Datavetenskap

Mikael Knutsson

Tomas Ljunggren

Skadlig kod med Java i mobiltelefoner

Examensarbete, C-nivå

2002:04

Skadlig kod med Java i mobiltelefoner

Mikael Knutsson

Tomas Ljunggren

Denna uppsats är skriven som en del av det arbete som krävs för att erhålla en kandidatexamen i datavetenskap. Allt material i denna uppsats, vilket inte är vårt eget, har blivit tydligt identifierat och inget material är inkluderat som tidigare använts för erhållande av annan examen.

Mikael Knutsson

Tomas Ljunggren

Godkänd, 2002-06-04

Handledare: Robin Staxhammar

Examinator: Stefan Alfredsson

Sammanfattning

Inom en snar framtid kommer antalet mobila terminaler med stöd för utbytbara applikationer att öka kraftigt. Detta innebär att mobila terminaler som till exempel våra mobiltelefoner kommer att kunna göra mycket mer än att upprätta samtal. Vårt mål med denna uppsats är att utreda möjligheten att skapa och sprida applikationer som verkar på ett skadligt sätt för användaren. Vi begränsar oss till applikationer för de mobila terminaler som stödjer javaplattformen. Då utveckling av applikationer för dessa enheter görs tillgänglig för tredjepartsutvecklare kommer säkerheten att sättas på prov. Det är viktigt att säkerheten mot skadliga applikationer är god eftersom mobila terminaler och särskilt mobiltelefoner är produkter som riktar sig mot en mycket stor grupp användare, en grupp för vilken tekniska lösningar och säkerhetsfrågor bör vara så transparenta som möjligt.

Nödvändig kunskap för att kunna genomföra uppsatsen har vi till stor del hämtat från Internet och specifikationer i form av digitala dokument eftersom ämnet är så nytt att det ännu inte finns böcker som behandlar det. Vi har studerat bakgrunden till javaplattformen i mobila terminaler, dess säkerhetsmodell samt områden där olika angrepp kan tänkas gå att genomföra. Därefter har vi genomfört en analys av möjligheterna att skapa och sprida skadlig kod inom dessa områden samt gjort ett antal tester.

Vi har kommit fram till att javaplattformen i mobila terminaler är säker i den omfattning som krävs av dess säkerhetsmodell. Säkerhetsbrister kan ändå uppstå i olika mobila terminaler eftersom tillverkarna själva implementerar javaplattformen i dem. Flertalet av de få brister i säkerheten vi funnit är inte specifika för mobila enheter med stöd för javaplattformen, utan är allmänna angrepp som går att utnyttja även på andra plattformar och enheter.

Malicious code with Java in mobile phones

Abstract

In the nearest future, the number of mobile devices supporting interchangeable applications will increase. Mobile devices, like our mobile phones, will be able to do more than setting up voice calls. Our goal with this Bachelor project is to investigate the possibility to create and distribute applications that act malicious to users. We will only concern devices implementing the Java platform. When third party developers create applications for mobile devices, the security of the Java platform will be tested. It is important that the occurrences of malicious applications are kept at a minimum because mobile devices such as mobile phones are directed towards a huge group of consumers. This is a group for which issues about technique and security should be as transparent as possible.

We have collected necessary knowledge mainly from the Internet and various specifications in digital format. These sources were used since the subject is relatively new and no books treat it yet. We have studied the background of the Java platform in mobile devices, its security model and areas were potential attacks could be made. After that, analyses of the possibilities to create and spread malicious code through these areas are made and some tests are conducted.

We have reached the conclusion that the Java platform in mobile devices is safe in the scope of its security model. We are not, however, saying that security flaws cannot appear when the Java platform is implemented erroneously in specific devices. Most of the security flaws we discovered are not specific for mobile devices supporting the Java platform, but are general and applicable to other platforms and devices as well.

Tack

Vi skulle vilja tacka följande personer:

Robin Staxhammar, vår handledare vid Karlstads universitet, för den konstruktiva kritik han givit oss angående uppsatsens innehåll och upplägg.

Rikard Skogberg, vår handledare vid Telia Mobile AB i Karlstad, för att han uppmuntrat oss och givit oss stöd i vårt arbete.

Hasse Ellström och Berndt-Uno Nyström, våra opponenter, för en givande opposition.

Övriga åtta examensarbetare vid Telia Mobile AB i Karlstad för trevligt sällskap och givande utbyte av idéer.

Övrig personal vid Telia Mobile AB i Karlstad.

Innehållsförteckning

Figurförteckning

Tabellförteckning

1 Inledning

1.1 Bakgrund

Inom en relativt snar framtid kommer många av de mobila terminaler, till exempel mobiltelefoner, som släpps på marknaden att ha inbyggt stöd för Java. Stödet fås genom en nyligen framtagen delmängd av javaplattformen kallad Java 2 Micro Edition (J2ME) och en komponent i J2ME kallad Mobile Information Device Profile (MIDP). J2ME och MIDP gör det möjligt för användare att ladda ned applikationer till sina terminaler från Internet, och därefter exekvera applikationerna lokalt i dem. Detta är en intressant teknik med många möjligheter. [2]

Det investeras mycket pengar och arbete i att få Java i små enheter att fungera. Skulle det finnas allvarliga säkerhetsluckor finns det risk för att den personliga integriteten skadas. Detta leder till minskad användning, minskad marknad och sämre avkastning för företag som utvecklar eller tillhandahåller tjänster.

Att mobila terminaler öppnas för tredjepartsapplikationer på detta sätt är nytt och väcker tankar kring virus och andra former av skadlig kod. Ett visst skydd mot användning av terminalens olika delar finns i specifikationen av J2ME men redan nu har vissa terminaltillverkare byggt ut plattformen med egna terminalspecifika funktioner. Detta är bakgrunden till varför vi i uppsatsen kommer utreda vilka möjligheter som finns att skriva kod som utför icke önskvärda operationer. Detta på uppdrag av Telia Mobile AB i Karlstad. [9][10]

1.2 Syfte och mål

Uppsatsens syfte är att utreda och testa möjligheterna att skriva skadlig kod för MIDP. Med begreppet skadlig kod menas programkod som när den exekveras eller installeras utför för användaren icke önskvärda operationer. Eftersom skadlig kod inte är skadlig förrän den exekverar i en enhet utreds även möjligheterna att få användare att ladda ned skadliga applikationer.

