvození PHP kódu, druhý řádek se pokouší informace získat pomocí interních funkcí PHP a pro případ, že by to nestačilo, jsem ještě přidal řádek třetí, který se pokusí soubor přečíst pomocí externího programu `cat`. To je program, který mimo jiné umožňuje výpis obsahu zvoleného souboru. Volání tohoto programu jsem ještě doplnil o řetězec "`2>&1`", který zajistí, že se v případě neúspěchu do PHP skriptu předá i výpis případného chybového hlášení programu `cat`.

Nyní není nic snažšího, než se pokusit k webu připojit. Toho docílíme použitím webového prohlížeče, nebo například i příkazem `curl http://localhost/test.php`[27]. V mém případě jsem obdržel následující výpis:

```
Warning: file_get_contents(): open_basedir restriction in effect.
File(/home/uzivatel/tajemstvi.txt) is not within the allowed path(s):
(/var/www/) in /var/www/test.php on line 2

Warning: file_get_contents(/home/uzivatel/tajemstvi.txt):
failed to open stream:
Operation not permitted in /var/www/test.php on line 2

Odpoved je 42!
```

Jak je vidět, tak PHP samotné se skriptu pokoušelo zabránit díky nastavení `open_basedir`, ale ve výsledku to nebylo nic platné, protože skript potom spustil externí program, na který se již toto omezení nevztahuje. Druhý pokus byl tedy úspěšný a správně odhalil, že obsah souboru `tajemstvi.txt` je "`Odpoved je 42!`". V další části se pokusím mu v tom zabránit nadobro.

5.2 Modul AppArmor v praxi

Víme, že zákeřný skript, který vyzradil tajemství našeho uživatele, byl spuštěn prostřednictvím služby Apache. Pokusíme se ji tedy lépe izolovat od zbytku systému a tedy i od

[27] `curl` je jednoduchý příkaz, který stáhne a vypíše obsah webové stránky zadané pomocí její URL, je možné ho tedy použít k simulaci otevření stránky v prohlížeči.

střeženého tajemství. V první řadě potřebujeme vědět, který spustitelný soubor je za službu odpovědný. Toho docílíme například příkazem `ps aux | grep -i apache`. Z jeho výpisu se dozvíme, že démon běží pod uživatelem `www-data` a jde o spustitelný soubor `/usr/sbin/apache2`. Protože je ve standartní cestě, bude nám ve většině případů stačit psát jen `‘‘apache2’’`. Spustíme tedy příkaz `aa-genprof apache2`.

Nyní byl do kernelu zaveden téměř prázdný AppArmor profil pro binárku `/usr/sbin/apache2`, prozatím v režimu “complain”. Nyní je příhodný čas na to, abychom Apache restartovali např. příkazem `service apache2 restart`. Budeme to ale muset udělat v novém okně, protože příkaz `aa-genprof` stále ještě běží. Proces restartu Apache nejdříve ukončí starý již běžící proces a poté spustí nový. Tentokrát již v “confined” režimu a tedy pod dohledem AppArmoru. To si také lze ověřit příkazem `aa-status`, v jehož výpisu najdeme (mimo jiné) řádky podobné těmto:

```
6 processes are in complain mode.
/usr/sbin/apache2 (4644)
/usr/sbin/apache2 (4648)
/usr/sbin/apache2 (4649)
...
```

Nyní je čas Apache trochu “procvičit”, aby se pokusil přistupovat k různým prostředkům a bylo to zaznamenáno do logu. Velmi pomůže Apache zrestartovat, což již bylo učiněno. Dále můžeme zopakovat například příkaz `curl http://localhost/test.php`, který na serveru opět spustí skript `test.php`, který si přejeme spolu Apachem profilovat. Když se vrátíme zpět do terminálu, kde stále ještě běží příkaz `aa-genprof`, zjistíme, že se nabízí možnost automaticky prohledat log a vygenerovat profily z toho, kam již Apache přistoupil. Tato možnost se zvolí klávesou “S”, ale je v ní drobný háček. Okamžitě po stisku této klávesy totiž na uvedeném systému celý program havaruje s chybovým hlášením, které odhalí, že je napsán v jazyce Python.

