HÖGSKOLAN
DALARNA

Examensarbete

Kandidatexamen

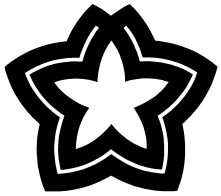
IT-Forensisk undersökning av flyktigt minne

På Linux och Android enheter

Forensic examination of volatile memory under Linux and Android

Författare: Niklas Hedlund
Handledare: Hans Jones
Examinator: Pascal Rebreyend
Ämne/huvudområde: Datateknik
Poäng: 15
Betygsdatum:

Högskolan Dalarna
791 88 Falun
Sweden
Tel 023-77 80 00



HÖGSKOLAN DALARNA

Examensarbete

Program
Digitalbrott och eSäkerhet

Omfattning
15 HP

Namn
Niklas Hedlund

Datum
2013-06-10

Handledare
Hans Jones

Examinator
Pascal Rebreyend

Titel
IT-Forensisk undersökning av flyktigt minne på Linux och Android enheter

Nyckelord
Flyktigt minne forensik Android Linux Volatility LiME

Sammanfattning

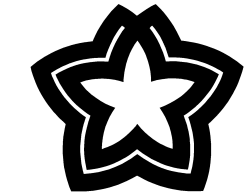
Att kunna gör en effektiv undersökning av det flyktiga minnet är något som blir viktigare och viktigare i IT-forensiska utredningar. Dels under Linux och Windows baserade PC installationer men också för mobila enheter i form av Android och enheter baserade andra mobila opperativsystem.

Android använder sig av en modifierad Linux-kärna var modifikationer är för att anpassa kärnan till de speciella krav som gäller för ett mobilt operativsystem. Dessa modifikationer innefattar dels meddelandehantering mellan processer men även ändringar till hur internminnet hanteras och övervakas.

Då dessa två kärnor är så pass nära besläktade kan samma grundläggande principer användas för att dumpa och undersöka minne. Dumpningen sker via en kärn-modul vilket i den här rapporten utgörs av en programvara vid namn *LiME* vilken kan hantera bägge kärnorna.

Analys av minnet kräver att verktygen som används har en förståelse för minneslayouten i fråga. Beroende på vilken metod verktyget använder så kan det även behövas information om olika symboler. Verktyget som används i det här examensarbetet heter *Volatility* och klarar på papperet av att extrahera all den information som behövs för att kunna göra en korrekt undersökning.

Arbetet avsåg att vidareutveckla existerande metoder för analys av det flyktiga minnet på Linuxbaserade maskiner (PC) och inbyggda system(Android). Problem uppstod då undersökning av flyktigt minne på Android och satta mål kunde inte uppnås fullt ut. Det visade sig att minnesanalys riktat emot PC-plattformen är både enklare och smidigare än vad det är mot Android.



HÖGSKOLAN
DALARNA

Degree project

Programme
Digital crime and eSecurity

Extent
15 HP

Name
Niklas Hedlund

Date
2013-06-10

Supervisor
Hans Jones

Examiner
Pascal Rebreyend

Title
Forensic examination of volatile memory under Linux and Android devices

Keywords
Volatile memory forensic Android Linux Volatility LiME

Abstract

The ability to be able to make a efficient investigation of volatile memory is something that gets more and more important in IT forensic investigations. Partially for Linux and Windows based PC systems but also for mobile devices in the form of the Android or devices based on other mobile operative systems.

Android uses a modified Linux kernel where the modifications exclusively are to adapt it to the demands that exists in a operative system targeting mobile devices. These modifications contains message passing systems between processes as well as changes to the memory subsystems in the aspect of handling and monitoring.

Since these two kernels are so closely related it is possible to use the same basic principles for dumping and analysing of the memory. The actual memory dumping is done by a kernel module which in this report is done by the software called *LiME* which handles both kernels very well.

Tools used to analyse the memory needs to understand the memory layout used on the system in question, depending on the type of analyse method used it might also need information about the different symbols involved. The tool used in this project is called *Volatility* which in theory is capable of extracting all the information needed in order to make a correct investigation.

The purpose was to expand on existing methods for analysing volatile memory on Linux-based systems, in the form of PC machines as well as embedded systems like Android. Difficulties arised when the analysing of volatile memory for Android could not be completed according to existing goals. The final result came to show that memory analysis targeting the PC platform is both simpler and more straight forward then what it is if Android is involved.

Innehåll

Figurer

Tabeller

1 Introduktion

Under de senaste åren har dumpning av det flyktiga minnet blivit något av en branchstandard inom IT-forensiken på PC-datorer. Anledning till det är att det ger möjligheter att få fram information (potentiella bevis) som inte skrivs till permanent lagring. Exempel på sådan information är osparade dokument, öppna nätverksanslutningar, data i urklipp mm. I vissa fall så kan även tidigare inskrivna lösenord återskapas, vilket kan vara väldigt användbart för att komma förbi kryptering.

Då de flesta datorer i världen kör Windows [22] så har analys metoder för Linux (och inbyggda system baserade på detta) halkat lite på efterkälken. En indikation på detta är att alla kurser i vårat program (Digitalbrott och eSäkerhet) som snuddar detta ämne har alla labbar baserade på Windows. För att råda bot på det så har det här arbetet utgjort en djupdykning ner i den värld som är Linux minnesforensik, det täcker insamling (dumpning) av minne samt exempel på hur en IT-Forensiker kan till väga för att analysera detta. Inkluderat var även Android vilket använder sig av en (om än modifierad) Linux-kärna.

1.1 Tidigare forskning

Blandad forskning riktad mot flyktigt minne har gjorts genom åren, mycket av det har fokusert mot Windows medans annan forskning har varit mera öppen. Exempelvis: Bradly Schatz [11] presenterar i sin artikel en metod som fokuserar på hur minnesdumpning kan göras rent generellt över olika plattformar. Genom att skapa en exakt kopia av minnet genom en alternativ modul, i kontrast mot att samla in datan genom operativsystemet På grund av detta beskrivs denna metod som mera säker mot eventuella rootkits som kan finnas på måldatorn.

De senaste åren har forskningen även skiftas över mot ämnen mera specifika för Android. Exempel på en sådan är grundläggande studier i hur livscykeln för information som lagras i det flyktiga minnet på en Android enhet. Dessa studier gjordes genom att meddelanden skickades till och från enheten varefter integriteten av datat studerades. Resultatet visade att möjligheten till att få ut data är mycket goda [15].

Annan senare forskning inriktade sig på möjligheten att under Android kunna välja att starta en annan mjukvara än det riktiga systemet vid uppstart. Funktionen används normalt av tillverkare för att kunna lägga in en återställningsmiljö. Tanken bakom detta var att kunna skapa en generell mjukvara som kan dumpa ut det permanenta minnet (flashminnet) ifrån vilken Android enhet som helst [18]. Metoden är inte applicerbar i det här projektet då den kräver en omstart av enheten vilket är något som skulle förstöra informationen i det flyktiga minnet. Diskussionen är däremot intressant på en teoretisk nivå.

Den mest relevanta forskningen återfinns i [14] där metoder för minnesdumpning under Linux och Android tagits fram. Ett resultat av detta arbete är mjukvaran (*LiME*) som används i det här examensarbetet för att göra minnesdumpningar, se sektion 5.

1.2 Syfte

Syftet med examensarbetet var att vidareutveckla existerande metoder för analys av det flyktiga minnet på Linux-baserade maskiner (PC) och inbyggda system (Android).

Ytterligare ett syfte är att utvärdera de framtagna metoderna ifrån ett forensiskt perspektiv samt att göra en uppskattning av om de skulle kunna fungera på daglig basis i den verkligen världen.

1.3 Mål

Målen för detta arbete är listade nedan;

- Forensisk minnesundersökning under Linux rent generellt (både PC och Android)
- Förstå Linux-kärnans minneslayout
- Att dumpa minne ifrån kärnan
- Genomföra analys av minnet
  - Tolka processstrukturer
  - Extrahera minnesmappningar etc
- Beskriva de problem som finns med att genomföra en analys av flyktigt minne på inbyggda system av typen Android.

Förutom dessa så fanns det även mer avancerade mål som endast kunde uppfyllas när de tidigare målen har avklarats.

- Beskriva Android's processlayout; Hur Dalvik[1] fungerar tillsammans med traditionella Linux processer
- Anpassa metoder till ett forensisk vedertaget tillvägagångssätt
- Utveckla metoder för att extrahera alla tillgängliga forensiska artefakter

1.4 Metod

Läsaren uppmärksammas på att den första delen av målen ovan inte specificerar om det gäller en Linux-kärna som körs på en vanlig PC eller om det är en Android enhet som avses. Det fanns en tanke bakom detta och det är att metoderna först skulle kunna testas emot en PC vilken är klart lättare att ha att göra med än en inbyggd plattform som Android. Framför allt så kan det göras på kortare tid.

Om de testade metoderna fungerade som de ska och visade på ett godtagbart resultat så kunde fokus sedan skiftas över mot Android. Alternativet till detta hade varit att direkt försöka implementera lösningar på Android plattformen Risken med det var att om någonting krånglade så hade det varit mycket svårare att hitta vart det krånglar, detta på grund av den ökade komplexiteten som plattformen medför.

Ett annat val som gjordes tidigt var att uteslutande använda programvaror med öppen källkod. Varför det är något att föredra kan tendera att bli väldigt politiskt och långdraget men i det här fallet så var den avgörande faktorn att de är lätt tillgängliga. Möjligheten att kunna undersöka källkoden och se hur utvecklarna har löst eventuella problem är dock intressant och upplysande, fast i det fallet ändå mera av en bonus. Nackdelen med öppna verktyg är att de ofta inte är lika kompletta och heltäckande som sina proprietära varianter, utan kräver mera handpåläggning.

1.5 Avgränsningar

Arbetet går inte in på djupare detaljer såsom hur minneshantering sker på hårdvarunivå och liknande.

Ingen analys av minne för permanent lagring kom att undersökas i det här examensarbetet Detta innebär att alla referenser till "minne" alltid syftar till flyktigt minne och även enhetens internminne (arbetsminne).

Växlingsfiler, det vill säga innehåll i minnet som har skrivits till disk på grund av utrymmesbrist, kom inte att undersökas. Anledningen till det är att de i våra moderna datorer inte används så ofta, mängden arbetsminne räcker till ändå. Under Android finns det inte någon sådan mekanism.

[1] Mera om Dalvik i sektion 2.3

När det gäller Android på fysiska enheter så gjordes allt jobb endast mot en enhet och det finns därför ingen garanti att det fungerar likadant på alla. Enheten under arbetet var en ZTE Blade med Cyanogenmod 10.1 vilket motsvarar Android 4.2.

Metoder och beskrivningar förutsätter att operativsystemet på IT-forensiken dator är Linux[2]. Detta innebär att inget nämns om eventuella ändringar och speciella verktyg som behövs för att det ska vara möjligt att genomföra allt på en dator med annat operativsystem

1.6 Målgrupp

Denna rapport riktar sig till personer med en IT-teknisk bakgrund och intresse för forensiska frågor.

[2]Distribution är inte viktigt då nödvändiga verktyg med alla rimlighet bör finnas till alla. För exemplen i denna rapport användes dock Arch Linux

2 Allmänt om Linux

Linux är i modern mening ett operativsystem baserat på öppen källkod vilket finns i flera olika varianter (distributioner). Dessa distributioner tycks bli mer och mer populära med tiden. Antagligen har faktumet att det är gratis en del med saken att göra.

Linux är ett så kallat fleranvändarsystem vilket innebär att det kan användas av flera användare, antingen samtidigt eller åtskilt. Nyckeln till detta är flera användarkonton vilket gör att systemet kan delas upp med olika behörigheter för olika delar. För att det ska fungera har varje konto en unik identifierare kallad för användar-ID.

Strikt räknat så refererar termen "Linux" egentligen endast till kärnan vilken skapades av finländaren Linus Torvalds år 1991[16]. Linux-kärnan är en central del i examensarbetet då den är ansvarig för att hantera det fysiska minnet under normal drift. Detta gör att den till högsta grad påverkar utdumpning och analys av det minne som används. Därför följer nedan en kort genomgång av de aspekter av kärnan som är mest väsentliga för det här projektet.

