Bachelor Informatica

UvA Universiteit van Amsterdam

Resource management in een Linux jailing systeem

Jos Kamphorst

23 augustus 2008

Supervisor(s): Guido van 't Noordende (UvA)

Signed: Guido van 't Noordende (UvA)

Samenvatting

Het project wat hier beschreven wordt gaat over de implementatie van een resource manager in een Linux jailing systeem. Het gebruikte jailing systeem is beschreven in "A SECURE JAILING SYSTEM FOR CONFINING UNTRUSTED APPLICATIONS" [3]. In dit werk wordt het CPU resource gebruik gelimiteerd door de uitvoering van system calls selectief te vertragen.

De resource manager houdt met behulp van *timeslots* bij of een system call direct moet worden uitgevoerd of vertraagd. Binnen deze *timeslots* is het toegestaan om, als prisoner proces, system calls uit te voeren. Deze resource manager implementatie sluit sommige, op CPU gebaseerde, denial-of-service attacks uit.

De resource manager maakt gebruik van een uitgebreide datastructuur die makkelijk naar een bestand geschreven kan worden. Deze datastructuur bevat alle tijden van getime'de system calls. Deze tijden worden gebruikt voor controle of de system call nog in een *timeslot* past. De datastructuur is gemaakt met het oog op de toekomst.

Inhoudsopgave

HOOFDSTUK 1

Inleiding in het jailing systeem

Bij het downloaden van een programma kan het soms zijn dat er kwaadwillende code tussen zit. Bij het uitvoeren van zo'n programma is het risico groot dat er schade wordt aangericht op het systeem. Bijvoorbeeld het verwijderen van bestanden en het lezen en/of versturen van gevoelige informatie. Niet alleen code van het internet is niet altijd te vertrouwen. Er zijn ook programma's die normaal gesproken als betrouwbaar bestempeld worden. Deze programma's kunnen een bug bevatten waarbij het mogelijk is, via een exploit, op het systeem te komen en een aanval uit te voeren.

Een jailing systeem zorgt voor een veilige omgeving, ook wel de jail genoemd. Binnen deze jail kan de onveilige code uitgevoerd worden. Als onveilige code, bijvoorbeeld `rm -r /etc`[1], aangeroepen wordt, dan zal dit binnen de jail omgeving niet uitgevoerd worden.

Er zijn verschillende jail omgevingen. Zo is er een *chroot-jail* omgeving wat gebruik maakt van `chroot`[2]. `chroot` staat voor "change root directory". Het zorgt ervoor dat een normale directory, bijvoorbeeld: `/home/jail-dir` fungeert als de root directory `/`. Het resultaat is een eigen directory omgeving voor een proces. Het zorgt er bijvoorbeeld voor dat `/usr/bin` niet meer benaderbaar is voor een proces binnen de jail, tenzij de directory `/home/jail-dir/usr/bin` bestaat. Het nadeel van deze benadering is dat binnen deze omgeving alles gedaan kan worden wat de gebruiker van het proces kan doen. Het proces wordt dus niet actief bijgestuurd. Als het proces genoeg rechten heeft kan deze weer uit de `chroot` omgeving breken[3]. Nog een nadeel is het aanroepen van `chroot`. Er wordt namelijk verwacht dat dit gebeurd met "root" rechten. Een normale gebruiker kan dus nooit een jail opstarten.

Een andere jail omgeving is een *system call interceptie gebaseerde jail*, hierna *jailer* genoemd. Deze *jailer* beveiligt de omgeving met het onderscheppen van *system calls* in user-mode. System calls[4] zorgen voor de interactie tussen een proces en het besturingssysteem. Het zijn routines waarvoor een proces zelf geen uitvoerrechten heeft. Voor het schrijven naar het scherm kan de system call `write()` gebruikt worden. Een proces roept `write()` aan, waarna het besturingssysteem de system call uitvoert. Als dit gebeurt is, dan krijgt het proces de `write()` "return status" van het besturingssysteem terug. Nu kan het proces weer verder met de volgende taken. Een *jailer* onderschept system calls voor (en soms na) een system call en kan door de system call te weigeren of de argumenten te bewerken het gedrag van een *prisoner* beïnvloeden.

De jail omgeving die gebruikt wordt als basis van het werk in deze scriptie is beschreven in "A SECURE JAILING SYSTEM FOR CONFINING UNTRUSTED APPLICATIONS" [3]. Het voordeel van deze *jailer* is dat het een "user-mode" system call interceptie gebaseerde jail omgeving is. Een gewone gebruiker mag deze dus ook opstarten in tegenstelling tot `chroot`.

[1] Dit is een `chroot` voorbeeld.
[2] `http://www.openbsd.org/cgi-bin/man.cgi?query=chroot`
[3] `http://wiki.netbsd.se/How_to_break_out_of_a_chroot_environment`
[4] Soms ook syscall genoemd

1.1 Traceren van de system calls

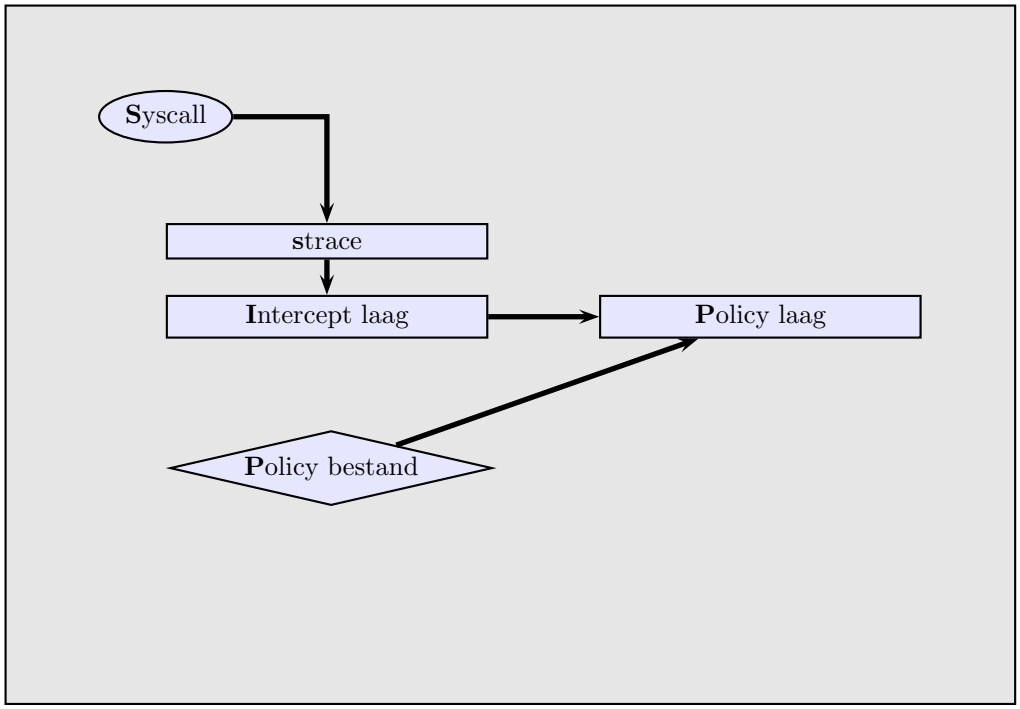
De gebruikte jail omgeving maakt gebruik van `ptrace` om system calls te onderscheppen. Dit is zelf een system call die normaal bedoeld is voor debuggers. Het maakt het mogelijk om van een proces de system calls en de argumenten daarvan te bekijken. `ptrace` maakt het ook mogelijk om de system call en/of argumenten bij te werken.