Není potřeba se tím znepokojovat, protože stejného průvodce vytvářením profilů můžeme spustit i pomocí příkazu `aa-logprof`. To také uděláme. Za pár okamžiků se již první část logu zpracuje a my budeme postupně informováni o všech aktivitách programu Apache a u každé dostaneme na výběr, jestli ji chceme v budoucnu povolit nebo ne.

Například budeme informováni o tom, že Apache přistoupil k souboru `/var/www/test.php` a bude nám zobrazena následující textová nabídka:

```
(A)llow, (D)eny, (I)gnore, (N)ew, (G)lob last piece, (Q)uit
```

Písmenkem "G" můžeme pravidlo zevšeobecnit z `/var/www/test.php` například na `/var/www/*`, což se nám bude jistě hodit, pokud budeme provozovat skriptů více. Poté přístup k této cestě povolíme tlačítkem "A". Podobně postupujeme i u dalších pravidel. Cílem je momentálně povolit vše, co se netýká adresáře `/home/`, ve kterém se hluboko skrývá naše `tajemstvi.txt`. Kromě přístupů k souborům AppArmor taktéž ohlásí další akce, jako je využívání[28] capabilit, spouštění dalších programů, atd...

V případě, že sledovaná aplikace spouští další program, nabídne `aa-logprof` několik možností. Asi nejzajímavější jsou "inherit", "profile", "child" a "unconfined". V případě "inherit" se s nově spuštěným programem nakládá tak, jako by byl součástí programu, který ho spustil a platí pro něj i stejné pravomoce a omezení. Pokud zvolíme "profile", bude pro daný program zvolen jeho vlastní profil. Podobnou možnost nabízí volba "child", která umožní nadefinovat speciální profil pro případ, kdy je program spouštěn tímto konkrétním rodičovským programem. Jde tedy asi o nejflexibilnější volbu, ale její použití může někdy být zbytečně složité. Poslední volba "unconfined" samozřejmě znamená, že nově spuštěný program nebude omezen nijak. To je v drtivé většině případů velmi nebezpečná a naprosto nemyslitelná volba. Také bych podotknul, že userspace pomůcky AppArmoru jsou o poznání více problematické, než samotný jaderný modul AppArmoru. Například verze `aa-logprof` nainstalovaná na mnou testovaném systému padala vždy, když byla v logu nalezena informace o spuštění dalšího programu. I proto je dobré se naučit profily editovat ručně, bez těchto pomocných programů.

Kromě toho `aa-logprof` po celou dobu informuje, pokud je některá z pravomocí vyžadovaných confined programem již obsažená v některé z abstrakcí a odpovídající abstrakce nám nabízí k přidání. To taktéž usnadňuje práci. Pokud se tedy program pokusí přistoupit například k souboru `/etc/php/apache2/php.ini`, může nám být automaticky nabídnuta abstrakce, která obsahuje vše potřebné k provozu PHP (pokud taková abstrakce ovšem v našem systému existuje).

Poté, co pomocí `aa-logprof` projdeme všechny nové položky v logu, se program zeptá, zda chceme změny profilu uložit, nebo ho ukončit bez uložení. Pravděpodobně bude nutné po každém uložení znovu restartovat Apache a celou proceduru "učení" opakovat tak dlouho,

[28] AppArmor není určen k tomu, aby programu capability přiděloval. Pouze monitoruje a omezuje jejich používání u programů, které je již mají.

dokud nepřestanou v logu přibývat nová upozornění. V mém případě jsem celý proces absolvoval asi třikrát po sobě, než začal Apache dělat vše, co jsem potřeboval a aniž by si AppArmor na něco stěžoval. V průběhu toho procesu mi bylo nabídnuto do profilu i nainkludování některých abstrakcí. Z těch jsem si po jejich prozkoumání zvolil pouze abstrakce `apache2-common` a `php5`, které obsahují oprávnění k souborům s nastavením, knihovnám a dalším základním prostředkům, které Apache a PHP vyžadují.