2.1 Processhantering

Läsaren kanske känner till att den vanliga modellen för processhantering innefattar en samling processer vilka alla representerar ett enskilt körande program. En enskild process kan sedan ha en eller flera trådar, beroende på hur många parallella uppgifter programmet behöver kunna göra samtidigt. Denna princip är sann även för Linux, dock är implementationen av det unik. Från kärnans synvinkel är en tråd bara en process som råkar dela minnesområde med en annan process. På grund av detta så tenderar utvecklarna att förkasta den vanliga terminologin och istället kalla bägge för tasks[3] (eller jobb som det blir på svenska) [7, Kapitel 3].

För att hålla ordning på alla processer i systemet så använder sig kärnan av ett så kallat processträd[4]. Anledningen till att det kallas för ett träd är att det utgår från en central process (i Linux kallad "init") som i sin tur startar andra processer. När detta sker så går det ut en gren i trädet vilket knyter den nya processen till den process som startade den ("föräldern"). Om detta får fortgå med processer som startar andra processer så skapas en hierarki av processer som liknar ett upp och nedvänt träd.

Varje gren i trädet innehåller information om ett enskilt jobb; vilken användare och grupp det körs som (behörighet), en unik identifierare ("PID"), identifieraren för föräldern, samt mycket mer som inte är relevant för denna beskrivning. Observera att i fallet med trådning som diskuterades tidigare så kommer dessa jobb att få olika identifierare men de kommer ha samma förälder.

2.2 Minneshantering

Då en stor del av det här projektet cirklar runt arbetsminnet och andra flyktiga entiteter så var det väsentligt att titta närmare på exakt hur Linux-kärnan hanterar det fysiska minnet.

2.2.1 Minneszoner

Linux-kärnan delar in det fysiska minnet i diverse zoner, namnet på dessa och deras innebörd är listade i tabell 1. Detta görs för att upprätthålla portabilitet med arkitekturer som har ett "trasigt" sätt att hantera fysiskt minnet. Ett exempel på ett sådan fall är något som kallas *Direct Memory Access* eller *DMA* vilket är en metod där CPU'n kan komma åt det fysiska minnet direkt utan att kärnan behöver agera mellanhand. Detta används framför allt av drivrutiner för att kommunicera

[3] Fortsättningsvis i den här rapporten så kommer "jobb", "tasks" eller "process" att användas för att beskriva samma sak

[4] Träd i det här fallet syftar inte på den datastruktur som existerar med samma namn, datastrukturen som används för lagring är en cirkulärt länkad lista och inget annat

till och från hårdvaran. Exempelvis så använder nätverksdrivrutiner *DMA* för att skriva data till kärnans databuffrar.

ZONE_DMA	Område där DMA kan utföras
ZONE_DMA32	Område för 32bit CPU'er kan utföra DMA
ZONE_NORMAL	Normalt minne som är permanent inlänkat i kärnan
ZONE_HIGHMEM	"Högt" minne som länkas in i kärnan vid behov, ej permanent

Tabell 1: Minneszoner i Linux-kärnan

Ett exempel på hur dessa zoner kan vara uppdelade ser vi i den klassiska *x86* arkitekturen som av historiska skäl har vissa begränsningar. Till exempel så kan *ISA*[5] enheter endast utföra *DMA* till de första 16MB av minnet. Detta innebär att **ZONE_DMA** kommer att bli 0 till 16MB. **ZONE_HIGHMEM** kommer av historiska skäl att bli allt tillgängligt minne över 896MB (för *x86*-arkitekturen). Denna gräns är inte relevant på nyare arkitekturer men då Linux ska fungerar även på äldre så måste utvecklarna ta hänsyn till saker som det. **ZONE_NORMAL** blir det minne som är kvar, vilket då blir 16MB - 895MB.

Väldigt annorlunda blir det om vi tittar på *x86_64* arkitekturen (det vill säga 64-bit) där vi endast har **ZONE_DMA** och **ZONE_NORMAL**. Detta då kärnan med denna CPU kan hålla allt minne permanent inlänkat och det därmed inte finns något behov för **ZONE_HIGHMEM**. [7, Kapitel 12].

2.2.2 Processminneslayout

Linux-kärnan använder sig av en platt minneslayout vilket skiljer sig ifrån den segmenterade modellen som används i andra operativsystem. I en platt modell så använder en process alla minnesadresser mellan noll och det högsta giltiga värdet, vilket är $2^{bits} - 1$ och *bits* är antingen 32 eller 64 beroende på arkitektur.

En process har normalt varken åtkomst eller tillgång till hela adressrymden. Vilka delar (eller vilka mappningar[6] som det egentligen är) den har åtkomst till beror på de rättigheter som har satts på den specifika mappningen. För att lista alla mappningar som är aktuella för en process så kan verktyget *pmap* användas[7]. I listning 1 visas en körning av detta verktyg på ett väldigt enkelt testprogram som inte gör något annat än att sova.

Den första raden är *text*-sektionen i den aktuella binären. *Text* innehåller den körbara maskinkoden vilket förklarar varför denna är markerad som "exekverbar" ("x" i mitten fältet). Den andra raden är *data*-segmentet vilket innehåller värdet på variabler. Rättigheterna för denna är satt till "läsning" och "skrivning" ("r" och "w" respektive) vilket stämmer då programmet behöver kunna både läsa och skriva till sina variabler. Observera att det inte finns någon "x" flagga vilket gör att det inte går att exekvera något från *data*-segmentet. På adress *0x7fff6ad6b000* så börjar *stack*-segmentet för processen. Även här så är det endast läsning och skrivning som gäller vilket skyddar mot så kallad "buffer overflow exploits" där maskininstruktioner trycks upp på stacken och sedan exekveras. [7, Kapitel 14]

Observera att varje enskild mappning utgör ett så kallat virtuellt minnes område och representeras i kärnan av en *vm_area_struct* (se bilaga A).

[5] *Industry Standard Architecture*, är en äldre kommunikations standard för enheter direkt anslutna till moderkortet

[6] En mappning kan beskrivas som en intern referens i kärnan mellan en minnesadress i det virtuella processminnet till en minnesadress i det fysiska minnet

[7] Samma information kan även ses i */proc/[PID]/maps, vilket är en vanlig textfil*, *pmap* gör det dock lite mera läsbart

```
[nojan@ahsoka ~]$ pmap 15043
15043:   ./a.out
0000000000400000      4K r-x--  /home/nojan/Dropbox/Forensic2/Project/Report/a.out
0000000000600000      4K rw---  /home/nojan/Dropbox/Forensic2/Project/Report/a.out
00007fee47fcc000   1680K r-x--  /usr/lib/libc-2.17.so
00007fee48170000   2044K -----  /usr/lib/libc-2.17.so
00007fee4836f000     16K r----  /usr/lib/libc-2.17.so
00007fee48373000      8K rw---  /usr/lib/libc-2.17.so
00007fee48375000     16K rw---    [ anon ]
00007fee48379000    132K r-x--  /usr/lib/ld-2.17.so
00007fee4856a000     12K rw---    [ anon ]
00007fee4859a000      4K r----  /usr/lib/ld-2.17.so
00007fee4859b000      4K rw---  /usr/lib/ld-2.17.so
00007fee4859c000      4K rw---    [ anon ]
00007fff6ad6b000    132K rw---    [ stack ]
00007fff6adff000      4K r-x--    [ anon ]
ffffffffff600000      4K r-x--    [ anon ]
 total             4068K
```

Listning 1: Exempel på minnesmappning

2.3 Android i ett nötskal

Som nämnts tidigare så är Android baserat på en Linux-kärna och det använder typiska Linux-egenskaper såsom användar-ID och processer. Det är även upp till kärnan att schemalägga körningen av processer. På denna grund har det dock byggts vidare med ramverk och säkerhetslösningar som gör det till det mobila operativsystem det är idag.

Arkitektonisk sett är Android uppbyggt i fyra lager (se Figur 1), vilka är;

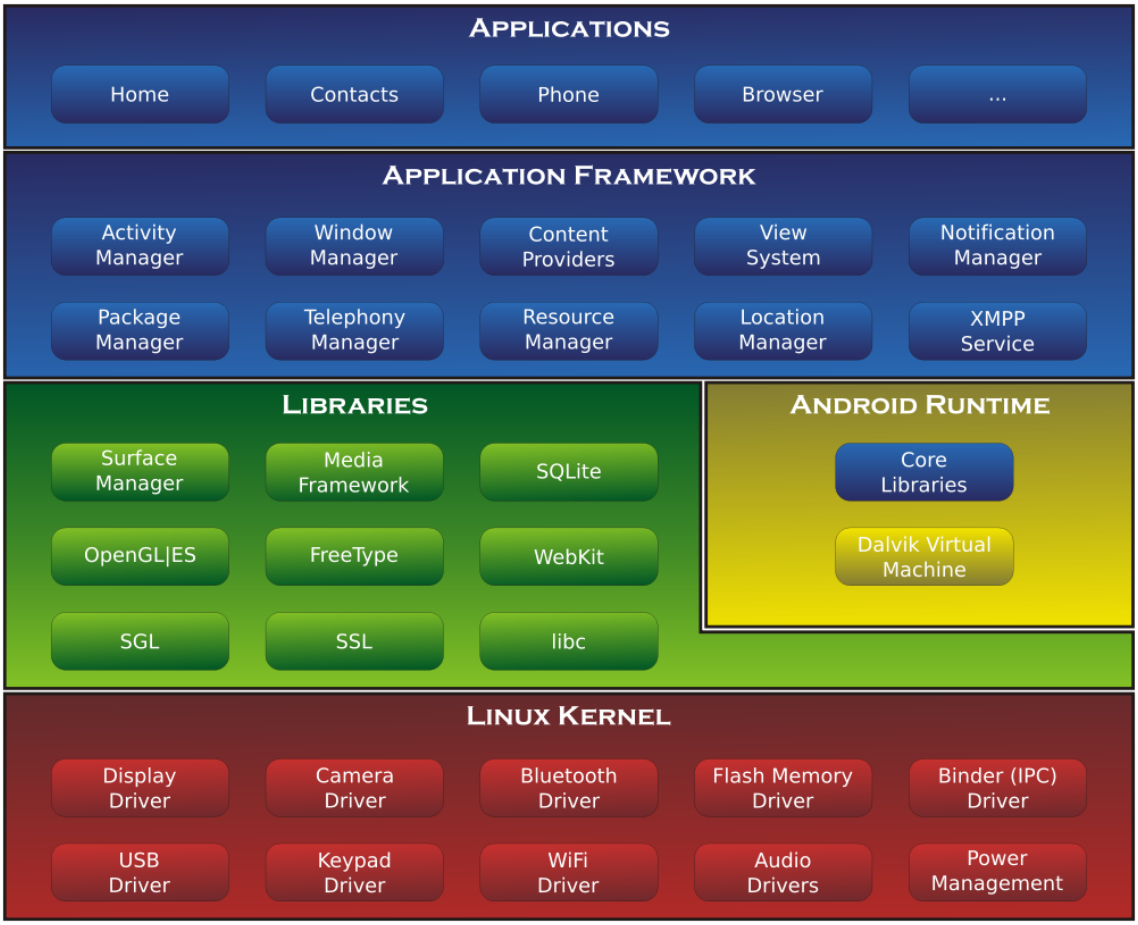
Lager 1
: Linux-kärnan med tillhörande drivrutiner (moduler) för sådana saker som Touchscreen, WLAN, Ljud osv

Lager 2
: Bibliotek och Android Runtime, inkluderar bland annat "Dalvik" (se sektion 2.3.2) och systembibliotek.

Lager 3
: Android's applikation ramverk. Innehåller de nödvändiga delarna i ramverket som gör att de överliggande applikationerna ("appar" i folkmun) kan fungera. Exempel på en del i ramverket är "Location Manager" vilken har till uppgift att tillhandahålla geografiska positionsangivelser på begäran.

Lager 4
: Klientapplikationer, inkluderar allting från webbläsare till applikationer för att ringa och skicka SMS. Kort sagt kan man säga att allt som kan ses på skärmen hamnar i det här lagret.

Lager ett och två består av källkod skriven i C medan ovanstående lager använder sig av Java som programspråk. Java är ett så kallat "hanterat" språk där källkoden kompileras till ett slags mellansteg mellan källkod och maskinkod. För att denna kod ska kunna köras så måste miljön ha en speciell tolk för detta. Under Android fylls denna roll av ett program som heter Dalvik, vilken skapar en virtuell miljö där en enskild applikation körs. Varje enskild instans av en sådan virtuell miljö körs under en egen process (Linux terminologi) med ett unikt användar-ID och med en egen instans av Dalvik. Det finns vissa undantag där flera applikationer från samma utvecklare kan komma att köras tillsammans, men i de allra flesta fall så fungerar det som beskrivits ovan.