`ptrace` wordt als volgt gebruikt door de jailer: Een programma wordt via de *jailer* gestart. Dit programma wordt ook wel de *prisoner* genoemd. De *jailer* verbindt zich middels een `ptrace` "attach" operatie met de prisoner. Als de *prisoner* nu een system call aanroept zal eerst de *jailer* verwittigd worden. De *jailer* kan zien wat voor system call de *prisoner* wil uitvoeren en wat voor argumenten hiervoor gebruikt worden. In de tussentijd doet de *prisoner* niets, tenzij de *prisoner* uit meerdere threads bestaat. Tijdens het bekijken van de syscall en de argumenten is het mogelijk om deze te wijzigen. Dit is het moment voor de *jailer* om risicovolle system calls te weigeren, of deze te wijzigen op basis van een policy [3]. Als de *jailer* alle operaties afgerond heeft, dan krijgt het besturingssysteem de opdracht om de *prisoner* verder te laten gaan. De system call wordt nu uitgevoerd. Als de system call uitgevoerd is, dan zorgt `ptrace` er weer voor dat de *jailer* de "return status" van de system call kan bekijken. Ook hier is het weer mogelijk om veranderingen toe te passen. Als de *jailer* hiermee klaar is, dan wordt de uitvoer van de *prisoner* hervat.

Het jailing systeem is niet "from scratch" gebouwd met `ptrace`. Hiervoor is `strace` gebruikt. Dit is een programma dat, met behulp van `ptrace`, een lijst van system calls en signals van een executable weergeeft. Door `strace` te bewerken was het mogelijk om snel een basis te leggen voor de jailer. De bijgewerkte `strace` kan nu dus system calls en bijbehorende argumenten van een proces onderscheppen en deze doorgeven, zodat ze, waar nodig, bijgewerkt kunnen worden.

1.2 De gelaagde opbouw van de jailer

Het jailing systeem is opgebouwd uit diversen lagen. Deze lagen zijn weergegeven in figuur 1.1.



Figuur 1.1: Basis jailing systeem

Een prisoner proces wordt opgestart binnen de jail omgeving. Het prisoner proces roept sys-

tem calls[5] aan om taken uit te voeren. Deze system calls komen binnen bij `strace`. `strace` filtert alle system calls, zodat de system call gegevens, zoals syscall ID en de parameters, doorgegeven kunnen worden aan de intercept laag.

De intercept laag voert allerlei acties uit op de sytem calls. Er moet bijvoorbeeld op een veilige manier geheugen gereserveerd worden waarin argumenten, die normaal in prisoner address space staan, worden weggeschreven. Hierdoor kan de *prisoner* deze niet meer modificeren. Voorbeelden hiervan zijn strings, zoals een pad of een bestandsnaam.

De policy laag is de laag waar de beslissingen genomen worden. Een paar voorbeelden zijn:

1. Moet een syscall wel doorgaan of moet de actie op een andere manier uitgevoerd worden?
2. Mag een file worden gelezen en/of geschreven?
3. Mag een bepaald ip-adres benaderd worden?
4. Mag een proces met een ander proces communiceren?

Via de policy laag heeft de gebruiker de mogelijkheid om de *prisoner* te beïnvloeden. Dit wordt gedaan via een policy bestand. Hier staan de instellingen van wat wel of niet toegelaten wordt, read-only en read-write paden, omgevingsvariabelen en dergelijke. Bij het opstarten van het jailing systeem leest de policy laag het uit.

Niet alleen worden er "at runtime" dingen beslist in de policy laag. Ook worden er dingen beslist tijdens het compileren. Hier staan niet dynamische zaken in, zoals; moet een bepaalde system call wel/niet uitgevoerd worden en/of moet het resultaat van een uitgevoerde system call altijd gecontroleerd worden.

1.3 Waarom een resource manager?

De *jailer* zorgt voor een veilige omgeving waar een prisoner proces kan draaien. Het geheugenbeheer wordt netjes geregeld, system calls die schade kunnen aanrichten worden onderschept en allerlei andere dingen worden bijgestuurd [3].

Dit alleen maakt het jailing systeem niet volledig veilig. Een van de dingen die missen in het huidige ontwerp, is een resource management faciliteit. Dit kan belangrijk zijn voor het voorkomen van denial-of-service aanvallen. Met resource management wordt ook een mogelijkheid gecreëerd om resources tussen verschillende processen eerlijk te delen.

1.3.1 Resources delen

Met de resource manager is het mogelijk om resources te beperken per proces. Een proces die beheerd wordt door een resource manager krijgt zo niet de kans om resources van de overige processen te stelen. Dit geldt niet alleen voor één proces. Als gebruik wordt gemaakt van een resource manager, dan is het mogelijk om meerdere processen, op het zelfde systeem, eerlijk de resources te laten delen.

1.3.2 Denial-of-service aanvallen

Het probleem van denial-of-service aanvallen is dat deze meestal met gewone middelen uitgevoerd worden [2]. Zo kan een vijandig proces een groot gedeelte van de CPU tijd van een systeem gebruiken. Het gevolg is dat andere processen op datzelfde systeem bijna, of helemaal geen, CPU tijd meer krijgen. Het gevolg kan een traag of niet functionerend systeem opleveren.

Hetzelfde geldt voor het geheugen gebruik. Een vijandig proces kan, net als het overbelasten van een CPU, ook het geheugen overbelasten. Als een proces enorm veel geheugen opslokt, dan wordt het noodzakelijk de swap partitie actief te gaan gebruiken. Hierdoor wordt het systeem een heel stuk trager. Dit is nog niet het ergste. Als de swap ruimte ook volledig vol raakt kan het zo zijn dat er andere processen gestopt moeten worden, of erger, ze crashen simpelweg[6]. In deze scriptie wordt het managen van CPU resources behandeld en niet dat van het geheugen.

[5] Soms ook syscall genoemd

[6] [2] 14.4.1 pagina 502

1.4 Scriptie overzicht

In deze scriptie wordt een resource manager voor CPU tijd besproken. In hoofdstuk 2 wordt het ontwerp besproken, daarna de implementatie in hoofdstuk 3.

Een kleine evaluatie van de resource manager wordt beschreven in hoofdstuk 4.

Een beschrijving van de mogelijke toekomstige uitbreidingen is te vinden in hoofdstuk 5.

In bijlage A is een lijstje te vinden met veel gebruikte belangrijke termen. Basis informatie over de code is te vinden in bijlage B. Bijlage C bevat een system call lijst die door de tekst gebruikt wordt.

HOOFDSTUK 2

Resource manager ontwerp

De complete *jailer* is gebaseerd op het onderscheppen, verwerken en monitoren van system calls. Het is van belang om te weten wat de mogelijkheden van de system calls zijn. Om hier achter te komen zijn alle Linux system calls bekeken. In deze lijst komen vier belangrijke subcategorieën naar voren welke weergegeven zijn in tabel 2.1.