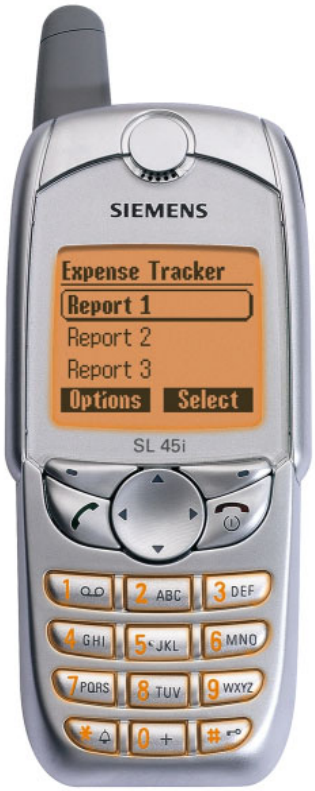
Studier utförs för att kunna analysera potentiella angreppsområden och få idéer på tänkbara attacker. Dessa attacker implementerar vi i form av tester som dokumenteras. Målet är sedan

att, genom att dra slutsatser kring vad som är möjligt och omöjligt utifrån analysen och testernas resultat, komma fram till hur säkra mobila terminaler med stöd för javaplattformen är.

1.3 Avgränsning

Uppsatsen omfattar endast nedladdningsbara applikationer skrivna för MIDP. Vi behandlar vad en applikation kan utföra, antingen ensam eller i samarbete med någon serverlösning. Tester av potentiellt skadlig kod kommer utföras på de olika terminaler som vid arbetets utförande finns tillgängliga: Motorola Accompli 008 och Siemens SL45i, se Figur 1.1 och Figur 1.2. Säkerhetsaspekter för terminalspecifik funktionalitet som inte är åtkomlig genom Java samt terminalspecifika säkerhetsbrister som inte har med Java att göra faller utanför uppsatsens avgränsning. Terminalspecifik funktionalitet åtkomlig genom Java i Siemens SL45i kommer behandlas för att utreda om utökningar av funktionaliteten i MIDP innebär säkerhetsrisker.

Figur 1.1: Motorola Accompli 008



Figur 1.2: Siemens SL45i

1.4 Uppsatsens upplägg

Uppsatsen inleds med en presentation av J2ME och en redogörelse för varför denna standard är lämplig för resurssnåla enheter. Eftersom uppsatsens syfte är att undersöka möjligheterna att utveckla skadlig kod presenteras därefter javaplattformens säkerhetsmodell och speciellt hur denna anpassats för J2ME. Efter detta följer en genomgång av den del av J2ME som benämns MIDP eftersom det är inom denna del vi skall utreda möjligheterna att utveckla skadlig kod. För att en applikation skall hamna i en av de enheter MIDP är ämnad för måste en användare ladda ned den till och installera den i en enhet. Vilka mekanismer som döljer sig bakom detta är vad som tas upp härnäst. Därefter är det dags att gå in på djupet med de systemnära delarna i J2ME. Här beskrivs hur applikationerna är uppbyggda och hur de exekveras för att kunna utröna möjligheterna att modifiera en kompilerad applikation och på så vis utveckla skadlig kod. I den sista presenterande delen av uppsatsen redogör vi kort för hur den utökade funktionaliteten hos Siemens SL45i ser ut. Detta för att visa att MIDP går att utöka med terminalspecifik funktionalitet som kan öppna nya säkerhetsluckor.

Kvarstår gör att analysera möjligheterna för utveckling och spridning av skadlig kod inom de områden vi gått igenom. Därpå följer de tester som analysen lade grund för. Tillsammans med analysen bildar testresultaten grunden för uppsatsens resultat och våra rekommendationer. Till sist presenterar vi våra slutsatser med avseende på projektet som helhet.

2 Java 2 Micro Edition

Standarden Java 2 är samlingsnamnet på ett flertal komponenter som tillsammans utgör en miljö i vilken applikationer kan utvecklas och exekveras. Gemensamt för applikationerna är att de är skrivna i programspråket Java och exekveras av ett speciellt program kallat virtuell maskin. Att använda sig av javaplattformen och en virtuell maskin gör att applikationerna blir oberoende av enhetsspecifika faktorer som hårdvara och operativsystem. Java 2 tillhandahåller funktioner som är generella gränssnitt mot den underliggande plattformen. [1]

Java 2 Micro Edition (J2ME) är en delmängd av Java 2 Standard Edition (J2SE) och javaplattformen, framtagen speciellt för små enheter med begränsade resurser. J2ME är tillräckligt generell, kompakt och effektiv för att tillverkare av sådana enheter hellre skall välja denna standard framför en egen skräddarsydd lösning. Program och applikationer som är skapade i J2ME kan köras på enheter som har en kompatibel virtuell maskin installerad. Även om resultatet blir något olika på olika enheter, blir det tillräckligt lika för att det skall vara möjligt att exekvera applikationen med samma resultat på olika enheter. I och med detta uppstår ett plattformsoberoende, vilket eftersträvas. [1][2]

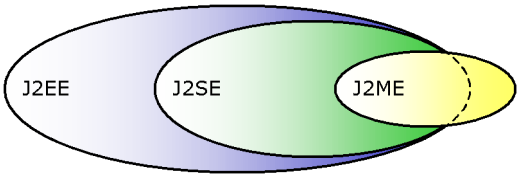
Mjukvaran i enheter i J2MEs målgrupp har fram till nu inte varit utbytbar. Enheterna har redan från början innehållt applikationer som inte går att radera eller byta ut. Användaren har inte heller haft möjlighet att installera nya applikationer på ett enkelt sätt. I enheter som stödjer J2ME är det möjligt att ladda ned och installera vilka applikationer man vill. Detta öppnar dörren för tredjepartstillverkare av programvaror. Tillverkaren bestämmer inte längre vilka applikationer som skall finnas i enheten. Vem som helst skall kunna tillverka egna applikationer och göra dem tillgängliga för allmänheten via exempelvis Internet. [2]

I avsnitt 2.1 presenteras bakgrunden till och på vilket sätt J2ME utvecklades. Detta för att införskaffa en bättre grund för förståelse av uppbyggnaden av J2ME. Avsnitt 2.2 behandlar den viktiga frågan om varför J2ME skall användas överhuvud taget. Kapitlet avslutas med en beskrivning av varje enskild komponent som utgör en del av J2ME och speciellt de delar uppsatsen kommer behandla, se avsnitt 2.3.