Figur 1: Lageruppbyggnaden i ett Android system (bild från wikimedia)

2.3.1 Skillnader mellan Linux och Android's kärna

För att anpassa Linux-kärnan till att fungera bra tillsammans med resten av Android så behövde utvecklarna göra vissa ändringar, dessa listas nedan.

Sättet som Android är uppbyggt på kräver att olika appar (processer i kärnan) kan kommunicera med varandra på ett enkelt sätt, till exempel sker detta när ett så kallat "intent"[8] skickas. I kärnan är det implementerat som ett *Inter process call* (*IPC*) ramverk kallat "Binder", vilket i sig är en vidareutveckling av "OpenBinder" som utvecklades av *Palm Inc* [9]. Versionen som används i Android är dock omskriven för att fungera bättre med resten av systemet men även av licensskäl.

Det finns situationer där olika processer kan behöva komma åt varandras minne (exempelvis appar från samma utvecklare där vissa komponenter delas). Detta hanteras av en komponent kallad "ashmem", vilket är en mekanism i kärnan som tillåter minnesdelning mellan flera processer via vanliga filanrop till en speciell filnod. Denna metod är vanlig i Unix världen och mekanismen är dessutom fullt åtkomlig ifrån "user space".[13]

Standard Linux-kärnan har en mekanism som identifierar och automatiskt stänger ner processer som skenar iväg och börjar äta upp allt minne i systemet. Denna kallas för *Out of memory killer* (*OOM*). Den är dock inte inblandad i de fall där det fysiska minnet börjar ta slut, utan där tar en annan mekanism över som växlar ut detta minne till den permanenta lagringen (hårddisk i de flesta fall). Denna process kallas för "swapping". Under Android däremot så finns det ingen riktig möjlighet eller anledning att hantera det på det sättet. Därför har *OOM*'en under Android

[8] En typ av signal som kan användas för att kommunicera inom eller mellan appar

designats så att den istället stänger ner processer som inte har använts på länge för att frigöra minne i systemet, en lösning som fungerar mycket bättre tillsammans med det användningsmönster som gäller för en mobil enhet.

Andra modifikationer som har gjorts i Androids kärna har och göra med strömbesparing och vilolägen. En av de primära komponenterna i detta är något som kallas för *wakelocks*, dessa är en mekanism som används för att förhindra att hela eller delar av systemet går ner i vila. Exempelvis kan en app för navigering begära en *wakelock* som säger att skärmen inte får stängas ner förens appen tillåter det. Det ska även tilläggas att *wakelocks* även kan användas på andra nivåer i systemet, såsom inifrån kärnan eller av systemtjänster.

Förutom detta så har givetvis andra ändringar tillkommit i form av drivrutiner för saker som GPS, Touchscreen och annat som behövs för att kommunicera med mobilnätet[1].

2.3.2 Android processlayout

Som nämnts tidigare så startar ett Linux-system genom att processen "Init" skapas av kärnan och denna process skapar i sin tur ytterligare processer som utgör resten av systemet. En av de processer som går igång när ett Android-system startar är en entitet som kallas "Zygote". Denna entitet agerar som ett slags understeg till den vanliga "Init"-mekanismen och har till uppgift att skapa instanser av Java motorn "Dalvik" när dessa behövs.

Nya instanser i det här fallet innebär inte att en ny "Dalvik"-process läses in och startas upp. "Zygote" jobbar istället så att den läser in "Dalvik" när den först startas för att sedan skapa kloner[9] av sig själv när en ny instans behövs [2]. Genom detta så behöver "Dalvik" endast startas[10] en gång! Exempel på detta är när användaren öppnar en ny app, alternativt om någon systemtjänst (systemtjänst i det här fallet syftar på högre tjänster i ramverket och inte någon lågnivå tjänst i kärnan) startas i bakgrunden.

Om man undersöker en hur en app som körs i systemet representeras i kärnan så kommer den köras som en egen process vars namn är en avkortad variant av det namn som appen har i Android[11]. Som förälder kommer processen att ha "Zygote" processen. Den kommer även köras med ett unikt användar-ID. Specialfallet av detta är om två appar har samma utgivare[12]. I detta fall kommer processerna köras med samma användar-ID, observera att det fortfarande rör sig om skilda processer. Det faktum att de får samma användar-ID gör att de under vissa förutsättningar kan komma åt varandras minne via "ashmem"[13]. Ett exempel på en processlistning ifrån en Android-enhet kan ses i listning 2, listan genererades av verktyget som beskrivs i sektion 7.

Notera att även om en användare skulle starta en navigations-app vilket kräver att positioneringstjänsten måste startas i bakgrunden, så kommer processen för den tjänsten att startas av "Zygote" och inte av navigations-appen.

[9] Sker genom systemanropet *fork*
[10] Med startas menas i det här fallet att läsas in i minnet för att sedan exekvera
[11] Dessa är på klassiskt Java manér en omvänd DNS stil, exempelvis; com.google.android.location-manager
[12] Mera specifikt signerade med samma nyckel
[13] Se sektion 2.3.1

```
PID Command UID Parent State
+ 1 init 0 0 TASK_INTERRUPTIBLE
+ 2 kthreadd 0 0 TASK_INTERRUPTIBLE
+ 3 ksoftirqd/0 0 2 TASK_INTERRUPTIBLE
+ 4 events/0 0 2 TASK_INTERRUPTIBLE
+ 5 khelper 0 2 TASK_INTERRUPTIBLE
+ 6 async/mgr 0 2 TASK_INTERRUPTIBLE
 --- SNIP ---
+ 87 binder 0 2 TASK_INTERRUPTIBLE
+ 88 l2cap 0 2 TASK_INTERRUPTIBLE
+ 89 krfcommd 0 2 TASK_INTERRUPTIBLE
+ 90 khsclntd 0 2 TASK_UNINTERRUPTIBLE
+ 91 ueventd 0 1 TASK_INTERRUPTIBLE
+ 92 mmcqd 0 2 TASK_RUNNING
+ 93 flush-31:5 0 2 TASK_INTERRUPTIBLE
+ 94 servicemanager 1000 1 TASK_INTERRUPTIBLE
+ 95 vold 0 1 TASK_INTERRUPTIBLE
+ 96 netd 0 1 TASK_INTERRUPTIBLE
+ 97 debuggerd.bin 0 1 TASK_INTERRUPTIBLE
+ 98 rild 1001 1 TASK_INTERRUPTIBLE
+ 99 surfaceflinger 1000 1 TASK_INTERRUPTIBLE
+ 100 zygote 0 1 TASK_INTERRUPTIBLE
+ 101 drmserver 1019 1 TASK_INTERRUPTIBLE
+ 102 mediaserver 1013 1 TASK_INTERRUPTIBLE
+ 103 installd 1012 1 TASK_INTERRUPTIBLE
+ 104 keystore 1017 1 TASK_INTERRUPTIBLE
+ 106 qmuxd 1001 1 TASK_INTERRUPTIBLE
+ 107 akmd2 0 1 TASK_INTERRUPTIBLE
+ 110 adbd 2000 1 TASK_INTERRUPTIBLE
 --- SNIP ---
+ 1076 droid.gsf.login 10001 100 TASK_INTERRUPTIBLE
+ 1197 ndroid.settings 1000 100 TASK_INTERRUPTIBLE
+ 1333 android.vending 10027 100 TASK_INTERRUPTIBLE
+ 1386 LocationService 10054 100 TASK_INTERRUPTIBLE
+ 1394 gle.android.gms 10001 100 TASK_INTERRUPTIBLE
+ 1415 gle.android.gcm 10001 100 TASK_INTERRUPTIBLE
+ 1581 onFriendService 10054 100 TASK_INTERRUPTIBLE
+ 1608 android.browser 10044 100 TASK_INTERRUPTIBLE
+ 1664 apters.calendar 10007 100 TASK_INTERRUPTIBLE
+ 1685 droid.apps.maps 10054 100 TASK_INTERRUPTIBLE
+ 1807 d.apps.uploader 10000 100 TASK_INTERRUPTIBLE
+ 1826 facebook.katana 10053 100 TASK_INTERRUPTIBLE
+ 1941 oadcastreceiver 10038 100 TASK_INTERRUPTIBLE
+ 2035 rew.apollo:main 10047 100 TASK_INTERRUPTIBLE
+ 2072 sh 2000 110 TASK_INTERRUPTIBLE
+ 2090 droid.gallery3d 10028 100 TASK_INTERRUPTIBLE
+ 2119 sh 0 2072 TASK_INTERRUPTIBLE
+ 2143 insmod 0 2119 TASK_RUNNING
```

Listning 2: Processlistning ifrån en Android-enhet

3 Nödvändig behörighet

För att kunna göra en lyckad undersökning av det fysiska minnet så krävs ökade privilegier. Under Linux(Android) så sker detta genom att kommandot måste köras av användaren "root" vilket är det användarkonto som har fulla behörigheter. På en PC är det inget större problem då denna inte är lika "nerlåst" som ett inbyggt system. Det enda som kan orsaka problem är att IT-Forensikern (här efter kallad "utredaren") i fråga måste ha tillgång till de relevanta lösenorden för systemet.

Exakt hur man kan gå tillväga för att få fram dessa lösenord är dock utanför ramen för den här rapporten.

3.1 Några ord om rootning

För att nå nödvändig behörighet på en Android-enhet så måste mjukvaran modifieras så att åtkomst till root-konton tillåts. På Android-språk kallas detta att "roota" enheten. Hur detta görs är mer eller mindre unikt ifrån enhet till enhet och kräver därför ett visst mått av efterforskning. Oftast sker detta genom att en sårbarhet i den specifika versionen av Android utnyttjas.

En annan metod som endast fungerar om enhetens bootloader (inbyggd mjukvara som hanterar enhetens uppstart) inte är låst, är att "flasha"[14] in nödvändiga komponenter. Detta sker genom ett speciellt läge i bootloadern (download mode / fastboot, exakt namn varierar mellan tillverkare) där enheten accepterar data via USB eller från minneskort och tillåter att detta skrivs direkt till telefonens interna lagringsminne (flashminne).

En mer automatiserad metod för "rootning" är mjukvaran *OneClickRoot*, vilket enligt deras webbsida tycks stödja de flesta versioner på marknaden. [8] Mjukvaran har dock antagligen inte fungerande stöd för de nyaste enheterna på marknaden. Eftersom det tar tid för "communityt" att lista ut hur rootningen ska gå till, samt för utvecklarna att implementera det i programmet.

Värt att notera är att enheter även i vissa fall "rootas" av konsumenter fast då i andra uppsåt som till exempel; att kunna uppgradera till en nyare version av Android (äldre enheter hamnar långt efter i utvecklingen), installera nya funktioner eller komma runt begränsningar som kan finnas från operatörens sida.

[14] Ett ord som i det här fallet kan översättas till "skriva till flashminnet"

4 Forensisk aspekter vid undersökning

En av grundbultarna i modern IT-forensik är att undersökningar ska göras på ett sådant sätt att originalet inte påverkas på ett onödigt sätt. Det måste även i efterhand gå att styrka att materialet vid slutet av undersökningen är lika som när undersökningen inledes. Detta sker generellt genom att en så kallad hash-summa (kan ses som fingeravtrycket av en fil) tas på filen, vilken sparas i handlingarna. Genom att senare ta en ny hash-summa och jämföra så kan autenticiteten styrkas.

Detta ställer inte till något större problem när det rör sig om permanent lagringsmedia såsom hårddiskar eller USB-minnen. Eftersom deras innehåll inte ändras om de ligger frånkopplade, då dessa är av så kallad icke-flyktig typ. Om undersökningen istället fokuserar på flyktigt minne så uppstår en hel hög nya problem att ta hänsyn till.

Flyktigt minne är som namnet antyder icke permanent. Detta innebär att minnes-cellerna är i konstant behov av spänning för att integriteten på det lagrade datat ska kunna garanteras. Om spänningen skulle försvinna, vilket i de vanligaste fallet är när enheten stängs av så kommer innehållet i minnet att försvinna. Forskning har visat att viss data kan återskapas alternativt fortfarande finnas kvar, [10] men det kan mer eller mindre anses som förlorat.