Risico's	Zie bijlage
Geheugen gerelateerd	C.1
Bestandssysteem gerelateerd	C.2
Denial-of-service gerelateerd	C.3
Overige risicovolle system calls	C.4

Tabel 2.1: System call categorieën

Geheugen categorie: In deze categorie staan alle system calls die relatief veel geheugen gebruiken/alloceren. Het is mogelijk dat bij (herhaaldelijk) aanroepen van deze system calls te veel geheugen van het systeem gebruikt wordt.

Bestandssysteem categorie: Deze categorie bevat alle system calls die gebruik maken van het bestandssysteem. Hier moeten de system calls in de gaten gehouden worden als we niet willen dat het bestandssysteem volloopt. Als een vijandig proces bijvoorbeeld een paar keer een `write()` met veel data aanroept, dan zit de harde schijf zo vol.

Denial-of-service categorie: De system calls in deze categorie geven een vergroot risico op een denial-of-service aanval. Deze system calls geven namelijk een zwaardere, of kunnen een zwaardere, belasting van het systeem veroorzaken. Zo is het mogelijke dat `kill()` via een vijandig proces enorm veel *signals*, mogelijk recursief, genereert wat op zijn beurt weer veel kernel resources (*systeemtijd*) inneemt. Door het limiteren van *systeemtijd*, via system calls, is het mogelijk om dit soort aanvallen te voorkomen.

Overige risicovolle system calls categorie: Dit zijn risicovolle system calls die verder niet in de scriptie behandeld worden, omdat dit niet binnen het domein van de resource manager valt. Ze worden hier wel weergegeven, omdat deze system calls systeembrede veranderingen kunnen aanbrengen op het systeem. Het aanbrengen van deze veranderingen kan grote gevolgen hebben voor het systeem. Hiervoor hebben ze wel hogere permissies nodig dan een normale gebruiker.

2.1 Systeem tijd limiteren

Alle system calls gebruiken CPU tijd. Om het CPU gebruik te beperken/controleren moet men kunnen voorspellen hoeveel CPU-/systeemtijd een system call kost. Het zal niet werken door te kijken of een proces 100% van de CPU gebruikt, waarna het proces afgesloten of bijgesteld wordt. Het proces is dan al zo ver gekomen dat het voorbij een limiet gaat. Hier heeft de resource manager dus niet de volledige controle over het prisoner proces.

Het is de bedoeling om het proces in de hand te houden, zodat de resource manager de volledige controle heeft over het prisoner proces. Een prisoner proces mag een gedeelte van de CPU tijd gebruiken en niet meer. Dit is mogelijk door systeemtijd (=CPU gebruik van system calls) te meten en te limiteren. Dit wordt geïmplementeerd door middel van *timeslots*. Een prisoner proces krijgt een *timeslot* van een instelbaar aantal microseconden per *tijdframe*. Voor het *tijdframe* wordt, in deze scriptie, één seconde gebruikt. Het volgende stelt een *tijdframe* voor met daar binnen een timeslot.

Figuur 2.1: Tijdframe

Een prisoner proces moet binnen een *timeslot* zijn system calls uitvoeren. Als een system call niet meer binnen een *timeslot* past, dan moet het hele prisoner proces wachten op het volgende *tijdframe* met een nieuw *timeslot*.

Het huidige ontwerp gebruikt *timeslots* voor de complete jail omgeving. Meerdere prisoner processen binnen één *jail* moeten één *timeslot* delen.

2.1.1 Denial-of-service aanvallen

Het implementeren van *timeslots* heeft nog een voordeel. Het maakt de kans erg klein dat de system calls in de denial-of-service categorie schade kunnen aanrichten. Dit komt omdat het aantal system calls per seconde veel lager komt te liggen dan normaal, tenzij het *timeslot* erg groot ingesteld is. Zo is het voor een `kill()` niet meer mogelijk om continu *signals* te versturen. Er is namelijk een gedeelte, van een *tijdframe*, dat er geen system calls uitgevoerd mogen worden. In dit gedeelte kan `kill()` dus ook niets doen.

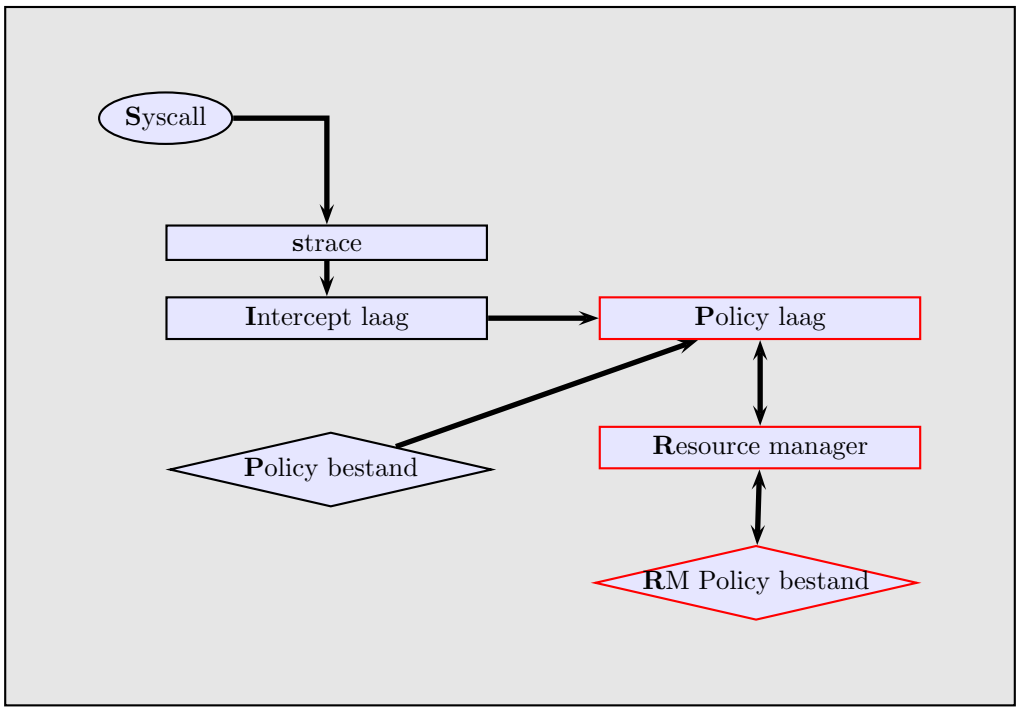
2.2 Data genereren

Voor het beheren/implementeren van *timeslots* moet er informatie beschikbaar zijn. De informatie die nodig is, is de tijd die een system call nodig heeft om uitgevoerd te worden. Als dit niet bekend is, dan is het niet mogelijk om te bepalen of een system call nog uitgevoerd kan worden binnen een *timeslot*.

Deze informatie bestaat nog niet, dus deze moet gegenereerd worden. De *jailer* heeft hiervoor een genereer mode. In deze mode weet de resource manager dat er nieuwe data gegenereerd moet worden. Dit wordt gedaan met behulp van een testprogramma. Dit kan elk stukje code zijn. Het beste is om hiervoor code te gebruiken die representatief is voor de code die later in de praktijk ook gebruikt wordt. De resource manager zal alle uit te voeren system calls timen. Als meerdere keren dezelfde system call uitgevoerd wordt door het testprogramma, dan wordt de *worst case* timing aangehouden. Dit is de timing waar de meeste microseconden voor gebruikt zijn.