2.1 Historia

Grunden till standarden Java 2 är programspråket Java. Därför inleds detta avsnitt med språkets bakgrund samt en introduktion till de egenskaper som gjort Java till ett populärt språk.

Figur 2.1 illustrerar de tre stora grundpelarna i javaplattformen. Figuren visar att J2ME är en delmängd i J2SE, som i sin tur är en delmängd i J2EE. I J2ME har även viss ny funktionalitet tillförts utöver den som kommer från J2SE. J2ME specificerades efter de två andra och av den anledningen presenteras J2SE och J2EE innan J2ME.



Figur 2.1: Javaplattformens hierarki

2.1.1 Programspråket Java

Java är ett programspråk utvecklat av företaget Sun Microsystems [29]. Programspråket är objektorienterat och går att använda för att skapa applikationer som är plattformsoberoende. Sun Microsystems var tidigt ute med Java men då under namnet Oak som härrör från det träd som stod utanför det kontorsrum där utvecklaren James Gosling arbetade med designen. Oak var tänkt att användas till små programmerbara enheter på konsumentmarknaden som till exempel kaffemaskiner. Tyvärr var företaget alldeles för tidigt ute och beslutade att lägga ned projektet eftersom marknaden inte bar i början av 90-talet. Efter en tid uppstod det på nytt, då kallat Java eftersom varumärket Oak redan var upptaget. Java och dess plattform har utvecklats i snabb takt de senaste åren och har delats upp mer och mer för olika typer av slutmarknader, som servrar, vanliga persondatorer och resurssnåla enheter som till exempel mobiltelefoner. [1][12]

Programspråket Java är syntaktiskt mycket likt programspråket C++. Faktum är att när Sun Microsystems började titta på olika programspråk för olika programmerbara produkter valde företaget att börja med C++. Dock lämpade sig inte programspråket för den breda och kraftigt varierande mängd av produkter där Sun Microsystems hade sina intressen. En av Javas starkaste egenskaper är att mycket av programmeringen redan är gjord i form av olika klasser som tillhandahåller önskad funktionalitet. Programmering i Java ter sig som ett arbete att använda befintliga delar (klasser) och få dem att samarbeta. [12]

Vad som gjort Java till ett programspråk som används i stor utsträckning är javaplattformens goda möjligheter till nästintill perfekt säkerhet. Denna egenskap gör Java till ett mycket bra val för att skapa applikationer som ingår i distribuerade system. [15]

En kompilerad källkodsfil i Java exekveras ovanpå en virtuell maskin, vilket är ett program som tolkar informationen i klassfilen och behandlar den på ett adekvat vis. Genom att utveckla virtuella maskiner för olika plattformar uppnås plattformsoberoende. Att exekvera en javaapplikation i Windows skall generera samma resultat som en exekvering i Linux. Detta ser vi som ytterligare en faktor som bidrar till att Java används mycket. [16]

2.1.2 Java 2 Standard Edition

En nybörjare i javaprogrammering kommer oftast först i kontakt med J2SE. Denna standard är riktad till en stor mängd områden, men används främst i samband med så kallade applets till vanliga persondatorer. En applet är en javaapplikation menad att exekvera i en webbläsare. J2SE är i många fall komplett och innehåller numera avancerade funktioner för grafik och ljud. Den ursprungliga generella virtuella maskin som applikationer i J2SE körs ovanpå kallas Java Virtual Machine (JVM). [15]

2.1.3 Java 2 Enterprise Edition

Med höga krav på effektivitet och säkerhet riktar sig Java 2 Enterprise Edition (J2EE) mot servermarknaden och de riktigt intensiva nätverksapplikationerna med hög belastning. Som plattform för distribuerade system som nyttjar olika typer av hårdvara visar Java sin fulla styrka. Eftersom J2EE är mer inriktat på företag än privatpersoner måste företagen inhandla en licens hos Sun Microsystems för att få använda sig av standarden och kalla sina produkter för J2EE-kompatibla. Dock är själva utvecklingspaketet fritt. [11]

För J2EE har mycket kraftfulla virtuella maskiner utvecklats som ett svar på kraven för effektivitet. En högpresterande virtuell maskin utvecklad av Sun Microsystems är Java Hotspot. Forskning pågår för att ta fram ännu bättre virtuella maskiner som stödjer en avancerad typ av kompilering kallad Just In Time (JIT). [13][14]

2.1.4 Java 2 Micro Edition

Dagens resurssnåla enheter, som exempelvis mobiltelefoner, saknar de resurser som krävs för att köra J2SEs virtuella maskin JVM. För att kunna köra applikationer i Java även på dessa resursbegränsade enheter skapades J2ME. Sun Microsystems utgick från J2SE och tog bort allt som ansågs alltför resurskrävande eller utgjorde säkerhetshot. J2ME är alltså en delmängd

av J2SE. Som programmerare ser man J2ME som en reducerad version av J2SE med betydligt färre klasser och fler restriktioner. [8]

De virtuella maskiner som J2ME använder sig av är också befriade från de delar som inte är absolut nödvändiga. De två virtuella maskiner som är specificerade i dagsläget går under benämningarna Compact Virtual Machine (CVM) för resurssnåla enheter, och Kilo Virtual Machine (KVM) för mycket resurssnåla enheter. Som namnet antyder går KVM att implementera så att den tar upp minne i storleksordningen kilobytes. [8][16]

2.2 Varför använda J2ME?

Anledningarna till att just J2ME lämpar sig som utvecklingsmiljö för applikationer till resurssnåla enheter är flera. Den övergripande stora fördelen som J2ME har är att det finns tillgängligt. Detta innebär att en tillverkare av en enhet redan har en färdig standard att använda sig av istället för att utveckla en egen. J2ME erbjuder flera fördelar för såväl utvecklaren som användaren:

J2ME ger enheterna mer funktionalitet. Som exempel kan J2ME jämföras med Wireless Application Protocol (WAP), som är en teknologi som funnits längre än J2ME. WAP möjliggör för enheter att hämta information från Internet och presentera den i enhetens skärm. Javaapplikationer hämtade från Internet erbjuder mer interaktivitet för användaren än WAP, och kan göra mycket mer än bara presentera information. Dessutom är applikationerna tillgängliga även när enheten inte är uppkopplad eftersom de sparas och exekveras lokalt i enheten. [1][2]