Även insamlingen vållar vissa problem vilka kan illustreras genom att återigen jämföra mot en hårddisk. När en utredare vill göra en insamling av datat ifrån en hårddisk kan denna montera ut den ur den misstänktes dator för att strömsätta den i sin egen utrustning och göra en kopia av datat utan att på något sätt påverka det data som finns på disken. Förutsatt att inte disken används emellan så kan en ny spegling göras en lång tid senare och kopian kommer vara identisk med den tidigare.

Detta skiljer sig från den procedur som kan användas för att samla in flyktigt minne, för det första så kan inte hårdvaran plockas ut. För det skulle krävas att strömmen bryts, vilket som diskuterades innan inte är någon bra idé. Lösningen på detta är att insamlingen måste ske på den hårdvara där minnet hör hemma. För att det ska vara möjligt så måste nödvändiga verktyg på något sätt injiceras till den maskinen och de måste även köras i dess regi. Detta skapar ett problem i det att det flyktiga minnet i regel även fungerar som arbetsminne för enheten, innebörden blir att det verktyg som används för att samla in minne måste läsas in i arbetsminnet innan det kan exekveras. Det innebär att programmet kommer att samla in data om sig själv tillsammans med resten av minnet. Det finns givetvis också en möjlighet att gammal borttagen data kan ha blivit överskriven när programmet lästes in.

När minnet väl har dumpats till en fil så kan samma procedur användas för lagring av filen som om det hade varit en dump av en hårddisk. Det vill säga; skapa en checksumma, dokumentera och spara filen på ett säkert medium där den inte riskeras att korrupteras.

4.1 Rootnings påverkan

Som nämnt tidigare så kräver en effektiv undersökning av en Android enhet att enheten först rootas. Den fråga som då uppkommer är; “Kan försöken från utredarens sida att uppnå root i själva verket förstöra bevisen till en sådan grad att det inte längre är värt att fortsätta undersökningen?”.

Det första konstaterandet som kan göras är att alla metoder eller instruktioner för rootning som innehåller raden; “Starta om telefonen”, innebär att det ej är lämpligt i en undersökning som inriktar sig på flyktigt minne. Tyvärr så verkar de flesta metoder som finns i den samlade skaran av Android-utvecklare bygga på principen att man gör en ändring till enhetens mjukvara för att sedan starta om den och låta den läsa in de nya ändringarna. Det bör även nämnas att detta innefattar skrivning till enheten permanenta minne vilket kan förstöra data där, vilket dock inte är relevant för den här diskussionen.

Om en omstart av enheten inte kan undvikas så bör utredaren åtminstone se till att det rör sig om en så kallad “soft reset” vilket innebär att omstarten sker genom mjukvara och inte genom att

strömmen bryts för att sedan slås till igen, vilket är fallet i en "hard reset". Detta då denna metod har störst chans att bevara så mycket av minnets innehåll som möjligt [10].

5 Minnesdumpning

Nedanstående information gäller för Linux-kärnan över lag och är därför lika sant för PC som det är för Android.

Innan någon som helst analys kan göras måste innehållet i interminnet dumpas till en fil. Historiskt sett har detta varit enkelt då det fysiska minnet har varit åtkomligt via två pseudo enheter kallade */dev/mem* och */dev/kmem*, vilka gav åtkomst på två sätt; */dev/mem* gav åtkomst till det fysiska minnet till skillnad från */dev/kmem* som ger åtkomst till hela det virtuella minnet (växlingsfil inkluderat).

Nackdelen med denna metod är att det endast ger åtkomst till en begränsad del av minnet (ZONE_NORMAL) och därmed på moderna datorer missar en stor del. Dessutom har trenden under de senaste åren varit att olika distributioner har inaktiverat skapandet av dessa enheter, och åtkomsten till minnet begränsats från kärnans sida. Ett exempel på detta kan vi se i de medföljande anteckningarna till "patchen" i [17].

För att komma runt detta måste dumpning ske ifrån det så kallade kernel-land (i kontrast mot user-land där alla användarapplikationer körs) då kärnan i sig har fri åtkomst till minnet. För att utföra detta måste vi be kärnan köra en speciell kod som gör själva dumpningen, vilket realiseras genom en så kallad "kernel module".

Även fast Linux-kärnan tillhör den monolitiska familjen (monolitisk i det avseendet att hela kärnan körs i samma adressrymd) har den möjlighet att dynamiskt länka in moduler under körning och det är alltså den möjligheten som utnyttjas i det här fallet.

LiME[6] (tidigare kallad *dmd*) är en modul designad för att dumpa minne ifrån en aktiv kärna och är det verktyg som kommer användas. En stor fördel med det verktyget är att det fungerar både för PC och Android. Tillvägagångssätter för att kompilera den skiljer dock mellan plattformarna. För mera detaljerade instruktioner för kompilering se bilaga D.

Modulen som sådan tar två argument (egentligen tre men endast de första två är väsentliga) vilka är;

format specificerar vilket format LiME ska använda på den slutgiltiga dumpfilen, de giltiga valen är "raw", "padded" och "lime".

path är sökvägen till vart dumpfilen ska sparas. Stöd finns även för att skicka dumpad data över nätverket istället för att spara ner det på lokal disk.

Skillnaden mellan de olika formaten är framförallt hur de hanterar fragmenterat minne, detta kan uppkomma antingen av segment som inte används, trasiga celler eller åtkomsträttigheter. Om formatet "raw" används så blir det en exakt kopia över hur *LiME* ser minnet. Detta format kan alltså komma att hoppa över delar och det går inte att garantera att dumpen är en exakt kopia av hela minnet. Formatet "padded" kommer att ersätta dessa områden med nollor varav det är garanterat att den interna strukturen kommer att vara korrekt. Det sista formatet kallat "lime" är ett lite speciellt format i det att varje dumpat minnesområde börjar med en header som beskriver vart i det fysiska minnet den specifika sektionen existerade i det fysiska minnet.

5.1 I PC miljö

Förutsatt att *LiME* har blivit korrekt kompilerad för den aktuella kärnan så är användandet av modulen förhållandevis enkelt, då *LiME* är skrivet så att den automatiskt börjar dumpa minne så snart den laddas in i kärnan. För att ladda in modulen används med fördel, Linux standardverktyg *insmod*. Den insatte känner även eventuellt till *modprobe* vilket generellt gör ett bättre jobb, den kräver dock att modulen är kopierad till rätt plats i filsystemet vilket medför onödig påverkan på systemet som ska undersökas. Användadet av *insmod* gör att det istället är möjligt att använda någon form av borttagbart lagringsmedium.

Rent generellt används *insmod* enligt följande;

```
# insmod <sökväg till modul> <arg1>=<val1> <arg2>=<val2> ... <arg-n>=<val-n>
```

Notera att det i kommandot finns möjlighet att ange argument, dessa skickas automatiskt vidare till den laddade modulen. Vilket är hur *LiME* kommer få information om de angivna argumenten (listade i den tidigare sektionen). Det inledande tecknet ("#") är en indikation att kommandot ska köras som användaren "root" och är alltså inte något som ska skrivas in. Ett exempel med riktiga värden skulle kunna se ut något sådant här;

```
# insmod /run/media/portable/lime.ko format=lime path=/run/media/portable/memdump.lime
```

Kommando kommer att blockera (köras) tills det att dumpningen har slutförts för att sedan återvända till prompt. Värt att notera är att även då dumpningen är slutförd kommer fortfarande modulen var inladdad i kärnan. För att ladda ur den används kommandot *rmmod*. Användningen av det kommandot är så lätt som att köra;

```
# rmmod lime
```

5.2 Under Android

Metoden för att göra samma sak under Android är snarlik mot PC, dock med några viktiga skillnader.

Ett exempel är en sådan enkel sak som hur modulen flyttas ifrån systemet där den kompilerades till Android enheten kan bli lite knepigt. Möjligheten finns att lägga allt som behövs på ett minneskort och sedan ansluta detta till minneskortsläsaren på enheten. Detta kräver dock realistiskt sett en omstart av enheten. Vilket skulle vara ett effektivt sätt att kasta bort all flyktig information som eventuellt skulle kunna vara av intresse. Ett annat alternativ är att använda det minneskort som sitter i telefonen, vilket är antagligen den lättaste och mest använda metoden. Nackdelen är att eventuell viktig bevisning på minneskortet (som raderade filer, mappar etc) kan komma att skrivas över. Därför är det viktigt att göra en kopia av minneskortet innan modulen kopieras över.

För att undvika att minneskortet avmonteras i enheten (vilket är fallet om det så kallade "USB-Lagring"[15] alternativet används) så kan verktyget *adb* användas. *ADB*[16] ska utläsas som "Android Debugging Bridge" och är en del av de fritt tillgängliga utvecklingsverktygen (Android SDK) som är tillgängliga för Android. En av funktionerna som *adb* stödjer är att överföra filer till och från enheten och detta kan användas för att kopiera över modulen till enheten ("push"), exempel;

```
[PC] adb push lime.ko /sdcard/lime.ko
```

Detta kommer att kopiera den lokala filen (ur PC hänseende) "lime.ko" till enhetens "/sdcard/lime.ko", "/sdcard/" vilken är sökvägen till (i de flesta fall) minneskortets monteringspunkt.

Efter att modulen har kopierats över är nästa steg att få ett skal på enheten så att nödvändiga kommandon för att ladda in modulen kan användas. Detta kan göras med hjälp av *adb*. Se nedan för exempel på hur kommandoföljden kan se ut;

```
[PC] adb shell
[ANDROID] su
[ANDROID] insmod /sdcard/lime.ko "path=/sdcard/dump.lime format=lime"
[ANDROID] exit
[ANDROID] exit
```

Notera att ett av kommandona som används ovan benämns *su*, vilket är en förkortning av "Super user" och är kort förklarat ett kommando som gör användaren inloggad som "root". Kontrasterande så behövs här inget lösenord på det sättet som är fallet på en PC. Något att däremot ta hänsyn till är att *su* (i 99 fall av 100) inte finns som standard ifrån fabrik utan är något som läggs in när telefonen "rootas" [20].

[15] Ett sätt att komma telefonens minneskort från en dator via en USB-kabel, beter sig då som om det vore ett anslutet USB-minne

[16] För att *ADB* ska fungera så måste det vara aktiverat på enheten. Inställningen i fråga finns under; Settings > Developer options > USB debugging. Eller motsvarande på andra språk

Ett annat problem som kan uppkomma under Android är att *insmod* kommandot har vissa egenheter, exakt vilka de är skiljer sig från enhet till enhet men några verkar vara;

- Sökvägen till modulen måste vara absolut, det går alltså inte att bara skriva namnet även om man står i samma mapp.
- Argumenten kan behöva bäddas in i citationstecken.

När dumpningen är slutförd så måste dumpen kopieras tillbaka till undersökningsdatorn. Det görs enklast genom att återigen använda *adb* men den här gången för att kopiera från enheten till datorn ("pull").

```
[PC] adb pull /sdcard/dump.lime
```

6 Minnesanalys

Minnesanalys är processen där en dump av en enhets minne analyseras och tolkas för att kunna återskapa information om det dumpade systemets status vid tillfället för dumpningen. Detta är något som verkligen passar in på frasen "bringa ordning ur kaos", detta då det är precis vad en minnesdump ser ut som för blotta ögat; kaos. Det är dock inte så konstigt då minnet är organiserat för att kunna läsas lätt av systemet och inte av en människa. Det krävs därför speciella program och metoder för att kunna tyda kaoset och göra om det till något som vi människor kan förstå. De vanligaste metoderna är listade nedan;

Symbolbaserat
: Den symbolbaserade metoden bygger på att analysatorn vet vart en viss datastruktur börjar i minnet varefter den kan tolka informationen. Om det till exempel rör sig om en länkad lista (en vanlig datastruktur för att lagra sekvensiell data) skulle analysatorn börja med att läsa det första elementet för att sedan följa referensen till det andra elementet. Och så vidare.

  Symbolbaserade metoder funderar även på icke länkade strukturer så som heltal, strängar eller flyttal.

  En nackdel med en symbolbaserad metod är den kommer att missa data som är lagrad på annan plats än vad som finns med i symboltabellerna. Detta kan exempelvis vara "malware" eller annan illasinnad programvara som försöker gömma sig i systemet.