2.3 Totaal plaatje

In figuur 1.1 is de schematische opstelling van het jailing systeem te zien. Dit is nog de versie zonder resource manager. Nu met resource manager komt het plaatje eruit te zien als in figuur 2.2.



Figuur 2.2: Integratie resource manager in het jailing systeem

Er is nu een resource manager laag toegevoegd en een nieuw policy bestand[1] voor deze laag.

De policy laag is verantwoordelijk voor het aanroepen van de resource manager. De resource manager zal via de policy laag alle informatie van een system call ontvangen, zoals de uitvoer duur, syscall ID, e.d.. De niet timing gerelateerde informatie van de system calls krijgt de policy laag weer van de intercept laag.

Alle getime'de informatie wordt met behulp van een datastructuur naar het "RM policy bestand" geschreven. Deze datastructuur is geïntegreerd in de resource manager, en wordt in detail beschreven in hoofdstuk 3.

[1] RM (=Resource Manager) policy bestand

HOOFDSTUK 3

Implementatie van de resource manager

Het implementeren van het ontwerp kan opgedeeld worden in twee delen. Het eerste deel is de basis van de resource manager. Deze basis moet zorgen dat system calls getimed worden en dat de resultaten opgeslagen worden.

Het tweede deel is de implementatie van de werkelijke resource manager. In dit tweede deel worden de opgeslagen resultaten gebruikt voor het managen van CPU resources.

3.1 Uitgebreide datastructuur

Tijdens het timen van system calls, wat later besproken wordt, maar ook na het timen, moet timing informatie beschikbaar zijn. Hiervoor is een uitgebreide datastructuur bedacht. Deze datastructuur is zo gemaakt dat de informatie makkelijk te benaderen is. Ook is er rekening gehouden met de toekomst. De mogelijkheid bestaat dat het huidige policy bestand[1] van de policy laag omgezet kan worden naar deze datastructuur. De datastructuur ondersteund verschillende soorten data. De data wordt opgeslagen in paren. Een paar bestaat uit een label en de daaraan gekoppelde data. De dataparen die op het moment beschikbaar zijn, worden weergegeven in tabel 3.1.

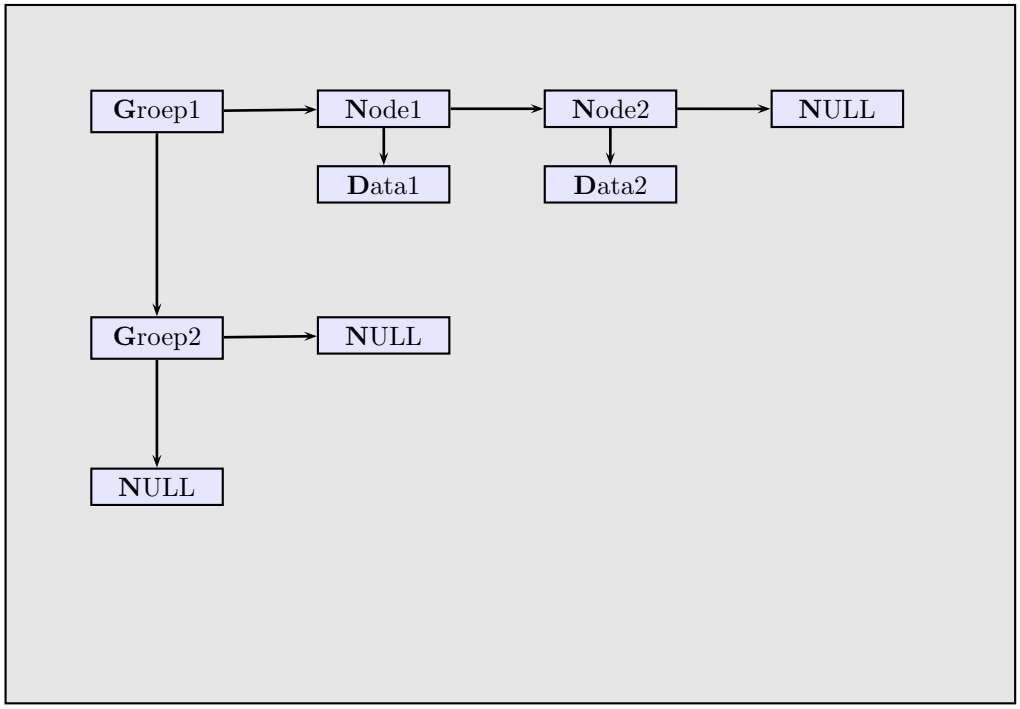
Data	Voorbeeld
LABEL=FLOAT	Waarde=1.2245
LABEL=INTEGER	Nummer=1245
LABEL=STRING	Naam=John

Tabel 3.1: Dataparen van de datastructuur

Een datapaar behoord tot een groep. Elke groep kan nul of meerdere dataparen bevatten.

[1] Zie weergegeven “Policy bestand” in figuur 1.1

3.1.1 Opbouw van de datastructuur



Figuur 3.1: Datastructuur

De datastructuur is op te delen in drie onderdelen. Het eerste onderdeel is een groep. Een groep heeft een naam, een pointer naar de volgende groep en naar een data node.

De data node bevat een pointer naar de data en naar de volgende node. Een node kan naar verschillende soorten data verwijzen. Op het moment kunnen de datatypes uit tabel 3.1 gebruikt worden. Er is gekozen voor nodes, zodat het makkelijker is om door de lijst van nodes te zoeken.

3.2 Timen system calls

In de *jailer* bestaan twee functies, die gebruikt worden als *pre_syscall* en *post_syscall*. De *pre_syscall* wordt aangeroepen voordat een system call uitgevoerd wordt en *post_syscall* na het uitvoeren.

Als *pre_syscall* aangeroepen wordt, dan wordt een timer gestart. Hierna wordt de system call volledig uitgevoerd. Na het uitvoeren wordt de timer weer gestopt via *post_syscall*. Het timing resultaat[2] wordt opgeslagen in de datastructuur. Het kan voorkomen dat er meerdere keren een zelfde system call uitgevoerd wordt. In dit geval wordt het worst-case(=de meest verstreken tijd) resultaat opgeslagen. Zo zal er van elke system call maar één waarde bestaan in de datastructuur.

3.3 Gegevens wegschrijven

Na het timen van de system calls wordt de informatie opgeslagen in een bestand. Als later de *jailer* weer opgestart wordt, dan is het mogelijk dit bestand op te geven, om de data uit te lezen. Met de informatie in dit bestand is het mogelijk om de *timeslots* in de resource manager te implementeren. Deze implementatie wordt besproken in 3.5.

Via de datastructuur is het heel makkelijk om de gegevens naar een bestand te schrijven of eruit te lezen.

[2]Het aantal verstreken μs

Een voorbeeld van zo'n bestand is te zien in Bestand 1.

Bestand 1 Indeling datastructuur bestand.

```
; Dit is commentaar.
[Groep] ; Dit is een groep
kill=25
write=56

[AndereGroep]
counter=3
```

De indeling van het bestand 1 met groepen tussen blokhaken, waaronder de data staat, lijkt sterk op de datastructuur in figuur 3.1. Alleen de nodes komen niet voor in het bestand, omdat dit "hulp" data is.

3.4 Initialisatie van de resource manager

De resultaten die naar een bestand geschreven zijn moeten tijdens het starten van de *jailer* weer geladen worden. Hier is een functie voor geschreven in de resource manager. Met deze functie worden de resultaten uitgelezen en de resource manager geïnitialiseerd.