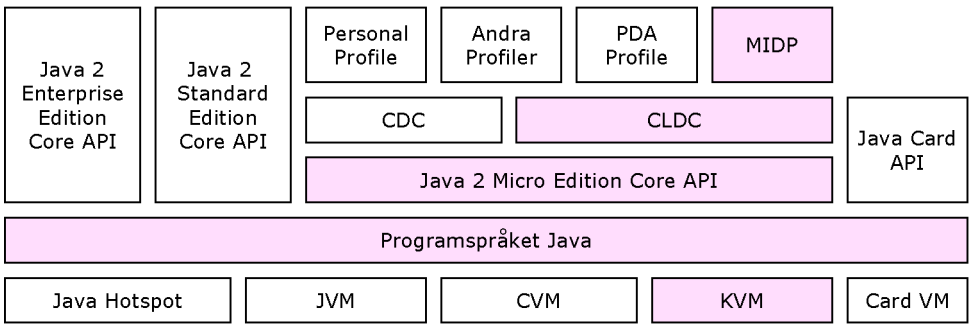
Portabiliteten som J2ME medför innefattar plattformsoberoende och dynamik i vilka program användaren väljer att ha i sin enhet. Det faktum att alla enheter som implementerar KVM kan köra alla program skrivna för KVM utan att dessa behöver kompileras om gör applikationerna plattformsoberoende. Detta gynnar utvecklaren. Att användaren själv kan välja vilka program som skall finnas i enheten genom att ladda ned dem från Internet när som helst gynnar både användaren och utvecklaren. [1][2]

En viktig aspekt är säkerheten. Eftersom vem som helst kan tillverka en applikation som kan laddas ned måste enhetens användare kunna lita på att den nedladdade applikationen inte orsakar några skador i systemet. J2ME har en mycket bra och genomarbetad säkerhetsmodell som vi presenterar i kapitel 3. Till att börja med är själva programspråket skapat på ett sådant sätt att säker och robust utveckling enkelt kan tillämpas genom till exempel undantagshantering och automatisk insamling av förbrukade objekt (Garbage collection).

Eftersom den virtuella maskinen i J2ME inte kan använda andra åtkomstmetoder mot enhetens hårdvara än de inbyggda som går att lita på, skall säkerhet kunna garanteras. Det värsta som skall kunna hända är att den virtuella maskinen slutar fungera. [1][2]

2.3 Schematisk överblick

J2ME hör till javaplattformen. En närmare inblick i strukturen presenterar många byggstenar som tillsammans ger en omfattande och komplex plattform. Figur 2.2 illustrerar plattformen. Då uppsatsen enligt specifikationen för examensarbetet endast omfattar Java i mobila terminaler kommer ett fåtal komponenter i plattformen behandlas, närmare bestämt de skuggade områdena i Figur 2.2. Innan specifikationen till en komponent i javaplattformen fastslagits glider olika områden genom experimentella utvecklingssteg in i varandra. Som exempel kan nämnas att J2ME såg dagens ljus först 1999 medan begreppet Java och mobila terminaler funnits betydligt längre. Kombinationen av dem uppstår och specificeras inte över en natt. [8]



Figur 2.2: Schematisk bild över javaplattformen [5][8][24]

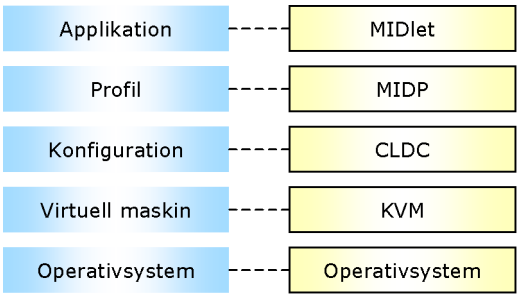
Strukturen i javaplattformen kan också komma att ändras i framtiden och på så sätt göra vissa områden inaktuella eller orsaka namnbyten. Det genomgående draget i Figur 2.2 är uppdelningen mellan de enheter som har minst resurser och de som har flest. Från smart cards med minneskapacitet på några få kilobytes, till mycket resurskrävande tillämpningar inom servrar. Ju längre åt vänster ett område är placerat i Figur 2.2, desto mer resurser i form av hårdvara krävs. [8]

Programmeringstekniken mellan olika områden i javaplattformen skiljer sig markant. Medan väldigt mycket är möjligt under J2EE, är möjligheterna begränsade i J2ME. Detta menar vissa skadar Javas ursprungliga idé om ett språk som kan köras överallt på alla maskiner. [17]

Varje nytt lager i modellen av plattformen introducerar ny funktionalitet till helheten. Ett lager placerat högre upp i Figur 2.2 innebär mer komplexitet än de underliggande. [5][8]

Figur 2.3 illustrerar den del av javaplattformen som uppsatsen behandlar, det vill säga Javaplattformen i mobila terminaler. I figuren finns lagren operativsystem, virtuell maskin (KVM), konfiguration (CLDC), profil (MIDP) och till slut applikation (MIDlet). En kort introduktion till var och ett av lagren redogörs i de följande avsnitten.

Terminalspecifika applikationsprogrammeringsgränssnitt, det vill säga utökad funktionalitet specifik för en viss enhet, tas även upp. Om sådana finns tillgängliga och används så skall de i Figur 2.3 placeras som ett lager bredvid och ovanför profilen. [35]



Figur 2.3: Schematisk bild över javaplattformen i mobila terminaler [5][8]

2.3.1 Operativsystem

Operativsystemet i enheterna har flera uppgifter. Den viktigaste för J2ME är att underlätta gränssnittet mot hårdvaran. Operativsystemet tillhandahåller åtkomsten mot hårdvaran så att ett ovanliggande lager kan nå hårdvaran via operativsystemet. Operativsystemen i enheter som implementerar en viss konfiguration, se avsnitt 2.3.3, har gränssnitt som den virtuella maskinen som implementerar konfigurationen kan använda sig av. Tack vare detta blir den virtuella maskinen hårdvaruoberoende. [8]

2.3.2 Virtuella maskiner

En kompilerad källkodsfil i Java kallas klassfil och innehåller en form av instruktioner som kallas bytekod. En virtuell maskin är ett program som exekverar applikationer skrivna i Java genom att tolka instruktionerna i klassfilernas bytekod och översätta dem till instruktioner som det underliggande operativsystemet förstår och kan utföra. Det finns egentligen inte någon bestämd virtuell maskin som måste användas utan det går bra med vilken som helst. Kravet är att den stödjer de minimikrav som står i specifikationen för konfigurationen, se avsnitt 2.3.3, som skall fungera tillsammans med den virtuella maskinen. [8]