Signaturbaserad
: Den här metoden är raka motsatsen mot den symbolbaserade. Istället för att utgå ifrån kända symboler så innebär den här metoden att analysatorn söker efter kända signaturer för olika typer av data. Fördelen med det är att det inte spelare någon roll om datat av intresse är inlänkat i någon struktur eller inte. Analysatorn kommer ändå att hitta det förutsatt att dess signatur kan definieras.

6.1 Volatility

Volatility[19] är ett program för att göra minnesanalys utifrån en tidigare existerande minnesdump och är skrivet i *Python*. Programmet som sådant är helt öppet[17], med innebörden att vem som helst får förändra eller använda det. *Volatility* stödjer analys av en stor mängd system, såsom; Samtliga Windows versioner (både 32 och 64 bit), Mac och Linux. Sedan version 2.3 finns även stöd för att analysera minnesdumpar från Android system.

Volatility använder "moduler" för att implementera specifika uppgifter, som till exempel att extrahera en listning av alla processer. Denna metod innebär att nya funktioner (analysmetoder) kan läggas till de olika plattformarna utan att själva kärnan i programmet måste ändras varje gång, vilket är något som leder till en väldigt ren och lättläst kod.

Majoriteten av modulerna i *Volatility* använder sig av en symbolbaserad analysmodell, vilket förklarades i sektionen ovan. För att det ska fungera så måste *Volatility* veta vart de olika symbolerna börjar (börjar i det här fallet syftar på vilken minnesaddress de finns lagrade på). Detta problem löser *Volatility* genom att använda en så kallad profil för olika system vilken tillåter *Volatility* att generera en tolkning av minnesdatan. För Windows-system är profilerna fördefinierade internt, detta av den enkla anledningen till att symbolerna ändras så sällan (Service pack eller nya Windows versioner).

För Linux och Android är inte symboltabellerna lika koncisa utan där skiljer det sig ifrån kärna till kärna och även mellan olika kompilering av samma kärn-version. Där använder sig därför *Volatility* av lösa profil-filer vilket gör det lättare för användare att lägga till nya profiler efter behov.

En profil för Linux är egentligen ett "zip"-arkiv innehållandes två filer. Den första av dessa är symboltabellen för kärnan vilken vanligen heter "Symbol.map". Filen genereras av kompilatorn

[17] GNU GPL v2 är licensen som används

när kärnan kompileras och är egentligen inget annat än en listning av alla symboler samt vilken minnesadress dessa återfinns på, se listning 3 för ett exempel på hur en del av filen kan se ut.

```
c005019c t fault_in_user_writeable
c00501ec t fixup_pi_state_owner
c00503b4 T handle_futex_death
c0050470 T exit_pi_state_list
c00505d4 T exit_robust_list
c00506d0 t refill_pi_state_cache
c005074c t lookup_pi_state
c00509e4 t futex_lock_pi_atomic
c0050b18 t fixup_owner
c0050c00 t free_pi_state
c0050cc4 t unqueue_me_pi
```

Listning 3: Utdrag ur symboltabell

Den andra filen är en så kallad "dwarfdump" gjord på en speciell kärn-modul som kommer med *Volatility*. Modulen i sig gör absolut ingenting utan dess enda mening är att länkas mot så många symboler i kärnan som möjligt. För att detta ska fungera måste därför denna speciella modul byggas emot den aktuella kärnan och nödvändiga headers' måster därför finnas tillgängliga. Det gäller alltså precis samma sak som när *LiME* kompilerade (vilket beskrevs tidigare). För att bygga profilen är det dock inte den kompilerade modulen som behövs utan "dwarfdumpen" av den. En "dwarfdump" är ett sätt att extrahera ut felsökningsinformation ifrån en redan kompilerad binärfil. Anledning till att det används här är dock för att detta även inkluderar information om symbollänkningar.

Dessa två filer innehåller alltså bägge information om olika symboler i kärnan och det *Volatility* gör är att den kombinerar information från bägge dessa till en jättetabell som den sedan använder för att analysera minnet [21].

6.1.1 Profiler för Android

Profiler för Android följer samma mönster och har samma innehåll som de för Linux. Det finns dock vissa saker som skapar problem. Det mest signifikanta problemet med Androids är (i kontrast mot PC) nerlåsning av systemet vilket gör det svårare att komma åt filer som behövs för att skapa profilen. Ett exempel på en sådan fil är symboltabellen för kärnan, vilken under de flesta PC baserade Linux distributioner är sparade på samma ställe som den kompilerade kärnan. Under Android är det desto svårare då den som regel inte skickas med på enheten. Varför kan man bara spekulera om men troligen så ser de helt enkelt ingen anledning till varför den skulle behövas på en kommersiell produkt.

Det resonemanget gör det dock svårt för minnesanalysatorer såsom *Volatility* då de är beroende av denna symboltabell för att fungera. Alltså måste det användas någon metod för att få fram denna tabell. I listningen nedan diskuteras ett urval av några metoder;

Kompilera en ny kärna
: Ett säkert sätt att få åtkomst till symboltabellen är att kompilera en ny kärna. För att det ska vara av någon nytta för analys måste dock den nykompilerade kärnan flyttas över till enheten. Enheten måste sedan startas om så att den nya kärnan startas.

  Den här metoden är dock endast lämpad för forskning och testningssyfte, ur forensiskt syfte är detta inte ett alternativ.

Extrahera från enhet
: Som nämnts tidigare så skickas symboltabellen som regel inte med på Android enheter. Med det sagt så finns det antagligen någon enhet på marknaden som har den medskickad. Om den finns så finns den på enhetens "boot" partition och kan packas upp med lämpliga verktyg.

Ovanstående rader är till stor del spekulation då det under projektet inte har funnits tillgång till någon enhet med symboltabell på "boot" partitionen så det har inte varit möjligt att testa detta.

Utnyttja export av symboler
: Som standard (om inte bortkonfigurerat vid kompilering) så exporterar kärnan de publika symbolerna till en virtuell fil under "/proc/kallsyms". Att använda den här filen kan fungera, det beror på vilka symboler programmet behöver för att slutföra den specifika uppgiften och om dessa har exporterats eller inte. Det är dock det lättare av de olika sätten då det är enkelt att kopiera ut den via *adb*, det är dock viktigt att döpa om filen "System.map" för att *Volatility* ska hitta den.

Från tillverkaren
: Det bästa vore om sådan information kunde fås direkt ifrån tillverkaren av enheten, de var trots allt de som kompilerade systemet innan enheten skickades till försäljning.

  Även om vissa tillverkare erbjuder utvecklarplattformar där utvecklare kan komma åt mera avancerad information om de olika enheterna som tillverkaren har i sitt sortiment, är det troligt att den här typen av lågnivå information inte är tillgänglig.

6.2 Andra metoder

I vissa fall kanske det inte är nödvändigt att sätta upp och använda en komplex mjukvara för att analysera en minnesdump då nödvändiga resultat kan nås genom att använda andra metoder. I den här sektionen så kommer ett urval av dessa metoder och verktyg att analyseras.

6.2.1 Söka efter strängar

En av de mest elementera typerna av analys man kan göra är att söka igenom dumpen efter textsträngar, vilket är något som kan resultera i intressant data om man har tur. Vad exakt som utgör en sträng kan vara något som är bra att ha i åtanke. Historiskt sett så har en sträng alltid varit en samling en byte heltal (1-255) följt av en nolla. Dessa heltal översätts till tecken (bokstäver, siffror, specialtecken etc) via en uppslagstabell kallad "ASCII-tabellen". Problemet med den representationen är att antalet möjliga tecken är för lågt för att vara användbart på internationell skala. För att råda bot på det använder moderna system något som kallas för "unicode". I "unicode" kan varje tecken representeras av flera bytes och det finns därmed plats för alla de tecken som behövs internationellt i samma system.

Ett program som stödjer både klassiska ASCII-strängar och (beroende på version) Unicode är det klassiska programmet kort och gott kallat *strings*. *Strings* är en del av GNU programsviten och finns även portat till andra operativsystem såsom Windows. [12]

Att köra *strings* på en minnesdump är inte idealt då mängden data som programmet genererar kommer att vara väldigt stor. Tyvärr så finns det egentligen inget hundraprocentigt sätt att filtrera bort skräpet ur utdatan som *strings* genererar utan det bästa är att gå igenom allting manuellt.

6.2.2 Filkarvning

När ett program som körs i systemet behöver komma åt en fil så läser det ofta in den filen i minnet. Detta då det påskyndar hanteringen i programmet i kontrast mot att läsa ifrån disk hela tiden. Effekten av detta är att en stor mängd filer med största sannolikhet finns i interminnet och till skillnad från lagring på disk så är filerna inte fragmenterade. Detta innebär att det är en prima kandidat för filkarvning!

Filkarvning är en process där ett speciellt program går igenom en mängd data och letar efter kända start och slut för olika typer av filer. Därefter kopierar programmet datan, emellan start och slut, till en nya fil. Resultatet är en samling "lösa" filer som har plockats ut ur minnet.

Ett exempel på ett program med öppen källkod som klarar detta är *foremost*, programmet skrevs från början för "Air Force Office of Special Investigations" och "The Center for Information Systems Security Studies and Research" men har sedan släppts fritt [5]. *Foremost* har stöd en mängd filtyper bland annat; *jpg*, *png*, *gif*, *mpg* och *ole*. Stöd finns även för exekverbara filer och *zip*-arkiv.

För att använda *foremost* så är grundkommandot;

```
# foremost -o output_dir/ -i memdump.padded
```

I ovanstående exempel så specificeras det inte vilka filformat som programmet ska söka efter, varför *foremost* kommer att söka efter alla den har stöd för. Argumentet "-o" anger en katalog där resultatet av karvningen ska sparas och "-i" är vilken fil den ska karva i.

En annan mjukvara som kan användas för karvning är *PhotoRec* vilken även den är baserad på öppen källkod (GPLv2).[3] Till skillnad från *foremost* så har *PhotoRec* ett dialog baserat gränssnitt vilket innebär att användaren konfigurerar de olika alternativen via ett interaktivt manér. Detta gör att *PhotoRec* kan vara enklare att använda för en ovan användare.

Om filkarvning ska utföras på en minnesdump kan formatet på dumpen påverka resultatet, som nämnt tidigare ("Om LiME" Sektion 5) så finns det tre att välja på. Av dessa är "LiME" formatet det minst lämpade, detta då dess "sektions headers" tolkas som relevant data och kan korruptera karvade filer. De återstående två formaten är snarlika och fungerar antagligen lika bra, men om man ska välja så är det "paddade" formatet det bästa då det bevarar den interna strukturen i dumpen. Det kommer även innebära att de minnes-"offset" som rapporteras av *foremost* kommer att stämma med de offset som gällde på den riktiga datorn.

6.2.3 Virussökning

Detta är inte någon riktig analysmetod i den meningen att den inte hjälper till att plocka ut data eller tolka komplexa strukturer. Däremot kan det användas för att snabbt få svar på frågan om den undersökta datorn eller Android-enheten är smittad av malware.

En lämplig antivirusprogramvara som passar in här är "Clam Antivirus" vilken är en gratis programvara med öppen källkod och som licenseras under GPL och har löpande uppgraderingar [4]. "Clam Antivirus" har en bakgrundstjänst (*freshclam*) som hanterar själva uppdateringen av antivirusdefinitionerna medans skanningar initieras av ett konsolbaserat klientprogram (*clamscan*).

Att starta en virusskanning av en minnesdump är synnerligen okomplicerat, följande kommando räcker i 99 fall av 100:

```
# clamscan "memdump.padded"
```

Där "memdump.padded" är filen från minnesdumpningen. Det format på minnesdumpen som är bäst för detta är även i det här fallet "padded" och anledningen är precis samma som i sektion 6.2.2.

6.3 Analys med Volatility

I enhet med den etablerade metoden så började testen med PC sidan. Minnesdumpen som användes under testningen togs från en PC med kärn-version *Linux 2.6.38* (vilken används i *Backtrack 5*), då inte *Volatility* har en profil från denna kärna från början så är första steget att skapa en profil.