Nyní nastal čas na přepnutí profilu do "enforce" režimu. To provedeme příkazem `aa-enforce apache2`. Okamžitě po provedení tohoto příkazu začne být Apache (i běžící) svazován svým AppArmor profilem. To se projeví mimo jiné tak, že při opětovném použití příkazu `curl http://localhost/test.php` již neuvidíme obsah souboru `tajemstvi.txt`. To je dáno jednak tím, že k danému souboru není v profilu povolen přístup. Navíc není ani povoleno spouštět externí programy, jako je `cat`.

Může být žádoucí ještě několikrát zkontrolovat log a doplnit profil o případná další pravidla. Nejsem si jistý, čím je to způsobeno, ale často se v režimu enforce projeví potřeba některých pravomocí, které režim complain neodhalil. Všeobecně platí, že je dobré log pravidelně kontrolovat a aktualizovat na jeho základě profil. Nelze to však dělat bezmyšlenkovitě. Nejen, že tak odhalíme nedostatky v profilech, které by mohly bránit našemu programu v korektním fungování, ale také nám to pomůže odhalit případné pokusy o průnik. Pokud například AppArmor do logu zaznamená, že se hlídaný program snaží provádět operaci, ke které nikdy nebyl nevržený, můžeme si být téměř jisti, že jde o bezpečnostní incident.

5.3 Hotový profil

Po provedení výše popsaného postupu by měl vzniknout profil v souboru `/etc/apparmor.d/usr.sbin.apache2`, který je možné zavést příkazem `aa-enforce apache2`. Jeho obsah by měl být přinejmenším podobný[29] následujícímu:

[29] Mám zkušenost, že automatické generování profilů často nevytvoří pokaždé úplně stejný profil i když se podruhé snažíme dodržet stejný postup. Zřejmě asi do hry vstupuje velké množství různých okolností. Na to jsem narazil, když jsem se snažil vyzkoušený postup zopakovat a zdokumentovat ho v této práci. Proto byl také uvedený kód profilu mírně upraven pomocí textového editoru. Přesto byl však plně otestován.

```
# Last Modified: Tue Nov 25 20:33:01 2014
#include <tunables/global>

/usr/sbin/apache2 {
        #include <abstractions/apache2-common>
        #include <abstractions/base>
        #include <abstractions/mysql>
        #include <abstractions/php5>

        capability dac_override,
        capability kill,
        capability setgid,
        capability setuid,

        deny /bin/dash x,
        deny /bin/cat x,
        deny /home/** r,

        /etc/apache2/** r,
        /etc/mime.types r,
        /run/apache2/* rw,
        /run/lock/apache2/* rw,
        /tmp/* rwk,
        /usr/lib{,32,64}/** mr,
        /var/log/apache2/* w,
        /var/www/* r,

        ^DEFAULT_URI flags=(complain) {
        }

        ^HANDLING_UNTRUSTED_INPUT flags=(complain) {
        }
}
```

Oproti profilu, který jsem si rozebíral v předchozí části práce, je v tomto několik dalších

výrazů. Například jsou zde pravidla uvozená klíčovým slovem “deny”. Tyto znamenají, že je daná operace zakázaná. Ve skutečnosti by nebylo třeba tato pravidla uvádět, protože co není AppArmorem povoleno, to je zakázáno. Pravidla “deny” slouží jen k tomu, aby bylo jasně dáno, že o jejich povolení nemáme zájem a jejich hlavní účel je, že pokusy o jejich využití jsou tiše zamítnuty a nezaplňují log. To také znamená, že je nemusíme opakovaně zamítat při průběhu programu `aa-logprof`, ale stačí je zamítnout jednou.