KVM, som är en reducerad version av den virtuella maskin som specificerats för J2SE (JVM), är mycket liten och kompakt till storleken vilket gör att den lämpar sig för enheter med begränsade resurser. För att få den så liten har mycket av den funktionalitet som JVM har tagits bort och annan funktionalitet ändrats till en kompromisslösning. KVM är den virtuella maskin som specificerats speciellt för de resurssnåla mobila terminaler som den här uppsatsen behandlar. I enheter med mer resurser, men som ändå klassas som del i J2MEs målgrupp, implementeras den virtuella maskinen CVM. I avsnitt 6.1 presenteras den virtuella maskinen mer i detalj. [5][8]

2.3.3 Konfigurationer

En konfiguration specificerar funktionalitet som är gemensam för alla enheter som ingår i en viss målgrupp. Endast den mest grundläggande funktionaliteten ingår i en konfiguration. Konfigurationer kräver ingen bestämd hårdvara, men har vissa minimikrav på dess prestanda och behöver ett operativsystem för att kommunicera med den. [8]

Konfigurationen är starkt knuten till den virtuella maskinen, men behöver inte använda sig av samma virtuella maskin som den specificerats för. Anledningen till det starka förhållandet är att den virtuella maskinen innehåller redan kompilerade versioner av biblioteken i konfigurationen. Detta dels för effektivitetens skull, men i J2MEs fall främst för att utvecklaren inte skall kunna införa alternativ till eller ändra i den grundläggande funktionaliteten. Det skulle öppna säkerhetsluckor om det var möjligt att göra så. [8]

Connected Device Configuration (CDC) är en konfiguration som ingår i J2ME och som riktar sig till enheter som inte är lika resursbegränsade som de som den här uppsatsen behandlar. Exempel på enheter som kan använda CDC är så kallade set-top boxes till TV-mottagare. Eftersom de enheter CDC är tänkt för oftast har en större mängd resurser i form av processorkraft och minne, har CDC mer funktionalitet än CLDC, se nedan. Det är fullt möjligt att köra applikationer skrivna för CLDC på en enhet med stöd för CDC, men inte tvärtom. [5]

Connected, Limited Device Configuration (CLDC) är som namnet antyder en reducerad version av CDC. CLDC riktar sig till väldigt resursbegränsade enheter såsom handdatorer och mobila terminaler, och är den konfiguration som uppsatsen behandlar. Mer specifikt om vad CLDC erbjuder och vad den har för restriktioner finns att läsa i avsnitt 4.1.1. [5]

2.3.4 Profiler

En profil är en utökning av en konfiguration med funktionalitet riktad till en viss typ av enheter inom konfigurationens målgrupp. Profilerna introducerar grafiska gränssnitt och erbjuder också möjligheten att behandla beständig data för att låta användaren spara och läsa filer. Vidare skall de implementera någon form av protokoll för nätverkskommunikation, såsom exempelvis HTTP, FTP eller TCP/IP. [16]

Personal Digital Assistance Profile (PDA Profile) är en profil för handdatorer och andra enheter som har fler användningsområden än de enheter den här uppsatsen behandlar. [5]

Mobile Information Device Profile (MIDP) är en profil som riktar sig till mobila terminaler såsom mobiltelefoner. MIDP diskuteras i detalj i kapitel 4. [5][9]

2.3.5 Utökade applikationsprogrammeringsgränssnitt

Tillverkare som anser att konfigurationen och profilen inte erbjuder tillräcklig funktionalitet för deras enhet kan utöka denna med egna funktionsbibliotek. Detta möjliggör mer plattformsberoende funktionalitet. Det utökar därmed möjligheterna för utvecklare och upplevelserna för användarna. Mer om utökade applikationsprogrammeringsgränssnitt finns att läsa i kapitel 7. [10]

2.3.6 Applikationer

Applikationerna är de färdiga programmen som exekveras ovanpå den virtuella maskinen i enheterna. En applikation som utvecklats för MIDP kallas för MIDlet. Exempel på applikationer som går att utveckla är e-postklienter, avancerade adressböcker, nätverksspel och till och med kalkylprogram kompletta med diagram. Även program innehållande skadlig kod är applikationer. [16]

3 Säkerhetsmekanismer

Det råder ingen tvekan om att säkerhet i syfte att skydda den personliga integriteten och att skydda företagsvärdlen har en framträdande roll i arbetet att utveckla applikationer. Det är extremt viktigt att tillämpningar som utvecklas idag och riktar sig mot den stora marknaden inte innehåller luckor i säkerheten som kan utnyttjas av illasinnade människor. Genom en övergång till applikationer som exekverar i ett distribuerat system exponeras applikationen över ett större geografiskt område. Detta leder till att antalet angreppspunkter ökar eftersom fler datorer ingår i systemet. Sådana system ställer höga krav på säkerhet i form av kryptering, autenticering, certifikat med mera, det vill säga allmänna säkerhetsmekanismer utöver javaplattformens grundläggande säkerhetsmodell, se avsnitt 3.1. [20]

Vad är det då som gör javaplattformen lämpad för utveckling av applikationer som tillgodoser de ovan nämnda kraven? För att kunna besvara den frågan tar vi upp den så kallade sandlådemodellen i avsnitt 3.2. Detta begrepp används ofta som en beskrivning på säkerhetsmodellen i javaplattformen. Sandlådemodellen innebär att en applikation har ett visst minnesområde att röra sig inom och inte kommer åt resurser utanför det. En användare tvingas att sitta still i en tänkt sandlåda och leka utan att grusa ner gräsmattan utanför den. [4][16]

Avsnitt 3.3 avlutar kapitlet med en jämförelse mellan säkerheten i J2ME och J2SE. J2SE har en omfattande virtuell maskin med säkerhetsmekanismer och speciella klasser för säkerhet, medan J2ME inte innehåller funktionalitet som ska kunna orsaka säkerhetsluckor. Javaplattformens sandlådemodell bibehålls i båda fallen, men på olika vis.

3.1 Allmänna säkerhetsaspekter vid programutveckling

Få är de plattformar som inte möjliggör utveckling av skadlig kod. Genom att ge programmeraren frihet och tillgång till resurser som kan utnyttjas på andra sätt än det var tänkt öppnas säkerhetsluckor. Det finns idag ett otal exempel på kod vars syfte är att orsaka skada på ett eller annat vis. Vad de flesta har gemensamt är att de utnyttjar befintliga säkerhetshål i känd mjukvara. Det hela bygger på att användaren litar på den kod som exekveras i enheten eller datorn, och det är endast genom att lura användaren som den skadliga koden kan placeras i dennes system. [20][21]

Först ges en översiktlig presentation av de mekanismer som ligger utanför javaplattformens säkerhetsmodell men som bidrar till att säkert kunna utbyta information mellan olika källor som anses tillförlitliga. Ännu en gång bygger detta på tillit eftersom en användare blint måste lita på att källan denne direkt eller indirekt kommunicerar med verkligen är den den utger sig för att vara.