3.4.1 Commandline opties

Tijdens het initialiseren van de resource manager wordt er ook gekeken naar de commandline opties. Er zijn in totaal vier opties geïmplementeerd:

`--rmgt-systime-us-per-sec <nummer>` : Dit is het aantal μs per seconde dat beschikbaar is voor system calls binnen een seconde. Dit geeft dus de grootte van een *timeslot* aan.

`--gen-rmgt-policy <bestand>` : Hier wordt een bestand opgegeven dat gevuld wordt met de timing data. Deze optie is ervoor als er nog geen timing data beschikbaar is. Deze dient dus bij het eerste gebruik van de resource manager aangeroepen te worden.

`--rmgt <bestand>` : Met deze optie wordt een bestand opgegeven. Uit dit bestand wordt de timing data gelezen. Als *–gen-rmgt-policy <bestand>* de eerste keer aangeroepen wordt, kan later dus *–rmgt <bestand>* met hetzelfde bestand gebruikt worden.

`--update-rmgt-policy <bestand1> <bestand2>` : Deze optie maakt het mogelijk een bestand met timing data te updaten. *<bestand1>* is het bestand dat gelezen wordt en *<bestand2>* is de ge-update'te versie van *<bestand1>*. Zo is het mogelijk om de originele gegevens te bewaren.

3.5 Timeslots implementeren

Als de resource manager geïnitialiseerd is, dan staat de datastructuur weer vol met, in het verleden getime'de, resultaten. Deze informatie wordt gebruikt door de timeslots.

Voor het implementeren van de *timeslots* zijn een aantal dingen nodig. Namelijk:

- Een `alarm()` die elke seconde een *signal* geeft[3]
- Opslag van het aantal verbruikte μs door system calls in het huidige timeslot
- Een datastructuur met de gegevens van het bestand (system call timing gegevens)
- De waarde `resm_max_usec`, wat de grootte van een *timeslot* representeert[4]

De resource manager bepaald telkens of een system call mag worden uitgevoerd. Dit wordt gedaan tijdens de *pre_syscall* en gaat als volgt:

Algorithm 1 *WAIT algoritme*

Require: Een integer $resm\_max\_usec > 0$.

1: **if** [*tijd nodig om syscall uit te voeren*] *niet beschikbaar in datastructuur* **then**
2: [*tijd nodig om syscall uit te voeren*] = $1ms$
3: **else**
4: [*tijd nodig om syscall uit te voeren*] = *waarde uit datastructuur*
5: **end if**
6: [totale tijd] = [totale verbruikte tijd] + [tijd nodig om syscall uit te voeren]
7: **if** [*totale tijd*] > *resm_max_usec* **then**
8: System call moet wachten
9: **else**
10: System call mag doorgaan
11: [*totale verbruikte tijd*] += [*tijd nodig om syscall uit te voeren*]
12: **end if**

[*totale verbruikte tijd*] is de totale tijd die al verbruikt is door andere system calls in het huidige timeslot. Het laten wachten van een system call wordt gedaan met een `pthread_cond_wait`. Hierdoor zal de hele *jailer* in "wait"-mode staan.

Soms blijkt dat de schatting, voor een system call uitvoer tijd, die in de datastructuur staat, te laag is. Dit wordt verholpen door de werkelijke tijd tijdens het uitvoeren van de system call te timen. Tijdens de *pre_syscall* wordt een timer gestart en deze wordt weer gestopt in *post_syscall*[5]. Het huidige wordt-case resultaat in de datastrucuur wordt vervangen door het werkelijke worst-case resultaat. Deze datastructuur met de up-to-date resultaten worden weer naar een bestand geschreven in de `--update-rmgt-policy` mode. Voor meer informatie over deze mode, zie hoofdstuk 3.4.1.

3.5.1 Het alarm signaal

In 2.1 wordt besproken wat een *tijdframe* is en dat een *timeslot* daar binnenvalt. Met een `alarm()` wordt een *tijdframe* afgebakend. Als een alarm signaal wordt gegeven, dan betekent dat het einde van het huidige *tijdframe* en een begin van een nieuwe. Tijdens dit signaal wordt het aantal verbruikte μs weer gereset naar 0. Aan het einde wordt de `pthread_cond_wait`, die aangeroepen wordt wanneer een system call moet wachten, weer opgeheven.

[3] Dit bakend een *tijdframe* af.
[4] Binnen een *tijdframe* van één seconde.
[5] Zie hoofdstuk 3.2

HOOFDSTUK 4

Resource manager evaluatie

Door het uitvoeren van een aantal commando's is het mogelijk om te zien of alles werkt zoals het hoort.

Eerst dient er een timing bestand aangemaakt te worden met de optie *–gen-rmgt-policy <bestand>*[1]:

Shell listing 1 Het genereren van een timing bestand.

```
$ bin/jailer --gen-rmgt-policy $PWD/timingdata -d ~/mansion-jail/0 \
  -p bin/jail-policy ls -al /usr/*

  [...Veel output...]

$ cat timingdata
[GLOBAL]
SYS_write=8083
SYS_open=12121
SYS_fstat=200
SYS_mmap2=195
SYS_close=220
SYS_ioctl=126
SYS_read=148
SYS_munmap=237
SYS_lstat=8012
SYS_getxattr=20548
SYS_socket=218
SYS_fcntl=121
SYS_connect=412
SYS__llseek=233
SYS_readlink=9619
SYS_clock_gettime=26
SYS_getdents=447
SYS_mremap=47
SYS_brk=18
SYS_shmdt=56
```

Het timen is dus gelukt. Er zijn waardes naar het bestand geschreven.

[1] Alle commando's worden uitgevoerd in de root directory van de jailer.

Nu twee tests met verschillende waardes voor *–rmgt-systime-us-per-sec <nummer>*, dus een andere grootte voor het timeslot.

Shell listing 2 Bewijs werking van resource manager.

```
[cmd1]  $ time bin/jailer --rmgt $PWD/timingdata -d ~/mansion-jail/0 \
        -p bin/jail-policy --rmgt-systime-us-per-sec 500000 ls -al /usr/*

        [...Veel output...]

        real    0m5.689s
        user    0m0.883s
        sys     0m1.293s

[cmd2]  $ time bin/jailer --rmgt $PWD/timingdata -d ~/mansion-jail/0 \
        -p bin/jail-policy --rmgt-systime-us-per-sec 250000 ls -al /usr/*

        [...Veel output...]

        real    0m9.334s
        user    0m0.847s
        sys     0m1.437s
```

Bij *cmd1* en *cmd2* is te zien dat de `sys`-tijden dicht bij elkaar liggen, terwijl de `real`-tijd van *cmd1* de helft is van de `real`-tijd van *cmd2*.

De *jailer* voert het commando `ls -al /usr/*` uit. Dit is bij *cmd1* en *cmd2* zo. De hoeveelheid system calls veranderd dus niet, wat ook te zien is aan de bijna gelijke `sys`-tijden.