Již jsem zmiňoval, že pokud je u cesty k souboru písmenko “x”, definuje se jím právo soubor spustit. Kromě toho je vhodné před “x” uvést ještě jedno z písmenek “p”,”c”,”u” nebo ”i”. Tato písmenka odpovídají možnostem “profile”, “child”,”unconfined” a “inherit”, které jsem popsal výše. Případně lze tato písmenka napsat jako velká. Potom se ještě před spuštěním daného programu vymažou všechny proměnné prostředí[30]. Je potřeba dát pozor na to, abychom tato rozšiřující nastavení neuváděli v pravidlech “deny”. Tam musí být vždy jen samotné “x”, jinak zavedení profilu selže.

Dále je v profilu pravidlo vztahující se k `/etc/apache2/**`. Poučenější uživatel ví, že hvězdička ve výrazu nahrazuje libovolný soubor nebo adresář. AppArmor ale navíc ještě zavádí hvězdičky dvě, které znamenají, že se pravidla mají vztahovat i rekurzivně na podadresáře. Pokud ale uvedeme hvězdičku jen jednu, tak se pravidla vztahují pouze na položky v tomto adresáři a na ty v podadresářích již ne. To umožní pravidla definovat o něco přesněji.

Na konci profilu jsou ještě dvě pojmenované, ale prázdné sekce uzavřené ve složených závorkách. Jde o subprofily typu “hat”. Pod tyto profily může program dobrovolně přecházet pomocí knihovny `libapparmor`. V našem případě jsou prázdné, protože pod nimi Apache při profilování neprováděl žádné zajímavé akce. Nicméně v profilu jsou uvedeny, protože je program `aa-autodep` automaticky vyčetl z binárky `apache2`. Též je možno si všimnout, že i když profil jako celek je v enforce režimu, tak podprofily jsou stále v režimu complain a čekají na otestování a explicitní vypntí režimu complain. Podobným způsobem se dají definovat také subprofily pro programy spouštěné s nastavením “child”. V této práci se

[30]“Environment variables” jsou proměnné v paměti, do kterých mohou programy ukládat textové řetězce, které jsou poté přístupné jak jim samotným, tak dalším programům, které jsou jimy spuštěny jako potomci. Na Linuxu se typicky používají k uložení desítek informací. Například se do nich ukládá jazykové nastavení, takže jakmile se uživatel přihlásí do systému a nastaví si např. češtinu, jsou automaticky všechny programy, které si spustí z této relace přihlášení taktéž nastaveny do češtiny. Důvod, proč AppArmor umožňuje jejich “sanitizaci” je ten, že představují další možnost, jak programům podstrkávat potenciálně škodlivá vstupní data.

subprofily nepracuji, ale považuji za vhodné se o jejich existenci zmínit, protože se s nimi v práci okrajově setkáváme právě díky `aa-autodep`.

5.4 Ověření funkčnosti konfigurace

Nyní zabezpečení skriptu ještě snížím odstraněním nastavení `open_basedir` v PHP. Po opakovaném pokusu o přístup skript vypíše následující:

```
Warning: file_get_contents(/home/uzivatel/tajemstvi.txt):
failed to open stream: Permission denied in /var/www/test.php on line 2
```

To, že PHP místo "`Operation not permitted`" nyní hlásí "`Permission denied`", je matným znamením toho, že k zablokování přístupu nedošlo na úrovni PHP, ale na úrovni operačního systému. Stejnou chybu by totiž PHP vypsalo i v případě, že by uživatel práva k souboru nastavil korektně.