För att skapa en profil så användes först en med *Volatility* medskickad *Makefile* för att skapa filen "module.dwarf", detta är *dwarfdumpen* som nämndes tidigare.

```
# cd cd volatility/tools/linux
# make
# ls -l module.dwarf
-rw-r--r-- 1 root root 1399694 2013-03-18 12:22 module.dwarf
```

Den andra filen som behövdes är symboltabellen för kärnan vilken under *Backtrack* fanns sparade under */boot* och hette på testmaskinen *System.map-2.6.38*. Filen döptes om till *System.map* och packades ihop tillsammans med *module.dwarf* till ett *ZIP*-arkiv som i sin tur döptes till "2.6.38-Backtrack5.zip".

För att *Volatility* ska hitta den nya profilen måste denna sparas i *Volatility*'s profilkatalog (standard *volatility/plugins/overlays/linux/*).

Som test av profilen så användes modulen "linux_pslist" vilken används för att lista aktiva processer.

```
# vol.py --profile Linux2_6_38-Backtrack5x64 -f memdump.lime linux_pslist
Volatile Systems Volatility Framework 2.3_beta
Offset             Name                 Pid             Uid             Gid
------------------ -------------------- --------------- --------------- ------
0xffff880016da0000 init                 1               0               0
0xffff880016da16c0 kthreadd             2               0               0
0xffff880016da2d80 ksoftirqd/0          3               0               0
0xffff880016da4440 kworker/0:0          4               0               0
0xffff880016da5b00 kworker/u:0          5               0               0
0xffff880016dc8000 migration/0          6               0               0
0xffff880016dc96c0 cpuset               7               0               0
0xffff880016dcad80 khelper              8               0               0
<<< SNIP >>>
0xffff880015adc440 sshd                 831             0               0
0xffff88001411db00 sshd                 27318           0               0
```

I utdatan från kommandot så finns det två jobb som bägge heter 'sshd'. Hypotesen här är att den första med *PID* 831 är den ursprungliga servicen och den andra är en ny som skapades för att hantera en ansluten användare. För att verifiera hypotesen så används modulen "linux_pstree" för att lista ett processträd. linux_pstree command.

```
# vol.py --profile Linux2_6_38-Backtrack5x64 -f memdump.lime linux_pstree -p 831
Volatile Systems Volatility Framework 2.3_beta
Name                 Pid             Uid
sshd                 831             0
.sshd                27318           0
..bash               27335           0
```

Detta bevisar att hypotesen antagligen stämmer då ovanstående processträd även visar att det andra 'sshd' jobbet har startar en instans av skalet *Bash*. Då *SSH* är en nätverksteknik så borde det även gå att få fram nätverksinformation. Vilket i *Volatility* enklast görs med modulen 'linux_netstat';

```
# vol.py --profile Linux2_6_38-Backtrack5x64 -f memdump.lime linux_netstat -p 831,27318
Volatile Systems Volatility Framework 2.3_beta
TCP      0.0.0.0:22            0.0.0.0:0             LISTEN           sshd/831
TCP      :::22                 :::0                  LISTEN           sshd/831
TCP      192.168.232.129:22    192.168.232.1:59515   ESTABLISHED      sshd/27318
```

Den sista raden visar tydligt att det finns en etablerad anslutning mellan 192.168.232.129 och 192.168.232.1 på port 22, vilket är porten som används för *SSH*.

Då *Volatility* fungerade tillfredsställande för PC plattformen så skiftades därmed fokus över till Android. En profil skapades för den aktuella kärnan på Android-enheten. Detta var inget som ställde till några problem då enheten i fråga använde sig av en egen kompilerad version av Android

(Cyanogenmod 10.1) och det därmed fanns fri tillgång till symboltabeller och annan information.

Ett försök gjordes sedan att återupprepa testfallet för PC med användandet av 'linux_pslist' modulen, detta resulterade dock i utdatan nedan.

```
#vol.py --profile LinuxAndroid-Blade-CM10_1-2_6_35ARM -f cm10.1-blade.lime linux_pslist
Volatile Systems Volatility Framework 2.3_beta
Offset     Name                 Pid             Uid              Gid    DTB
---------- -------------------- --------------- ---------------- ------ ----------
WARNING : volatility.plugins.addrspaces.arm: get_pte: invalid fine pde2_value 40c3f7a0
WARNING : volatility.plugins.addrspaces.arm: get_pte: invalid fine pde2_value 40c3f7a0
WARNING : volatility.plugins.addrspaces.arm: get_pte: invalid fine pde2_value 40c3f7a0
WARNING : volatility.plugins.addrspaces.arm: get_pte: invalid pde_value 28
No suitable address space mapping found
```

Vad exakt ovanstående fel berodde på är svårt att säga. En buggrapport öppnades emot *Volatility*'s utvecklingsteam och mjukvaran har gått igenom en mängd omskrivningar sedan dess, men problemet kvarstår. Med det sagt, *Volatility* 2.3 befinner sig fortfarande i betastadium och det är troligt att det löser sig när den skarpa versionen släpps.

7 Logisk dump av processinfo

För att få en djupare förståelse om hur Android hanterar processer på kärnans nivå så uppstod ett behov att extrahera information från systemet.

Den information som framför allt är av intresse för att utläsa processtrukturen är information relevant till processer, dels grundläggande information såsom; *process-id* (PID), *förälders process-id* (PPID), *användar-id* (UID), *grupp-id* (GID) och *status* (STATE) för processen. Samt minnesområden som denna process har mappat in till sitt adressområde.

Alla information nämnd ovan är tillgänglig i de datastrukturer beskrivna i bilaga A. Då dessa datastrukturer är endast tillgängliga inifrån kärnan måste även här utdumpningen av denna information ske inifrån kärnan. Lösningen på detta var att implementera en speciell kärn-modul[18] som kan extrahera och spara undan ovan nämnda information. Detta görs genom att kärnans datastrukturer traverseras varav informationen som finns sparad i dessa sparas ner.

Traverseringen i sig går till på så sätt att den börjar med den struktur som lagrar processer i systemet (*task_struct*), denna innehåller all information om olika identifierare och status. Strukturen innehållller även en hänvisning till en annan struktur som representerar processens minnes allokering. Vad modulen sen gör är att iterera över alla processens minnesmappningar och dumpar innehållet i dessa områden till en separata filer. Den sparar även ner metainformation, såsom minnesadress och behörigheter, till samma fil som tidigare processinfo skrevs till.

Slutproduketen är en binärfil med all processinformation lagrad i ett speciellt format (beskrivet i tabell 2). Samt en serie filer vars innehåll är rena dumpar av det minne som var inmappat till de olika processerna, namnet på dessa filer är addressen till vart i det fysiska minnet datat ursprungligen började på.

Beskrivning	**Längd (bytes)**
Processinformation (separerad med " "): PID Command UID PPID STATE	Dynamiskt
Separator (0xFF)	1
Enskild minnesmappning börjar	
Längd	8
Separator (0xFF)	1
Address	8
Separator (0xFF)	1
Flaggor	8
Separator (0xFF)	1
Namn	50
Separator (0xFF)	1
Om det finns flera minnesmappningar så börjar ovanstående om igen	
Footer	5
Nästa process börjar	

Tabell 2: Formatet på informationsfilen

För att avkoda informationsfilen som skapades av modulen används ett speciellt verktyg skrivet i Python vilket går igenom all data i filen och generar en rapport i *html* format, vilket innebär att den kan visas i en vanligt webbläsare.

Det kan anses vara onöndigt krångligt att dela upp det hela så att två olika verktyg behövs för att komma fram till ett användbart resultat. Det ligger helt enkelt något i det och en lösning där modulen själv skapade den slutgiltiga rapporten skulle vara enklare att använda. Anledningen till varför det inte är så är att möjligheterna till direkt filåtkomst inifrån kärnan är väldigt begränsade och det är även något som ska undvikas om möjligt. Med det i åtanke så skapades modulen med målet att göra så få diskskrivningar som möjligt.

[18] För källkod till denna se *http://github.com/nojan1/pinfodump*

7.1 Användande

Efter att modulen har kompilerats[19] för den aktuella kärnan, så behöver denna laddas in i kärnan. Även här så är *insmod* verktyget för jobbet, de argument som modulen lyssnar till är;

path
: Sökväg till en katalog där modulen ska skriva sina dumpfiler. Katalogen måste existera i förväg.

dumpmem
: Anger om modulen ska exportera mappat minne eller inte. Anges som 1 (exportera) eller 0 (hoppa över). Standard om det inte skickas med är 1.

Ett exempel på inläsning av modulen kan se ut så här;

```
# insmod pinfodump.ko path=/run/media/portable/pinfo dumpmem=0
```

Efter att modulen har körts färdigt är nästa steg att använda det i Python skrivna verktyget "dumpextract.py" för att generera en rapport. Scriptet accepterar följande parametrar;

```
dumpextract.py <DUMP PATH> <OUTPUT FILE>
```

Ett exempel med samma sökväg som innan blir då;

```
# python dumpextract.py /run/media/portable/pinfo pinfodump.html
```

Resultatet av ovanstående kommando är filen "pinfodump.html" vilket är själva rapporten och kan öppnas i valfri webbläsare.

[19] Kompileringsförfarandet är väldigt lika som för *LiME* och instruktionerna i bilaga D kan därför användas som riktlinjer även för denna modul

8 Diskussion

Minnesanalys är utan tvekan ett komplext område, dels när det gäller PC baserade system men ännu desto mer när man närmar sig inbyggda system. Det man kan säga på rak arm om de nuvarande teknikerna som finns för minnesanalys på Linux-baserade system, är att vi inte har nått "one click tool" stadiumet än. Med det menar jag att det fortfarande kräver så pass mycket handpåläggning från fall till fall så att ingen automatisk algoritm för problemet skapas.

Det bästa sättet att värdera de metoder som har diskuteras i den här rapporten är att tänka sig ett hypotetiskt scenario där en professionell IT-forensiker får uppgiften på sitt bord och måste producera resultat.

Scenario ett är att utredaren får in en PC-laptop med Linux som operativsystem. Datorn är påslagen och drivs av batteri. För att kunna dumpa minne (för att sedan analysera det) så skulle utredaren börja med att identifiera vilken version av kärnan som körs. Utredaren behöver sedan förbereda källkodsträdet för den aktuella kärnan och kompilera dumpningsmodulen mot rätt kärna. Modulen skulle sedan kopieras till ett USB-minne, varefter minnet ansluts till den beslagna datorn. Därefter skulle utredaren initiera dumpningen och spara ner den resulterade dumpfilen till USB-minnet varav dumpningsprocessen är slutförd.

Den första problempunkten ovan är antagligen att kompilera dumpningsmodulen för den kärna som finns på den beslagtagna datorn. Svårigheten ligger dels i att sätta upp källkodsträdet för kärnan men även själva kompileringen vilken kan var knivig om man aldrig har gjort det förut. Det är dock troligt att det endast kommer vara ett problem första gången personen gör det för att sedan kunna göras utan problem vid efterföljande tillfällen. Nästa stora problem är hur modulen läses in i kärnan på den beslagna datorn då det (som nämnts tidigare) är något som kräver root-åtkomst och indirekt ett lösenord. För att utredaren ska kunna få tag i det nödvändiga lösenordet så måste i princip lösenordshasharna exporteras ifrån datorn och knäckas i en annan dator vilket är en process som kan ta väldigt lång tid. Med en stor mängd tur så kan det gå att uppnå root-behörighet utan att ha root-användarens lösenord, detta genom kommandot "sudo". Det är dock nånting som beror på konfigurationen av den aktuella datorn samt när "sudo" användes senaste.

Nästa steg i utredningen kommer bli att analysera minnesdumpen vilket i sin tur kräver att en profil skapas. Detta är dock inget som borde skapa några större problem då allt som behövs kan extraheras ifrån den beslagna datorn.

Om man istället tänker sig ett scenario där det är en Android enhet som ska undersökas så blir dumpningsprocessen annorlunda. Först och främst rör det sig troligtvis om en *ARM* baserad *CPU* vilket kräver korskompilering av modulen. En andra skillnad är att enhetens minneskort troligtvis kommer användas istället för det USB-minne som kan användas på en PC.

Problempunkter under dumpningsstadiet liknar de för PC-sidan i den mån att kompileringsdelen kommer vålla störst problem första gången. Efter det tenderar det att bli tämligen repetitivt och någonting man snabbt lär sig. Att uppnå nödvändig root-behörighet är däremot nånting som skiljer sig en hel del från PC i det att ingrep troligen måste göras på enheten (rootning). Att göra ingrep på beslagtagen hårdvara är något som enligt forensisk sed ska undvikas men om metoden med kärn-modul ska användas så finns det inte mycket till alternativ.