3.1.1 Kryptering

Kryptering går i huvudsak ut på att man ersätter en bit data med en lika stor bit ny data som inte kan förutsägas om man inte känner till den algoritm som genererar den nya datan. Ungefär som ett chiffer alltså. Ofta handlar det om så kallade krypteringsnycklar. Vissa av dessa nycklar kan användas både för att kryptera och dekryptera, men det finns också nycklar som enbart kan utföra en av algoritmerna och därför används i par. Det finns en hel del standarder inom kryptering och man kan säga att dagens kryptering är säker. Detta eftersom tiden det tar att forcera ett krypto är långt från försumbar. Som exempel kan nämnas att det 1999 gjordes ett försök att dekryptera en enkel textsträng skapad med krypteringsstandarden RSA. Det tog då drygt tjugotvå timmar för en specialdesignad superdator hopkopplad med nära hundratusen datorer världen över att lyckas. [22]

För att automatisera kryptering och dekryptering vid datakommunikation har olika säkra protokoll som bygger på redan existerande osäkra protokoll skapats. De har den egenskapen att all data de skickar och tar emot är krypterad. Därmed blir känslig data mycket svår att tyda för en utomstående person. Exempel på säkra protokoll är Secure Sockets Layer (SSL) och Secure Shell (SSH). [22]

3.1.2 Autenticering

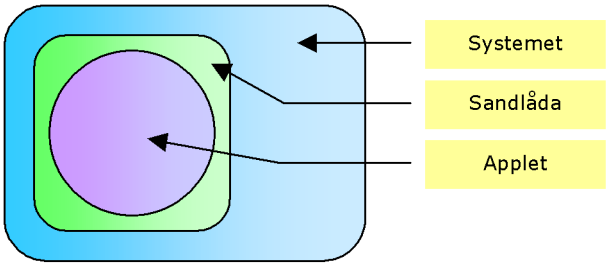
Ett vanligt förfarande är att låta en enhet koppla upp sig mot en server som kanske befinner sig på andra sidan jordklotet. Hur kan användaren då vara säker på att ha kommit rätt? Det är helt möjligt att en illasinnad person ställer sig mellan enheten och servern och låter all kommunikation gå genom sitt system där den kan fångas upp, analyseras och förvanskas för att sedan skickas vidare. Autenticering spelar här en stor roll och i korthet kan man säga att systemet bygger på kryptering och signering. Signering innebär att sändaren inkluderar en unik identifierare i sitt meddelande som gör att mottagaren kan vara säker på vem sändaren är och att meddelandet inte förvanskats på vägen. [22]

3.1.3 Certifikat

Vad gäller nedladdning av program och tillit till upphovsmän hjälper inte autenticering. Ett sätt att öka tilltron till en server varifrån en applikation laddas ned är att sätta upp olika typer av certifikat. Certifikat utfärdas av en aktör som användaren måste lita på. Litar användaren på aktören, kan denne nämligen också lita på det program som försetts med aktörens certifikat. Därför kan det enligt vår uppfattning vara bra att hålla sig informerad om vilka de stora aktörerna är och hur andra företag har löst certifieringen. [22]

3.2 Sandlådemodellen

Javaplattformens sandlådemodell är en säkerhetsmodell som innebär att många operationer är förbjudna eller kan förbjudas under utvecklingen av en applikation genom att upprätta olika policies. En applikation får på detta sätt endast tillgång till bestämda delar av systemet och detta område är den tänkta sandlådan. En applet är, som nämnts i avsnitt 2.1.2, en javaapplikation menad att exekvera i en webbläsare. För sådana applikationer blir sandlådemodellen tydlig eftersom applikationen inte kan bilda sig en uppfattning om vilket system den kör på eller komma åt systemet på annat sätt än via säkra kanaler och gränssnitt. En enkel illustration kan ses i Figur 3.1. [4]

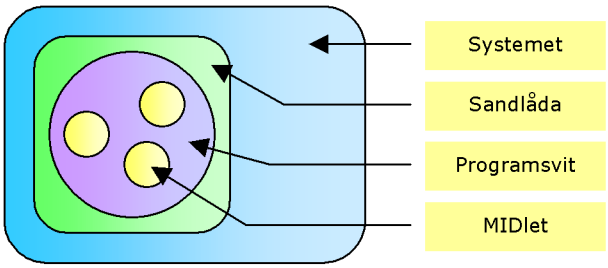


Figur 3.1: Sandlådemodellen för applets [4]

Enligt traditionen är en användare tvungen att lita på den programvara som exekveras. Säkerhet uppnås genom att bara införskaffa programvara från de tillverkare man litar på. När den skadliga programvaran väl har kommit in i systemet har den som regel tillgång till allt. Genom att använda sig av javaplattformens säkerhetsmodell kan användare även låta programvara från mer okända tillverkare exekveras i sina enheter. [4]

Figur 3.2 illustrerar en något annorlunda bild som beskriver sandlådemodellen för MIDlets. En eller flera MIDlets ligger grupperade i en så kallad programsvit, och dessa delar på ett minnesutrymme som de kan kommunicera genom. J2MEs sandlådemodell är mindre komplex än javaplattformens ursprungliga modell eftersom J2ME är en reducerad del av

javaplattformen som inte stödjer funktionalitet som kräver speciella säkerhetsmekanismer. Mer om detta i avsnitt 3.3. [4]



Figur 3.2: Sandlådemodellen för MIDlets [4]

3.2.1 Programspråket Java

En närmare granskning av komponenterna i säkerhetsmodellen visar att språket i sig är designat med tanke på säkerhet. Stöd för direkt access till minnet med pekarvariabler saknas och undantagshanteringen är omfattande och har en klar plats i designen av systemet. Det är i princip omöjligt att ta sig utanför det minne som är tilldelat applikationen. Om ett medvetet eller omedvetet fel skulle inträffa kommer undantag att kastas vilket leder till att en felhanterare körs eller att applikationen avslutas utan att skada det övriga systemet. Denna grundläggande säkerhet är även inkluderad i J2ME eftersom samtliga komponenter i javaplattformen använder samma programspråk. [1][2][15]