Je důležité si všimnout i toho, že pokus o přečtení souboru pomocí externího programu nevypíše žádnou chybu. To je dáno tím, že volání `system()` v jazyce PHP zřejmě implicitně chyby nevypisuje a to jak správně ošetřovat v PHP chyby není pro tuto práci zásadní. Budeme tedy předpokládat, že pokud se nevypíše nic, bylo zabezpečnění úspěšné. Tuto hypotézu lze ověřit přečtením systémového logu, kde najdeme řádky podobné těmto:

```
[467668.042615] type=1400 audit(1416934306.887:394181):
apparmor="DENIED" operation="open" profile="/usr/sbin/apache2"
name="/home/uzivatel/tajemstvi.txt" pid=13997 comm="apache2"
requested_mask="r" denied_mask="r" fsuid=33 ouid=1023

[467668.043972] type=1400 audit(1416934306.887:394182):
apparmor="DENIED" operation="exec" profile="/usr/sbin/apache2"
name="/bin/dash" pid=29871 comm="apache2"
requested_mask="x" denied_mask="x" fsuid=33 ouid=0
```

Z toho vyrozumíme, že se Apache (skrz PHP) nejdříve pokusil přečíst ("r") soubor `tajemstvi.txt` a potom se pokusil spustit ("x") program `/bin/dash`. To ovšem může překvapit, jelikož jsme od PHP skriptu chtěli spustit pouze program `cat`, nikoliv `dash`. Lze již vytušit k čemu došlo. Volání `system()` nespouští program `cat` přímo, ale pomocí výchozího interpreteru

příkazů[31], který umožňuje kromě obyčejného spouštění programů i některé pokročilejší příkazy[32].

Abych si ještě jednou vyzkoušel, že AppArmor skutečně funguje, zkusil jsem v profilu ručně nahradit následující řádky:

```
deny /bin/dash x,
deny /bin/cat x,
deny /home/** r,
```

tímto kódem:

```
/bin/dash ix,
/bin/cat ix,
/home/** r,
```

Ten umožní přístup k celému adresáři `/home/` i spuštění programů `dash` a `cat`. Po uložení profilu ho lze znovu zavvést do kernelu příkazem `aa-enforce apache2`. Ihned lze vyzkoušet co se stane při přístupu na webovou stránku příkazem
`curl http://localhost/test.php`
Uvidíme, že `tajemstvi.txt` je vyzrazeno hned dvakrát:

```
Odpoved je 42!
Odpoved je 42!
```

Tím bylo prokázáno, že AppArmor má skutečně vliv na to, zda se problematická webová aplikace dostane k citlivým datům a dokáže tomu efektivně zabránit za předpokladu, že je správně nastaven a není kompromitován samotný kernel.

Změny v textu profilu vrátíme zpět a pomocí `aa-enforce apache2` tuto otestovanou verzi zavedeme do jádra.

[31] Typicky `/bin/sh`, v případě Ubuntu je nastavený, aby odkazoval na `/bin/dash`

[32] V našem testovacím skriptu toho využívám k přesměrování chybové hlášky (viz výše). Chyba při spouštění `/bin/dash` se tedy nezobrazí, ale chyby, ke kterým dojde v programu `/bin/cat` při přístupu k `tajemstvi.txt`, by se již zobrazily

6 Zhodnocení výsledků

S výsledky práce jsem spokojen, protože se mi podařilo předvést, že modul AppArmor je vhodný pro praktické použití na Linuxovém serveru s typickým softwarovým vybavením.

7 Závěr

Cílem práce bylo seznámit čtenáře s problematikou zabezpečení Linuxových systémů a blíže rozebrat bezpečnostní modul AppArmor. Dále jsem chtěl přiblížit čtenáři jeho fungování v teoretické rovině a poté předvést jeho nasazení v praxi na modelovém případu, včetně ověření a zhodnocení funkcionality.