Het timeslot in *cmd1* is twee keer zo groot als die van *cmd2*. *cmd1* kan dus in één seconde twee keer zoveel system calls verwerken als *cmd2*. Dit betekend dat *cmd2* het dubbele aantal timeslots, dus secondes, nodig heeft als *cmd1*. Zo wordt de de dubbele `real`-tijd van *cmd2* verklaard. Dat de `sys`-tijd van *cmd2* iets hoger ligt, kan worden verklaard door de `alarm()` overhead. `alarm()` is immers ook een system call en wordt in *cmd2* twee keer zoveel aangeroepen als in *cmd1*.

Hiermee is bewezen dat de resource manager werkt.

HOOFDSTUK 5

Toekomstige uitbreidingen van de resource manager

De basis voor een resource manager is gelegd. Na deze basis zijn er nog veel dingen die uitgebreid kunnen worden.

5.1 Datastructuur voor de toekomst

De datastructuur is, zoals al eerder gezegd, gemaakt met het oog op de toekomst. Zo is het mogelijk om meerdere data groepen te gebruiken in de datastructuur. Deze groepen kunnen ook in een bestand verwerkt worden zoals te zien is in bestand 1 in hoofdstuk 3.3. Wat misschien opgevallen is, is dat de shell listing 1 in hoofdstuk 4 maar één groep genereert. Dit is de groep `GLOBAL`. Deze groep bevat alle timing resultaten.

Het is de bedoeling dat in de toekomst meerdere groepen aangemaakt worden. Bijvoorbeeld een groep `LIMITS`. In deze groep wordt dan opgegeven hoeveel system calls van een bepaald type te gelijk mogen worden uitgevoerd. Zo is het aantal threads of forks te beperken die tegelijk worden uitgevoerd. Een voorbeeld is te zien in bestand 2.

5.2 Aparte groepsindelingen

In hoofdstuk 3.5 is beschreven, dat voor een system call die niet in de datastructuur voorkomt, 1ms gerekend wordt. Dit staat nu vast in de code, maar dit kan ook in een groep gezet worden. In bestand 2 is een groep `DEFAULTS` gedefinieerd met een variabele `unknown_syscall`. Dit zou gebruikt kunnen worden om de waarde dynamisch te maken.

Sommige system calls kunnen in categorieën worden opgedeeld. Elke categorie heeft een waarde. De waarde staat voor het aantal μs dat een system call nodig heeft om uitgevoerd te worden. In bestand 2 is een groep `CATEGORIES` gedefinieerd met een variabele `filesystem_cat`[1] . De categorienaam kan dan gebruikt worden als waarden voor een variabele in de groep `GLOBAL`. Dit heeft als voordeel dat een standaard bestand gemaakt kan worden. Hier staan dan allemaal `CATEGORIES`-variabelen in met alle bestaande system calls in `GLOBAL`. Alle `GLOBAL`-variabelen worden verbonden aan een categorie. Zo is het niet meer nodig om eerst een bestand te genereren bij de eerste keer opstarten van de resource manager. Tijdens het uitvoeren van de resource manager kan eventueel het bestand ge-update worden, zodat getime'de system calls een echte waarde krijgen.

Deze indeling zal een groter overzicht opleveren en maakt het ook mogelijk om één waarde te veranderen voor een hele categorie.

[1] Dit is alleen een voorbeeld. In de praktijk zal de `filesystem_cat` categorie nooit kunnen bestaan, want bestandssysteem system calls zijn één van de meest dynamische system calls die er zijn qua uitvoertijd.

Bestand 2 Toekomst: Indeling bestand.

```
[LIMITS]
threads=5 ; Maximaal vijf threads naast elkaar draaien
fork=6

[DEFAULTS]
unknown_syscall=1000000 ; Waarde voor niet getime'de system calls

[CATEGORIES]
filesystem_cat=20000

[GLOBAL]
SYS_write=filesystem_cat
SYS_open=filesystem_cat
SYS_fstat=200
SYS_mmap2=195
SYS_close=filesystem_cat
SYS_ioctl=126
SYS_read=filesystem_cat
SYS_munmap=237
SYS_lstat=8012
SYS_getxattr=20548
SYS_socket=218
SYS_fcntl=121
SYS_connect=412
SYS__llseek=233
SYS_readlink=9619
SYS_clock_gettime=26
SYS_getdents=447
SYS_mremap=47
SYS_brk=18
SYS_shmdt=56
```

5.3 Andere manier van WAIT

In hoofdstuk 3.5 is kort beschreven hoe het WAIT mechanisme werkt. Het WAIT mechanisme is niets anders dan het gebruik van een `pthread_cond_wait`. Dit wordt gebruikt om een system call, die niet genoeg tijd in de *timeslot* heeft, te laten wachten. Het gevolg is dat alles moet wachten. System calls van andere threads die nog wel in het *timeslot* passen dus ook.

Het is in de toekomst beter om dit op een andere manier te implementeren, zodat de system calls van andere threads wel door kunnen gaan. De system call die moet wachten is dan de enige wachtende.

BIJLAGE A

Verklarende woordenlijst

Jailer : Het jailing systeem proces dat waakt over prisoner processen.
Prisoner : Het proces dat bewaakt wordt door de jailer.
System calls : Een interface tussen een proces en het besturingssysteem. [2] [1]
Timeslot : De bruikbare tijd voor de system calls binnen een seconde.
Tijdframe : Tijdsperiode waar een time slot binnen valt. In deze scriptie wordt één seconde gebruikt. Zie 2.1.
pre_syscall : Een symbolische functie die uitgevoerd wordt direct voordat een system call uitgevoerd wordt.
post_syscall : Een symbolische functie die uitgevoerd wordt direct nadat een system call uitgevoerd is.
strace[1] : Een programma om system calls te traceren. De jailer is op een aangepaste versie van `strace` gebaseerd.
$JAIL_DIR : De root directory van de *jailer* broncode.

[1] `http://sourceforge.net/projects/strace/`

BIJLAGE B

Gebruik van de huidige code

In deze bijlage wordt de source-tree en een aantal belangrijke functies van de resource manager broncode beknopt behandeld. De code is in de C-taal geïmplementeerd.

B.1 Build omgeving

Er is een nieuwe directory aangemaakt in de jailing systeem source-tree. In deze directory wordt, op een paar uitzonderingen na, alle code geschreven. Dit is om te zorgen dat de resource manager zo modulair mogelijk is. Hierdoor blijft het overzicht over de code optimaal.

Om de code te compileren is gebruik gemaakt van GNU Make[1]. Bij het commando `make` worden alle subdirectories doorlopen en de daar gevonden code gecompileerd. Hiervan wordt één static library gemaakt. Eén static library heeft als voordeel, dat tijdens het integreren van de resource manager in de jailer, er maar een library meegelinked hoeft te worden.

Ook is er aan documentatie gedacht. Hiervoor wordt Doxygen[2] gebruikt. Met het `make docs`[3] commando wordt de documentatie gegenereerd. De documentatie is dan te vinden onder `$JAIL_DIR/resource_mgr/Docs/html`.

B.2 Functies in de resource manager laag

Hier worden de belangrijke functies beschreven die in de resource manager laag staan.

Functies in **$JAIL_DIR/resource_mgr/resm_timing.{h,c}**[4]:

`int mjr_resourcem_init(int *argc, char *argv[]);`
Deze functie zorgt voor de initialisatie van de resource manager. De dingen die zij doet zijn: de initialisatie van de datastructuur, de commandline opties verwerken en bestanden uitlezen. Dit is de eerste functie van de resource manager die aangeroepen wordt. Meer hierover is te vinden in hoofdstuk 3.4.