Att analysera en minnesdump ifrån Android kräver precis som för PC att en profil skapas, till skillnad ifrån PC är det dock inte lika lätt att komma åt nödvändig information. Fram för allt är det symboltabellen som ställer till problem, det är till och med så att det för tillfället inte finns någon heltäckande alltid fungerande metod att komma åt denna tabell ifrån en enhet. På grund av detta så är analysen den helt klart svåraste punkten när det det kommer till Android och det är inte alls osannolikt att utredaren får lov att falla tillbaka på andra mera grundläggande metoder, såsom; söka efter strängar, karvning etc.

Jag nämnde inledningsvis att det inte finns något helautomatiskt verktyg som snabbt och enkelt kan analysera minnet hos en enhet. En frågeställning man kan tänka sig då är; Hur långt fram i tiden ligger detta och är det även möjligt? Min åsikt är att det skulle kräva att enheter fick ett generellt sätt att dumpa ut minne via ett hårdvarugränssnitt, detta skulle innebära att ingen åverkan behöver

göras i mjukvaran. Istället skulle utdumpningsprocessen bli beroende av att utredarna har tillgång till korrekt hårdvara för att kommunicera med enheten, detta har fördelen att det blir oberoende av tillverkare och modell med kravet förstås att den är utrustade med det speciella gränssnittet.

För en generell analysmetod så ser jag två alternativ. Det ena är den symbolbaserade metoden men med förändringen att utredare har ett löpande samband med tillverkare och får tillgång till symboltabeller så fort en ny modell släpps. Det som troligen kan vara ett problem med detta är att det kräver att tillverkarna ska lägga tid, pengar och resurser på något som de inte har någon direkt vinning av. I ett företag med målet att öka sina inkomster blir det därför kontraproduktivt.

En annan typ av analysmetod är skulle vara att utveckla en signaturbaserad programvara som kan försöka lära sig vart de olika datastrukturerna börjar och på det sättet försöka lista ut vilken information som finns där. Nackdelen med en sådan lösning är att den kan komma att missa viktig information av den enkla anledningen att den inte förstår vad det är den tittar på. Denna typ av programvara med den noggrannhet som krävs i en forensisk undersökning finns vad jag vet inte på marknaden än och jag har inte heller sett några konkreta bevis på att en sådan heller håller på att utvecklas.

9 Slutsatster

Två av de svårare målen som sattes inför examensarbetet var "Anpassa metoder till ett forensisk vedertaget tillvägagångssätt" och "Utveckla metoder för att extrahera alla tillgängliga forensiska artefakter". Jag skulle säga att dessa två mål ej har uppnåtts till fullo. Med den första av de två punkterna så faller det till stor del på de stora ingrepp som måste göras för att få ett resultat, på PC sidan är det inte så illa men för Android är det värre. Till exempel så uppstår det genast problem om en enhet som behöver undersökas inte är rootad. Detta då de metoder som finns för att roota en Android-enhet inte är skapad med forensiskt användande i åtanke och det inte går att garantera hundraprocentig integritet av det fysiska minnet efter proceduren.

Det andra av de två icke uppfyllda målen kan inte anses uppfyllt då arbetet har fokuserat på processer och processminne. För att det målet ska anses uppfyllt så skulle mera metoder behöva tas fram för att specifikt extrahera sådana artefakter som lösenord, krypteringsnycklar, öppna filer och annat som kan vara av intresse i en forensisk utredning.

En del av examensarbetet var även att utvärdera om dessa metoder kan fungera att användas på daglig basis av en professionell IT-forensiker, svaret på den frågan är inte helt enkelt. Oavsett plattform så tar dessa metoder väldigt mycket tid i anspråk vilket inte är helt önskvärt om man har en hög av fall att utreda. Som tidigare nämnts är det dessutom långt ifrån en lösning där man kan klicka på en knapp och få allt serverat utan istället då kräver det mera skicklighet ifrån utredarens sida. Det är troligt att ytterligare utbildning kan krävas om inte nödvändig kompetens redan finns.

Sett endast till resultat så är metoderna beskrivna i denna rapport fullt dugliga på PC sidan. Med dugliga menas här att man kan få ut ett resultat som kan ligga till grund för en riktig utredning. Om man istället ser till Android så fungerar metoderna inte lika bra och det är då framför allt på analyssidan. Här finns det alltså utrymme för ytterligare förbättringar i att ta fram alternativa analysmetoder och då med fördel en som inte är symbolbaserad.

Referenser

[1] Android Kernel Features. Description of the unique kernel feature in the android kernel, Maj 2013. URL `http://elinux.org/Android_Kernel_Features`.

[2] Dan Bornstein. Dalvik VM internals, 2008. URL `https://sites.google.com/site/io/dalvik-vm-internals`.

[3] CGSecurity. Photorec official webpage, September 2013. URL `http://www.cgsecurity.org/wiki/PhotoRec`.

[4] ClamAv. Clam antivirus offical webpage, April 2013. URL `http://www.clamav.net/lang/en/`.

[5] Foremost. Foremost offical webpage, April 2013. URL `http://foremost.sourceforge.net/`.

[6] LiME. LiME official webpage, April 2013. URL `http://code.google.com/p/lime-forensics/`.

[7] Robert Love. *Linux Kernel Development Third Edition.* Addison Wesley, 2010. ISBN 0-672-32946-8.

[8] OneClickRoot. OneClickRoot official webpage, April 2013. URL `http://www.oneclickroot.com/`.

[9] Palm Source Inc. Open binder documentation, April 2005. URL `http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html`.

[10] Sans Institute. Techniques and tools for recovering and analyzing data from volatile memory, 2009. URL `http://www.sans.org/reading_room/whitepapers/forensics/techniques-tools-recovering-analyzing-data-volatile-memory_33049`.

[11] Bradley Schatz. Toward reliable volatile memory acquisition by software. 2007. doi: 10.1016/j.diin.2007.06.009. URL `https://www.dfrws.org/2007/proceedings/p126-schatz.pdf`.

[12] Strings. GNU strings man page, April 2013. URL `http://linux.about.com/library/cmd/blcmdl1_strings.htm`.

[13] John Stultz. Lwn, [RFC][PATCH] anonymous shared memory (ashmem) subsystem, Juli 2011. URL `http://lwn.net/Articles/452035/`.

[14] Joe Sylve, Andrew Case, Lodovico Marziale, and Golden G. Richard. Acquisition and analysis of volatile memory from android devices. *Digital Investigation*, 8(3–4):175 – 184, 2012. ISSN 1742-2876. doi: 10.1016/j.diin.2011.10.003. URL `http://www.sciencedirect.com/science/article/pii/S1742287611000879`.

[15] Vrizlynn L.L. Thing, Kian-Yong Ng, and Ee-Chien Chang. Live memory forensics of mobile phones. *Digital Investigation*, 7, Supplement(0):S74 – S82, 2010. ISSN 1742-2876. doi: 10.1016/j.diin.2010.05.010. URL `http://www.sciencedirect.com/science/article/pii/S174228761000037X`. The Proceedings of the Tenth Annual DFRWS Conference.

[16] Linus Torvalds. What would you like to see the most in minix?, Augusti 1991. URL `http://groups.google.com/group/comp.os.minix/msg/b813d52cbc5a044b?dmode=source&pli=1`.

[17] Arjan van de Ven. Introduce /dev/mem/ restrictions with a config option. Kernel mailing list, Januari 2008. URL `http://lwn.net/Articles/267427/`.

[18] Timothy Vidas, Chengye Zhang, and Nicolas Christin. Toward a general collection methodology for android devices. *Digital Investigation*, 8, Supplement(0):S14 – S24, 2011. ISSN 1742-2876. doi: 10.1016/j.diin.2011.05.003. URL `http://www.sciencedirect.com/science/article/pii/S1742287611000272`. The Proceedings of the Eleventh Annual DFRWS Conference.

[19] Volatility. Volatility official webpage, April 2013. URL `http://code.google.com/p/volatility/`.

[20] Volatility Wiki. Volatility wiki page for android forensics, April 2013. URL `http://code.google.com/p/volatility/wiki/AndroidMemoryForensics`.

[21] Volatility Wiki. Volatility wiki page for linux forensics, April 2013. URL `http://code.google.com/p/volatility/wiki/LinuxMemoryForensics`.

[22] W3C. Os platform statistics, April 2013. URL `http://www.w3schools.com/browsers/browsers_os.asp`.

Bilagor

A Relevanta datastrukturer för processminneshantering

Nedan följer en uppräkning av de viktigaste datastrukturer som används av Linux-kärnan för att hålla ordning på processer och dess minnesmappningar.

A.1 task_struct

Nyckel strukturen när det kommer till jobb ("tasks") är *task_struct* vilken håller information så som; process-id, förälder, aktuell status etc. Den fulla strukturen är för stor (1,7 KB för en 32 bitars maskin) för att listas här, den är dock definierad i *<linux/sched.h>* i kärnans källkodsträd.

Strukturen innehåller förutom en pekare till sin förälder även pekare till nästa och föregående element, det rör sig nämligen om en cirkulärt länkade lista. Detta ger möjligheten att på ett enkelt sätt iterera över alla jobb som finns i systemet. Riktlinjerna för utvecklare uppmanar dock till en sparsam användning av detta då det kan bli tidsödande om listan blir lång.

A.2 mm_struct

mm_struct strukturen agerar som en sorts mellanhand mellan ett jobb och dess minne.

Listning 4 är ett utdrag av definitionen för strukturen, den fulla är definierad i *<linux/mm_types.h>* och är även bifogad i bilaga B.

```
struct mm_struct {
        struct vm_area_struct * mmap;           /* list of VMAs */
        struct rb_root mm_rb;
        struct vm_area_struct * mmap_cache;     /* last find_vma result */
        <<< SNIP >>>
        unsigned long mmap_base;                /* base of mmap area */
        unsigned long task_size;                /* size of task vm space */
         /* if non-zero, the largest hole below free_area_cache */
        unsigned long cached_hole_size;
        /* first hole of size cached_hole_size or larger */
        unsigned long free_area_cache;
        pgd_t * pgd;
        /* How many users with user space? */
        atomic_t mm_users;
        /* How many references to "struct mm_struct" (users count as 1) */
        atomic_t mm_count;
        int map_count;                          /* number of VMAs */
        struct rw_semaphore mmap_sem;
        spinlock_t page_table_lock;             /* Protects page tables and some counters */
        <<< SNIP >>>
        //The memory areas for code, heap, stack and arguments
        unsigned long total_vm, locked_vm, shared_vm, exec_vm;
        unsigned long stack_vm, reserved_vm, def_flags, nr_ptes;
        unsigned long start_code, end_code, start_data, end_data;
        unsigned long start_brk, brk, start_stack;
        unsigned long arg_start, arg_end, env_start, env_end;
};
```

Listning 4: Utdrag ur *mm_struct*

Strukturen har två olika typer av länkade strukturer som lagrar de olika virtuella minnes områdena (*vm_area_struct*) och dessa är *mm_rb* och *mmap*. Skillnaden mellan dessa är att den första är ett röd-svart sökträd och den andra en vanlig länkad lista. Anledningen till att det är två är att de är bra på olika saker, till exempel så är sökträdet väldigt snabbt när det kommer till att söka medans listan är oslagbart smidig när det kommer till att iterera över alla element. Notera dock att den dubbleringen av data som detta orsakar endast rör sig om interna minnespekare, det sker alltså ingen onödig dubbellagring av det data pekarna pekar på.

Det finns även variabler som beskriver vart de olika processegmenten (*text*, *data*, *heap* och *stack*) börjar respektive slutar.