V práci jsem stručně otevřel problematiku Linuxového prostředí, popsal základní aspekty zabezpečení, které by měl každý začínající správce Linuxového serveru znát. Důraz byl kladen na nejčastěji se vyskytující bezpečnostní chyby, možnosti jejich zneužití a prostředky jejich řešení. Dále byla v práci popsána dostupná rozšíření tohoto základu, která poskytují pokročilejší možnosti zabezpečení. Poté byl v práci podrobněji rozebrán modul AppArmor, který mezi tato rozšíření spadá. V práci byly popsány jak základní principy fungování AppArmoru, tak i ukázka úspěšné aplikace AppArmoru na modelový případ. Čtenář s pouze základní předchozí zkušeností se správou Linuxového serveru by tedy po přečtení práce měl mít přehled o problematice AppArmoru dostatečný k tomu, aby ho byl schopen sám nasadit v testovacím provozu a za použití aktuální dokumentace postupně začít AppArmor nastavovat pro svoje konkrétní potřeby.

Tato práce byla zpracována na základě analýzy odborné literatury, technické dokumentace a praktických zkušeností s nasazením popisovaných technologií.

8 Seznam použitých zdrojů

8.1 Seznam literatury

[1] BERLICH, R. ALL YOU NEED TO KNOW ABOUT... The early history of Linux: Part 2, Re: distribution [online]. 2001. [cit. 20.11.2014].

<http://oldlinux.org/Linux.old/docs/lu9-All_you_need_to_know_about
-The_early_history_of_Linux_part_2.pdf>.

[2] SCHROEDER, M, SALTZER, J. A Hardware Architecture for Implementing Protection Rings [online]. 1972. [cit. 20.11.2014].
<http://www.multicians.org/protection.html>

[3] Kolektiv autorů. Linux Security HOWTO: Files and File system Security [online]. 2004. [cit. 20.11.2014].
<http://www.tldp.org/HOWTO/Security-HOWTO/file-security.html>

[4] The Linux Foundation, mfillpot. Understanding Linux File Permissions [online]. 2010. [cit. 20.11.2014].
<http://www.linux.com/learn/tutorials/309527-understanding-linux-file-permissions>

[5] KERRISK, M. CAP_SYS_ADMIN: the new root [online]. 2012. [cit. 20.11.2014].
<http://lwn.net/Articles/486306/>

[6] Kolektiv autorů. Linux Security HOWTO: Physical Security [online]. 2004. [cit. 20.11.2014].
<http://www.tldp.org/HOWTO/Security-HOWTO/physical-security.html>

[7] Kolektiv autorů. TLDP: The File system [online]. 2004. [cit. 20.11.2014].
<http://tldp.org/LDP/tlk/fs/filesystem.html>

[8] Kolektiv autorů. Security Quick-Start HOWTO for Linux: Servers, Ports, and Packets [online]. 2004. [cit. 20.11.2014].
<http://www.tldp.org/HOWTO/Security-Quickstart-HOWTO/appendix.html>

[9] LEVY, E. Smashing the Stack for Fun and Profit [online]. 1996. [cit. 20.11.2014].
<http://phrack.org/issues/49/14.html>

[10] COWAN, C, WAGLE, P, Pu, C, BEATTIE, S, WALPOLE, J. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade [online]. 2000. [cit. 20.11.2014].
<http://tmp-www.cpe.ku.ac.th/~mcs/courses/
2005_02/214573/papers/buffer_overflows.pdf>

[11] DEWRY, W, LUTOMIRSKI, A. Dokumentace zdrojového kódu Linuxu: SECure COMPuting with filters [online]. 2012. [cit. 20.11.2014].
<https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt>

[12] ANDREASSON, O. Iptables Tutorial 1.1.19 [online]. 2003. [cit. 20.11.2014]. <http://www.linuxhowtos.org/Security/iptables.htm>

[13] PAVLÍČEK, M. Chroot prostředí - I [online]. 2003. [cit. 20.11.2014]. <http://www.abclinuxu.cz/clanky/bezpecnost/chroot-prostredi-i>

[14] WATZKE, D. Unixové nástroje – 18 (nice, renice, umask) [online]. 2010. [cit. 20.11.2014]. <http://www.abclinuxu.cz/clanky/unixove-nastroje-18-nice-renice-umask>