Tabel B.1: mjr_resourcem_init

[1] `http://www.gnu.org/software/make/`
[2] `http://www.stack.nl/~dimitri/doxygen/`
[3] Dit moet gedaan worden in de directory `$JAIL_DIR/resource_mgr`
[4] `$JAIL_DIR` staat voor de root directory van de *jailer* broncode

`int mjr_resourcem_end(void);`
Dit is de functie die het laatste aangeroepen wordt in de resource manager. Zij is verantwoordelijk voor het schrijven van de timing gegevens naar een bestand en het opruimen van de datastructuur.

Tabel B.2: mjr_resourcem_end

`void alarmHandler(int sig);`
Deze functie wordt bij elke `alarm()` *signal* aangeroepen. Dit kondigt een nieuwe *timeslot* aan. Meer informatie hierover is te vinden in hoofdstuk 3.5.

Tabel B.3: alarmHandler

`int checkTimeSyscall(int syscall, long *worstCaseTime);`
Deze functie controleert of een system call nog in een *timeslot* past. Hier wordt ook de actie ondernomen als er gewacht moet worden. Hier wordt dus de `pthread_cond_wait` aangeroepen. Voor meer informatie over dit aanroepen, zie hoofdstuk 3.5.

Tabel B.4: checkTimeSyscall

B.2.1 Datastructuur

Hier wordt de basis uitgelegd van de datastructuur samen met de belangrijkste functies door middel van een stukje code.

Programma 1 Basis werking

```
...
#include ...
#include "policydata.h"
#include "configwrite.h"

/* Declaratie datastructuur */
struct policy_group *lijst = NULL;

addGroup(&lijst, "Groep1"); /* Een groep toevoegen */

addGroup(&lijst, "Groep2"); /* Nog een groep toevoegen */

long value1 = 10;
long value2 = 20;

/* Een integer waarde toevoegen aan de groep Groep1 */
addIntValueItem(&lijst, "Groep1", "Data1", value1);

/* Nog een waarde voor de groep Groep1 */
addIntValueItem(&lijst, "Groep1", "Data2", value2);

/* Een waarde uit de datastructuur afdrukken op het scherm */
long newValue;
getIntValue(lijst, "Groep1", "Data1", &newValue);
printf("Dit is een waarde uit de datastructuur : %ld\n", newValue);

/* Schrijven van de data naar een bestand */
writeConfig("bestandsnaam" , lijst);

/* Datastructuur opruimen */
freeList(lijst);

...
```

De code in programma 1 laat de basiswerking van de datastructuur zien. In de code is geen foutafhandeling verwerkt. Dit is wel mogelijk, want elke functie geeft een “fout code” terug.

De code in programma 1 is zo gemaakt, dat deze precies hetzelfde representeert als de datastructuur in figuur 3.1.

Belangrijke functies

Functies in **$JAIL_DIR/resource_mgr/datastruct/policydata.{h,c}** :

`int addGroup(struct policy_group **listAnchor, const char *groupName);`
Met deze functie is het mogelijk om een groep toe te voegen aan de datastructuur.

Tabel B.5: addGroup

`int addItem(struct policy_group **listAnchor, const char *groupName, void *dataItem, const int dataItemType);`
Deze functie maakt het mogelijk om een data item aan een groep toe te voegen. Dit is een globale functie waarmee allerlei soorten data toegevoegd kunnen worden. De soorten data staan in tabel 3.1.

Tabel B.6: addItem

`int addIntValueItem(struct policy_group **listAnchor, const char *groupName, const char *label, long value);`
Dit is een zelfde soort functie als `addItem` in tabel B.2.1, maar hiermee is het direct mogelijk om een integer waarde toe te voegen aan de datastructuur. Het direct toevoegen van integer waardes is makkelijk voor het toevoegen van getime'de system calls.

Tabel B.7: addIntValueItem

`int getIntValue(struct policy_group *listAnchor, const char *groupName, const char *label, long *value);`
Met deze functie is het mogelijk om een integer waarde uit de datastructuur te lezen. Dit is makkelijk voor het uitlezen van getime'de system call waardes.

Tabel B.8: getIntValue

`int freeList(struct policy_group *listAnchor);`
Deze functie is erg belangrijk. Bij het aanroepen van deze functie wordt het geheugen van de lijst vrijgegeven. Met het geheugen wordt bedoeld; de lijst structuur zelf en de data in de lijst.

Tabel B.9: freeList

Functie in **$JAIL_DIR/resource_mgr/config_reader/configwrite.{h,c}** :

`int writeConfig(char *path, struct policy_group *listAnchor);`
Met deze functie is het mogelijk om de datastructuur naar de harde schijf te schrijven.

Tabel B.10: writeConfig

Functie in **$JAIL_DIR/resource_mgr/config_reader/configread.{h,c}** :

`int readConfig(char *path, struct policy_group **configAnchor);`
Voor het lezen van data uit een bestand moet deze functie gebruikt worden.

Tabel B.11: readConfig

B.2.2 Code documentatie

Door de hele code staat veel commentaar. Het is dus aan te bevelen om de code ook eens te lezen. Het meeste is geschreven in Doxygen formaat. In hoofdstuk B.1 wordt uitgelegd hoe de Doxygen documenten kunnen worden gegenereerd.

Alle code die niet in de resource manager staat, maar bijvoorbeeld in de policy laag, wordt voorafgegaan met `/* Resource manager */` commentaar. Zo is duidelijk te zien welke code bij de resource manager hoort.

BIJLAGE C

System call lijst

In deze bijlage staan de system call categorieën die besproken worden in de scriptie.

C.1 Geheugen gerelateerde risico's

System call	Nº	Description
fork	2	Makes a copy of the parent process. Uses Copy on Write.. The child process is an exact memory copy of the parent except the executable code. The executable code is shared between parent and child. The total memory usage of an fork will be $MemUsageParent - ExecMem = MemUsageChild$
execve	11	Executes a given program. Allocates new memory for this program.
brk	45	Change data segment size. Need to intercept this in the interception layer.
uselib	86	Loads a shared library into memory.
ipc	117	System V IPC system calls. Function that can be used for messages, semaphores, and shared memory. Need to examine all possible `unsigned int call`'s.
clone	120	Create a child process. **How much memory takes a thread?**
create_module	127	Create a loadable module entry. Attempts to create a loadable module entry and reserve the kernel memory that will be needed to hold the module.
vfork	190	Create a child process and block parent. Child shares parent memory. Including stack. No copy-on-write is used.