3.2.2 Den virtuella maskinen

Alla javaprogram körs ovanpå en virtuell maskin. J2SEs virtuella maskin JVM har en hel del inbyggda säkerhetsmekanismer. Som exempel kan nämnas följande:

- Säker konvertering mellan datatyper. Detta inkluderar att olika datatyper kan blandas endast på ett specificerat vis, samt att en viss datatyp alltid upptar en bestämd storlek i minnet.
- Frånvaro av pekarvariabler och pekaroperationer. Detta gör att minnesadresser inte kan modifieras fritt och att minne som inte tillhör applikationen inte kan adresseras.
- Automatisk skräpinsamling (garbage collection). Detta är en mekanism som medför att programmeraren inte behöver frigöra allokerat minne explicit. Risken för minnesläckage, användning av redan frigjort minne, samt frigöring av redan frigjort minne elimineras därmed helt.
- Intervalltestning av index till arrayer. Denna egenskap förhindrar adressering av minne som inte tillhör arrayen och därmed även minne som inte tillhör applikationen.

- Kontroll av ogiltiga referenser. Detta är ytterligare en egenskap som förhindrar otillåten adressering.

[2][4][15]

Utöver detta kan JVM i viss mån utforska programmets sekvens i förväg för att rapportera om eventuella fel eller andra problem. Det är inte mycket som kan göras utan att JVM vet om det. Det finns inget sätt för programmeraren att veta var det allokerade minnet ligger någonstans eftersom det är upp till tillverkaren av en virtuell maskin att specificera hur minnet skall organiseras, och därmed också upp till tillverkaren om denne offentliggör denna specifikation eller inte. [1][8][15]

Även KVM har kontroll över exekveringen om än i mindre skala. Detta beror på de begränsade resurser som avsätts åt den virtuella maskinen i de enheter den är ämnad för. Kontrollen sker här utan någon undersökning av vad som kommer att ske i framtiden. För att detta skall fungera måste klassfilen där programmet finns lagrat som bytekod genomgå en särskild process innan den överförs till och exekverar på en enhet. Mer om denna process finns i avsnitt 3.2.3 och 6.3. [4]

3.2.3 Verifiering

Innan en applikation för javaplattformen tillåts exekvera genomgår den en process som kallas verifiering. Vid verifieringen kontrolleras att applikationens klassfil är giltig och inte innehåller otillåtna instruktioner. Den verifiering som sker då KVM laddar en klassfil är mycket mindre komplex än den verifiering som JVM utför. Detta för att få KVM mer kompakt. För att kompensera den reducerade verifieringen i KVM finns en process som kallas preverifiering som måste utföras innan applikationen kan verifieras och exekveras av KVM. Preverifieringen sker utanför den mobila enheten, efter att applikationen kompilerats. Det går att installera en applikation som inte preverifierats, men KVM kommer inte att kunna verifiera den och därmed inte exekvera den. Vid preverifieringen behandlas en applikations klassfiler på ett speciellt sätt som JVM ignorerar men KVM förstår och använder sig av för att kunna verifiera och exekvera applikationens klassfiler. Se avsnitt 6.2.3 och 6.3 för en mer detaljerad beskrivning av vad preverifieringen gör. [4]

3.2.4 Metoder skrivna i andra programspråk

Javaplattformen har stöd för anrop av metoder och moduler som är skrivna i till exempel C++ eller Assembler. Ett sätt att komma runt javaplattformens gränssnitt mot hårdvaran är att använda sig av så kallade ”native methods”, alltså alternativa hårdvarugränssnitt skrivna i ett

annat programspråk. Dessa laddas in som dynamiska funktionsbibliotek. Javaplattformens egna gränssnitt mot hårdvaran går att lita på, men det är möjligt att ett sådant som är tillverkat av en tredje part inte går att lita på. För att slippa detta problem har man helt tagit bort möjligheten att använda sig av andra gränssnitt mot hårdvaran än de som finns implementerade i enheten eller samexisterar med webbläsaren. Detta gäller både för applets i J2SE och för MIDlets i J2ME. Sandlådemodellen upprätthålls här på ett effektivt sätt. [15]

3.3 Jämförelse mellan J2SE och J2ME

Ur säkerhetssynpunkt finns några viktiga skillnader mellan J2SE och J2ME. En stor bidragande orsak till skillnaderna är att J2SE körs på maskiner med mer resurser i form av minne, medan de enheter J2ME är ämnat för är relativt begränsade på det området. [4]

I J2SE finns ett antal områden som öppnar dörrarna för skadlig kod. Bland annat ett stort bibliotek med funktioner och möjligheten att ladda in alternativa gränssnitt mot hårdvaran. I de konfigurationer och profiler som finns i J2ME idag har antalet funktioner i standardbiblioteket reducerats kraftigt och möjligheten att använda alternativa gränssnitt mot hårdvaran eliminerats. [4][18]

J2SE har två mekanismer som benämns ”Security manager” och ”Protection domains”. Dessa kan specificera vilka operationer som tillåts användas och vilken kod som tillåts exekvera. Detta bland annat för att de säkerhetsluckor som öppnas då andra gränssnitt mot hårdvaran än javaplattformens egna används inte skall gå att utnyttja av illasinnade programmerare. I J2ME finns mekanismerna inte med på grund av utrymmesskäl och för att möjligheten att använda andra gränssnitt mot hårdvaran än javaplattformens egna är borttagen. [4]

En klass för J2SE genomgår en verifiering när den laddas i JVM. J2MEs KVM innehåller som tidigare nämnts i avsnitt 3.2.3 bara en del av denna verifiering och den del som inte finns med, preverifieringen, måste köras innan klassen kan exekveras ovanpå KVM. I avsnitt 6.3 behandlar vi preverifiering mer ingående. [4][18]

Applikationer skrivna i Java verkar vara säkra men säkerhet kan åstadkommas på olika sätt. Medan J2SE använder en omfattande virtuell maskin innehållande säkerhetsmekanismer och speciella klasser för säkerhet, implementerar inte J2ME funktionalitet som skall kunna orsaka säkerhetsluckor. I J2ME har de delar som skulle kunna användas i skadligt syfte helt sonika uteslutits. Javaplattformens sandlådemodell bibehålls i båda fallen, men på olika vis.