A.3 vm_area_struct

Strukturen *vm_area_struct* representerar individuella områden av minne, vart området börjar samt diverse flaggor. Dessa flaggor specificerar sådant som åtkomsträttigheter (läsa, skriva, exekvera) men även andra mera specifika saker som; vilket håll minnet växer. Den sista kan verka konstig men kom ihåg att portabiliteten är viktig. [7, Kapitel 14]

Som innan så är ett utdrag av definitionen listad i listning 5 och den fulla är medskickad som bilaga C.

```
struct vm_area_struct {
        struct mm_struct * vm_mm;       /* The address space we belong to. */
        unsigned long vm_start;         /* Our start address within vm_mm. */
        unsigned long vm_end;           /* The first byte after our end address
                                           within vm_mm. */

        /* linked list of VM areas per task, sorted by address */
        struct vm_area_struct *vm_next, *vm_prev;

        pgprot_t vm_page_prot;          /* Access permissions of this VMA. */
        unsigned long vm_flags;         /* Flags, see mm.h. */

        struct rb_node vm_rb;

        <<< SNIP >>>
};
```

Listning 5: Utdrag ur *vm_area_struct*

B The full mm_struct

```
struct mm_struct {
        struct vm_area_struct * mmap;           /* list of VMAs */
        struct rb_root mm_rb;
        struct vm_area_struct * mmap_cache;     /* last find_vma result */
#ifdef CONFIG_MMU
        unsigned long (*get_unmapped_area) (struct file *filp,
                                unsigned long addr, unsigned long len,
                                unsigned long pgoff, unsigned long flags);
        void (*unmap_area) (struct mm_struct *mm, unsigned long addr);
#endif
        unsigned long mmap_base;                /* base of mmap area */
        unsigned long task_size;                /* size of task vm space */
         /* if non-zero, the largest hole below free_area_cache */
        unsigned long cached_hole_size;
        /* first hole of size cached_hole_size or larger */
        unsigned long free_area_cache;
        pgd_t * pgd;
        /* How many users with user space? */
        atomic_t mm_users;
        /* How many references to "struct mm_struct" (users count as 1) */
        atomic_t mm_count;
        int map_count;                          /* number of VMAs */
        struct rw_semaphore mmap_sem;
        spinlock_t page_table_lock;             /* Protects page tables and some counters */

        struct list_head mmlist;                /* List of maybe swapped mm's.
                                                 * These are globally strung
                                                 * together off init_mm.mmlist, and are protected
                                                 * by mmlist_lock
                                                 */


        unsigned long hiwater_rss;      /* High-watermark of RSS usage */
        unsigned long hiwater_vm;       /* High-water virtual memory usage */

        unsigned long total_vm, locked_vm, shared_vm, exec_vm;
        unsigned long stack_vm, reserved_vm, def_flags, nr_ptes;
        unsigned long start_code, end_code, start_data, end_data;
        unsigned long start_brk, brk, start_stack;
        unsigned long arg_start, arg_end, env_start, env_end;

        unsigned long saved_auxv[AT_VECTOR_SIZE]; /* for /proc/PID/auxv */

        /*
         * Special counters, in some configurations protected by the
         * page_table_lock, in other configurations by being atomic.
         */
        struct mm_rss_stat rss_stat;

        struct linux_binfmt *binfmt;

        cpumask_t cpu_vm_mask;

        /* Architecture-specific MM context */
```

```
        mm_context_t context;
        /* Swap token stuff */
        /*
         * Last value of global fault stamp as seen by this process.
         * In other words, this value gives an indication of how long
         * it has been since this task got the token.
         * Look at mm/thrash.c
         */
        unsigned int faultstamp;
        unsigned int token_priority;
        unsigned int last_interval;

        unsigned long flags; /* Must use atomic bitops to access the bits */

        struct core_state *core_state; /* coredumping support */
#ifdef CONFIG_AIO
        spinlock_t              ioctx_lock;
        struct hlist_head       ioctx_list;
#endif
#ifdef CONFIG_MM_OWNER
        /*
         * "owner" points to a task that is regarded as the canonical
         * user/owner of this mm. All of the following must be true in
         * order for it to be changed:
         *
         * current == mm->owner
         * current->mm != mm
         * new_owner->mm == mm
         * new_owner->alloc_lock is held
         */
        struct task_struct *owner;
#endif

#ifdef CONFIG_PROC_FS
        /* store ref to file /proc/<pid>/exe symlink points to */
        struct file *exe_file;
        unsigned long num_exe_file_vmas;
#endif
#ifdef CONFIG_MMU_NOTIFIER
        struct mmu_notifier_mm *mmu_notifier_mm;
#endif
};
```

C The full vm_area_struct

```
struct vm_area_struct {
        struct mm_struct * vm_mm;       /* The address space we belong to. */
        unsigned long vm_start;         /* Our start address within vm_mm. */
        unsigned long vm_end;           /* The first byte after our end address
                                           within vm_mm. */

        /* linked list of VM areas per task, sorted by address */
        struct vm_area_struct *vm_next, *vm_prev;

        pgprot_t vm_page_prot;          /* Access permissions of this VMA. */
        unsigned long vm_flags;         /* Flags, see mm.h. */

        struct rb_node vm_rb;

        /*
         * For areas with an address space and backing store,
         * linkage into the address_space->i_mmap prio tree, or
         * linkage to the list of like vmas hanging off its node, or
         * linkage of vma in the address_space->i_mmap_nonlinear list.
         */
        union {
                struct {
                        struct list_head list;
                        void *parent;   /* aligns with prio_tree_node parent */
                        struct vm_area_struct *head;
                } vm_set;

                struct raw_prio_tree_node prio_tree_node;
        } shared;

        /*
         * A file's MAP_PRIVATE vma can be in both i_mmap tree and anon_vma
         * list, after a COW of one of the file pages.  A MAP_SHARED vma
         * can only be in the i_mmap tree.  An anonymous MAP_PRIVATE, stack
         * or brk vma (with NULL file) can only be in an anon_vma list.
         */
        struct list_head anon_vma_chain; /* Serialized by mmap_sem &
                                          * page_table_lock */
        struct anon_vma *anon_vma;      /* Serialized by page_table_lock */

        /* Function pointers to deal with this struct. */
        const struct vm_operations_struct *vm_ops;

        /* Information about our backing store: */
        unsigned long vm_pgoff;         /* Offset (within vm_file) in PAGE_SIZE
                                           units, *not* PAGE_CACHE_SIZE */
        struct file * vm_file;          /* File we map to (can be NULL). */
        void * vm_private_data;         /* was vm_pte (shared mem) */
        unsigned long vm_truncate_count;/* truncate_count or restart_addr */

#ifndef CONFIG_MMU
        struct vm_region *vm_region;    /* NOMMU mapping region */
#endif
#ifdef CONFIG_NUMA
```

```
        struct mempolicy *vm_policy;    /* NUMA policy for the VMA */
#endif
};
```

D Kompilering av LiME

LiME är en klassisk kärnmodul i den mån att den kompileras på samma sätt som andra moduler. Detta innebär att den måste byggas emot ett källkodsträd av den aktuella kärnan. Anledningen är att detta träd innehåller nödvändiga filer ur kärnans speciella byggsystem samt nödvändiga "headers" och andra filer. Ett exempel är filen "Module.symvers" vilken innehåller verisionsinformation angående moduler.

LiME projektet finns under googlecode och kan enkelt laddas ner därifrån, i skrivande stund är den rekommenderade versionen 1.1-r14. För att hämta källkoden och kompilera modulen görs enklast med följande steg;

```
# wget http://lime-forensics.googlecode.com/files/lime-forensics-1.1-r14.tar.gz
# tar xzvf lime-forensics-1.1-r14.tar.gz
# cd src/
# make
```

Den medskickade filen "Makefile" [20] bygger modulen emot ett källträd under */usr/lib/modules/[KVER]/build* där *KVER* är en variabel i Makefilen som anger vilken version av kärnan det gäller. Efter att kommandot "make" har slutförts kommer den kompilerade filen "lime-[KVER].ko" (exempel "lime-3.8.1.ko") att finnas i katalogen och detta är den färdiga modulen som kan laddas in i kärnan.

Värt att notera är att variabler i en Makefile kan överlagras när kommandot "make" körs, exempelvis så kommer;

```
make KVER=2.6.39
```

Att istället bygga emot version 2.6.39, vilket innebär att den kommer söka kodträdet under */usr/lib/modules/2.6.39/build* och den färdiga modulen kommer att heta "lime-2.6.39.ko". Detta kräver förstås att källkoden för den kärnan finns tillgänglig i systemet.

D.1 Speciell hänsyn för Android

Att kompilera för Android kräver ett speciellt tillvägagångssätt då de i de allra flesta fall använder sig av en processor ur ARM familjen. Detta skiljer sig ifrån en PC som oftast faller under x86 familjen. Det innebär att en modul som ska fungera på Android måste kompileras för en annan processorartitektur, vilket på fackspråk kallas för kors-kompilering.

Förkrav för kors-kompilering emot Android är;

- Android SDK (Standard mjukvaran för applikationsutveckling)
- Android NDK (Native Development Kit; Kompilator, länkare etc)[21]
- Källkod för kärnan i enheten
- Konfiguration för kärnan (filen *.config*)

Källkoden för kärnan måste matcha den kärna som körs på telefonen, detta innebär att den måste hämtas ifrån tillverkaren av telefonen. Rör det sig om rena "Google telefoner" så kan den rena kärnan användas (med rena menas här en kärna direkt ifrån Google som ingen annan tillverkare har petat runt i).

Om man bortser från källkoden så behövs även konfigurationen ifrån den aktuella kärnan på enheten, det enklaste sättet att få tag i denna är med hjälp av verktyget *adb*.

[20] Fil som innehåller instruktioner till make kommandot, kort förklarat en beskrivning över hur ett projekt ska kompileras

[21] Fortsättningsvis kommer sökvägen till denna benämnas "NDKPATH"

```
# adb pull /proc/config.gz
# gunzip config.gz
# mv config KERNEL-SOURCE/.config
```

Där "KERNEL-SOURCE" är sökvägen till kärnans källkod.

Det sista steget för att förbereda för kors-kompilering är genomförandet av en *modules_prepare* på källkodsträdet. Detta görs för att förbereda trädet för kompilering av moduler, och innefattar; kompilering av specifika bygg-skript, generera "headers" mm. För att initiera denna process används följande kommandon;

```
export CCPATH=[NDKPATH]/toolchains/arm-linux-androideabi-4.4.3/prebuilt/linux-x86/bin/
make ARCH=arm CROSS_COMPILE=$CCPATH/arm-linux-androideabi- modules_prepare
```

Nu när miljön är redo så behövs en speciell Makefile för att kunna kompilera *LiME*. Nedan finns en sådan listad;

```
obj-m := lime.o
lime-objs := tcp.o disk.o main.o

KDIR := /home/user/Android/kernel-source
PWD := $(shell pwd)
CCPATH := [NDKPATH]/toolchains/arm-linux-androideabi-4.4.3/prebuilt/linux-x86/bin/

default:
        $(MAKE) ARCH=arm CROSS_COMPILE=$(CCPATH)/arm-linux-androideabi- \
         EXTRA_CFLAGS=-fno-pic -C $(KDIR) M=$(PWD)

clean:
        rm -f *.ko
```

Variabeln *KDIR* pekar på sökvägen till kärnans källkod och *CCPATH* till sökvägen där de ARM specifika verktygen lagras, dessa är alltså en del av *Android NDK*. För att kompilera modulen så räcker det med att köra kommandot "make" och om kompileringen lyckas så kommer filen "lime.ko" att skapas.

För att verifiera att kompileringen har gått som tänkt kan verktyget "modinfo" användas eftersom detta verktyg läser ut information ur modulen och kan därmed användas för att verifiera resultatet.

```
# modinfo lime.ko
filename:       /home/user/Android/lime-src/lime.ko
license:        GPL
depends:
vermagic:       2.6.36.3 SMP preempt mod_unload ARMv7
parm:           path:charp
parm:           dio:bool
parm:           format:charp
```

Raden av intresse är "vermagic" vilket visar att den här modulen är tänkt för en kärna med version "2.6.36.3" vilken körs under arkitekturen "ARMv7".

E Ordlista

Flyktigt minne
: Ett flyktigt minne är per definition ett minne vars innehåll är flyktigt. Det innebär att minnets innehåll endast kan garanteras så länga minnet är spänningssatt. Exempel på typer av flyktigt minne är *SRAM* och *DRAM* vilka bägge bland annat används till datorers interminnnen.

Header-fil
: En header-fil är ett komplement till källkodsfiler i programmeringspråken *C* och *C++*. Poängen med header-filer är att definera variabler och funktioner på en central plats, detta gör att andra filer kan inkludera den filen och på det sättet använda sig av funktioner definerade där.

Symbol
: En symbol (i sammanhanget minnesanalys) kan beskrivas som en beskrivning av en specifik variabel eller funktion. Symbolen i sig består av en minnesaddress och namnet på entiteten.