Tabel C.1: **Memory system calls**

C.2 Bestandssysteem gerelateerde risico's

System call	Nº	Description
write	4	Writes n bytes from an existing buffer to a fd.
mknod	14	Creates a filesystem node. **Filling fileystem with nodes?**
lseek	19	Seeks in a fd. Can write out of bounds.
acct	51	Switch process accounting on or off. When accounting is enabled a file will be appended with info about terminating processes. How to trace the amount of space used by the file?
truncate	92	Truncate a file to a specified length. Can increase file size.
ftruncate	93	Truncate a file to a specified length. Can increase file size.
_llseek	140	Reposition read/write file offset. Possible to seek past file.
writev	146	Write data into multiple buffers. Writes data to fd.
pwrite64	181	Write to a file descriptor at a given offset. Possible write to disk via fd.
sendfile	187	Transfer data between file descriptors. Possible to write to disk via fd.
truncate64	193	Truncate a file to a specified length. Can increase file size. Truncate with a bigger size than the current file size.
ftruncate64	194	Truncate a file to a specified length. Can increase file size. Truncate with a bigger size than the current file size.
setxattr	226	Set an extended attribute value.
lsetxattr	227	Set an extended attribute value.
fsetxattr	228	Set an extended attribute value.
sendfile64	239	Transfer data between file descriptors. Possible to write to disk via fd.
mknodat	297	Create a special or ordinary file relative to a directory file descriptor. See `mknod (14)`.
splice	313	Splice data to/from a pipe. It is possible to copy data instead of moving it.
tee	315	Duplicating pipe content. Possible to write to disk via fd.
fallocate	324	Manipulate file space. Can allocate a bigger range than the current range of the fd.

Tabel C.2: **Filesystem system calls**

C.3 Denial-of-service gerelateerde risico's

System call	Nº	Description
fork	2	Create a child process. This causes a heavy load on a system. It needs to create a complete new process.
execve	11	Execute a file. This causes a heavy load on a system. It needs to create a complete new process.
sync	36	Commit buffer cache to disk. Can be a expensive when lots of data has to be written to disk. It can interfere other processes which are using the disk.
kill	37	Send signal to a process. Signal flooding.
ipc	117	System V IPC system calls. Function that can be used for messages, semaphores, and shared memory. Need to examine all possible `unsigned int call`'s.
fsync	118	Synchronize a file's in-core state with storage device. See `sync (36)`
msync	144	Synchronize a file with a memory map (map to disk). See `sync (36)`
fdatasync	148	Synchronize a file's in-core state with storage device. See `sync (36)`
vfork	190	Create a child process and block parent. See `fork (2)`
tkill	238	Send a signal to a thread. Signal flooding.
tgkill	270	Send a signal to a thread. Signal flooding.
sync_file_range	314	Sync a file segment with disk. See `sync (36)`

Tabel C.3: **DOS attack system calls**

C.4 Overige risicovolle system calls

System call	Nº	Description
setuid	23	Sets the effective user ID of the calling process. User needs CAP_SETUID permissions.
stime	25	Sets system time. User needs CAP_SYS_TIME permissions.(System wide changes)
swapon	87	Start swapping to file/device. User needs CAP_SYS_ADMIN permissions. (System wide changes)
reboot	88	Reboot or enable/disable Ctrl-Alt-Del User needs CAP_SYS_BOOT permissions. (System wide changes)
setpriority	97	Program scheduling priority. User needs CAP_SYS_NICE permissions. (Impact on whole system.)
statfs	99	Get file system statistics. Returns sensitive system information. This information can be used for a break-in attempt.
fstatfs	100	Get file system statistics. See `statfs (99)`
ioperm	101	Set port input/output permissions. User needs CAP_SYS_RAWIO permissions.
iopl	110	Change I/O privilege level. User needs CAP_SYS_RAWIO permissions.
vhangup	111	Virtually hangup the current tty. User needs CAP_SYS_TTY_CONFIG permissions.
idle	112	Make process 0 idle. Only PID 0 can call this function.
swapoff	115	Stop swapping to file/device. Only usable if the user has CAP_SYS_ADMIN permission. (Impact on whole system.)
sysinfo	116	Returns information on overall system statistic. See `statfs (99)`
setdomainname	121	Get/set domain name. Only usable if the user has CAP_SYS_ADMIN permission. (Impact on whole system.)
create_module	127	Create a loadable module entry. User needs CAP_SYS_MODULE permissions. (Impact on whole system.)
init_module	128	Initialize a loadable module entry. User needs CAP_SYS_MODULE permissions. (Impact on whole system.)
delete_module	129	Delete a loadable module. User needs CAP_SYS_MODULE permissions. (Impact on whole system.)
quotactl	131	Manipulate disk quota. User needs to be root.

Tabel C.4: **Privileged user system calls**

System call	Nº	Description
sysfs	135	Get file system type information. See `statfs (99)`
setfsuid	138	Set user identity used for file system checks. User needs CAP_SETUID permissions.
setfsgid	139	Set group identity used for file system checks. User needs CAP_SETGID permissions.
sched_setparam	154	Set scheduling parameters. User needs CAP_SYS_NICE permission. (System wide impact)
sched_getparam	155	Get scheduling parameters. User needs CAP_SYS_NICE permission.
sched_setscheduler	156	Set scheduling algorithm/parameters. Kernels before 2.6.12: Process needs CAP_SYS_NICE privileges. Newer kernels: Possible to set nice limits. (System wide impact)
setresuid	164	Set real, effective and saved user or group ID. Process needs CAP_SETUID privileges.
query_module	167	Query the kernel for various bits pertaining to modules. Gets info about loaded modules.
nfsservctl	169	Syscall interface to kernel nfs daemon. Possible to manipulate NFS daemon.
setresgid	170	Set real, effective and saved group ID. Process needs CAP_SETUID privileges.
capset	185	Set capabilities. Process needs CAP_SETPCAP privileges.
setresuid32	208	Set real, effective and saved user IDs. Process needs CAP_SETUID privileges.
setresgid32	210	Set real, effective and saved group IDs. Process needs CAP_SETGID privileges.
setuid32	213	Sets the effective user ID of the calling process. Process needs CAP_SETUID privileges.
setgid32	214	Set group identity. Process needs CAP_SETGID privileges.
setfsuid32	215	Set user identity used for file system checks. No permissions to run filesystem check in jail.
setfsgid32	216	Set group identity used for file system checks. No permissions to run filesystem check in jail.

Tabel C.5: **Privileged user system calls**

System call	Nº	Description
pivot_root	217	Change the root file system. Process needs CAP_SYS_ADMIN capability.
sched_setaffinity	241	Set a process's CPU affinity mask. Process needs CAP_SYS_NICE capability.
statfs64	268	Get file system statistics. See `statfs (99)`
fstatfs64	269	Get file system statistics. See `statfs (99)`
mbind	274	Set memory policy for a memory range. Process needs CAP_SYS_NICE capability when MPOL_MF_MOVE_ALL flag is set.
kexec_load	283	Loads a new kernel image to memory. Process needs root permissions.
ioprio_set	289	Set I/O scheduling class and priority. Process can alter itself within the IOPRIO_CLASS_RT - IOPRIO_CLASS_IDLE otherwise it needs CAP_SYS_ADMIN capability. If a process wants to alter another process it needs CAP_SYS_NICE capability.
unshare	310	Disassociate parts of the process execution context. Process needs CAP_SYS_ADMIN capability when CLONE_NEWNS flag is specified.

Tabel C.6: **Privileged user system calls**

Bibliografie

[1] Daniel Bovet and Marco Cesati. Understanding the Linux Kernel. O'Reilly Media, Inc, third edition, 2005.

[2] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. Operating system concepts. John Wiley & Sons, inc, sixth edition, 2003.

[3] Guido van 't Noordende, Ádám Balogh, Rutger Hofman, Frances M.T. Brazier, and Andrew S. Tanenbaum. A secure jailing system for confining untrusted applications. Proc. 2nd International Conference on Security and Cryptography (SECRYPT), 2007. http://www.cs.vu.nl/ guido/publications/ps/secrypt07.pdf.