[15] MENAGE, P, JACKSON, P, LAMETER, Ch. Dokumentace zdrojového kódu Linuxu: CGROUPS [online]. 2006. [cit. 20.11.2014]. <https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt>

[16] SIMES. How to break out of a chroot() jail [online]. 2002. [cit. 20.11.2014]. <http://www.bpfh.net/simes/computing/chroot-break.html>

[17] WRIGHT, C, COWAN, C, MORRIS, J, SMALLEY, S, KROAH-HARTMAN, G, WireX Communications, Inc., Intercode Pty, Ltd, NAI Labs, Network Associates, Inc., IBM Linux Technology Center. Linux Security Modules: General Security Support for the Linux Kernel [online]. 2002. [cit. 20.11.2014]. <http://www.usenix.org/event/sec02/wright.html>

[18] ŠINDELÁŘ, J. Novell uvolňuje bezpečnostní řešení AppArmor pro Linux jako open-source [online]. 2006. [cit. 20.11.2014]. <http://www.zive.cz/clanky/novell-uvolnuje-bezpecnostni-reseni-apparmor-pro-linux-jako-open-source/sc-3-a-128537/default.aspx>

[19] CORBET, J. How patches get into the mainline [online]. 2009. [cit. 20.11.2014]. <http://lwn.net/Articles/318699/>

[20] http://wiki.apparmor.net/index.php/AppArmor_Core_Policy_Reference

[21] Kolektiv autorů, INSANITYBIT. Compile And Patch Your Own Secure Linux Kernel With PaX and Grsecurity [online]. 2012. [cit. 20.11.2014]. <http://www.insanitybit.com/2012/05/31/compile-and-patch-your-own-secure-linux-kernel-with-pax-and-grsecurity/>

[22] Kolektiv autorů, Canonical, Ltd. Ubuntu documentation: Security/AppArmor [online]. 2014. [cit. 20.11.2014]. <https://help.ubuntu.com/lts/serverguide/apparmor.html>

[23] Kolektiv autorů, Canonical, Ltd. Ubuntu documentation: Community/AppArmor [online]. 2014. [cit. 20.11.2014].
<https://help.ubuntu.com/community/AppArmor>

9 Přílohy

9.1 Ukázka použití seccompu v C

Tento program jsem napsal v rámci studia seccompu pro názornou demonstraci jeho využití při psaní programů v jazyce C. Program nejdříve spustí externí program echo, což se projeví výpisem textu "before", poté je aktivován seccomp a je znovu zopakován pokus o spuštění programu echo. V případě, že seccomp funguje, dojde k násilnému ukončení celého programu kernelem. Pokud by seccomp selhal, echo se spustí a vypíše text "after".

```
/*
 * seccomp.c (Tomas Mudrunka 2014)
 *
 * This demonstrates how to use SECCOMP_MODE_STRICT to sandbox code on Linux.
 * You need kernel compiled with CONFIG_SECCOMP=y.
 * This prohibits everything except read(2), write(2), _exit(2), and sigreturn(2).
 * Trying to use other syscalls will result in SIGKILL.
 * If you need to enable more syscalls you can use SECCOMP_MODE_FILTER instead.
 * See man 2 prctl for more...
 *
 */

#include <stdlib.h>
#include <string.h>
#include <sys/prctl.h>
#include <linux/seccomp.h>
#include <sys/syscall.h>

#define DISPLAY(msg) (syscall( SYS_write, 2, msg, strlen(msg) ))
#define exit(status) { syscall( SYS_exit, status ); abort(); }

int main() {
        system("echo␣before");

        if(prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT) == 0)
                DISPLAY("SECCOMP␣Enabled!\n");
        else
                DISPLAY("SECCOMP␣Fail!\n");
        //fflush(NULL);

        system("echo␣after");
        exit(0);
}
```