4 Mobile Information Device Profile

Mobile Information Device Profile (MIDP) är en profil i J2ME som tillsammans med konfigurationen Connected, Limited Device Configuration (CLDC) är ämnad för resurssnåla mobila terminaler med inbördes variation i utseende och funktionalitet. En applikation utvecklad för MIDP kallas för MIDlet. En MIDlet skall utan att modifieras kunna köras både på en mobil terminal med stor färgskärm och flera funktionsknappar och en enhet med en liten svartvit skärm utan funktionsknappar. Detta ställer vissa krav, vilka presenteras i avsnitt 4.1. [9]

Utvecklingen av en MIDlet skiljer sig något från annan mjukvaruutveckling. Det finns steg som är speciella just för MIDlets och som för uppsatsens inriktning är viktiga. Därför redogörs i avsnitt 4.2 för hur livscykeln för en MIDlet ser ut.

MIDP är för en utvecklare ett bibliotek med klasser och funktioner som kan användas för att underlätta programmeringen. CLDC erbjuder en väldigt generell och grundläggande funktionalitet, medan MIDP innehåller funktionalitet som är mer specifik för mobila terminaler. I avsnitt 4.3 ges en kort presentation av vad profilens och konfigurationens applikationsprogrammeringsgränssnitt tillhandahåller. [8][9]

En analys av områden som skulle kunna utnyttjas för att skapa skadlig kod med funktionaliteten i CLDC och MIDP finns i kapitel 8, avsnitt 8.1. Vår utredning av CLDC och MIDP har inte resulterat i upptäckten av några stora hål i säkerheten. Däremot har vi funnit att möjligheten finns att göra en användare irriterad eller förvillad.

4.1 Arkitektur

4.1.1 Connected, Limited Device Configuration

CLDC är den konfiguration som ligger till grund för profilen MIDP. Se Figur 2.2 och Figur 2.3 för illustrationer av detta. En konfiguration specificerar den funktionalitet som är gemensam för de enheter som ingår i konfigurationens målgrupp. I CLDCs fall är dessa enheter mobila terminaler, handdatorer och andra mindre enheter med begränsade resurser samt möjlighet till internetuppkoppling. [8]

För att använda CLDC behövs ingen bestämd hårdvara med given design. Det är upp till tillverkaren av enheten att implementera CLDC och den virtuella maskin (KVM) som

applikationen skall exekveras ovanpå. De enda kraven riktas till mängden beständigt och icke beständigt minne, samt processorn. Minneskraven är 128 kB beständigt minne för lagring av KVM och CLDC-biblioteken och 32 kB icke beständigt minne för att kunna exekvera KVM och skapa objekt av klasser. Processorn skall ha 16 eller 32 bitars register. [8]

KVM kommunicerar med enhetens hårdvara via dess operativsystem. För att KVM skall kunna kommunicera med operativsystemet krävs att det anpassats för att tillhandahålla de funktioner som KVM använder. Alternativet är att KVM anpassas för att använda de funktioner som operativsystemet redan tillhandahåller. Tillverkaren behöver med andra ord anpassa minst en av dessa två entiteter. Exempel på tjänster som operativsystemet med tanke på CLDCs krav inte behöver ge några garantier för är väntetider och möjligheten att parallellt exekvera två eller flera trådar inom en process. [8]

Bland den funktionalitet som CLDC implementerar kan nämnas följande:

- Programspråket Java och virtuella maskiner. CLDC tillhandahåller funktionalitet som ligger nära programspråket. Med detta menar vi klasser och metoder som är nära sammanlänkade med Java och kan ses som utökningar av själva språket, som exempelvis utökade egenskaper och möjligheter för primitiva datatyper och undantagshantering av division med noll. KVM är försett med kompilerade versioner av CLDCs rotbibliotek (se nästa punkt) som inte går att ersätta med alternativa sådana.
- Javas rotbibliotek. Dessa bibliotek innehåller systemnära och triviala delar som hanterar applikationens exekvering och tillhandahåller gränssnitt mot det yttre systemet. Här finns också den funktionalitet som ligger nära programspråket som nämndes i föregående punkt.
- In- och utmatning av data. Grundläggande funktionalitet för dataströmmar som exempelvis inmatning från ett tangentbord eller utmatning till en skärm.
- Nätverksanslutning. En speciell klass, `Connector`, finns tillgänglig för att hantera kommunikation över nätverk. Klassen implementerar ingen nätverksanslutning, men finns för att stödja möjligheten till en sådan i lager ovanför CLDC i Figur 2.3. I avsnitt 4.1.2 nämns att MIDP, som ligger i lagret ovanför CLDC, implementerar en HTTP-uppkoppling.
- Säkerhet. Genom ett begränsat antal fördefinierade klasser och ingen möjlighet att använda annan hårdvarunära funktionalitet än enhetens egna bibehålls javaplattformens säkerhetsmodell.

[8]

Följande är exempel på funktionalitet som CLDC inte stödjer, utan lämnar åt ett överliggande lager i javaplattformen. Detta på grund av att konfigurationer ska vara så generella som möjligt för sina målgrupper och låta profilerna specialisera sig på en bestämd typ av enhet:

- Livscykeln för applikationen i enheten. Hur installation, exekvering och borttagning av en applikation går till specificeras inte av CLDC.
- Användargränssnitt. CLDC har ingen funktionalitet för att visa grafiska objekt på en skärm eller ta emot instruktioner från funktionsknappar. Detta på grund av att enheterna i CLDCs målgrupp kan ha alltför olika fysiska förutsättningar i form av in- och utmatning.
- Hantering av händelser. Stöd för anrop av en viss kodsekvens när till exempel en knapp trycks ned saknas.

[8]

Nedan följer exempel på områden som inte stöds i CLDC på grund av begränsade resurser i målenheterna. Dessa områden är inte menade att implementeras i ett högre lager just på grund av att de kräver alltför mycket resurser: (Stöd för dem finns i CLDC 1.1, som presenteras i avsnitt 4.4.)

- Flyttal. Detta främst eftersom hårdvaran i enheterna inte antas stödja flyttal och simulering av dem i mjukvara anses alltför resurskrävande och ineffektivt.
- Felhantering. Felhanteringen finns kvar, men i mycket begränsad skala. Endast de mest fundamentala felhanterarna är implementerade i CLDC.
- Destruering (finalization) av instansierade objekt. När ett instansierat objekt inte används längre och skall tas bort av den automatiska skräpinsamlaren finns ingen metod ämnad för städning som anropas. Med städning menas till exempel att stänga öppnade filer eller spara undan beständig data.

[8][24]

Begränsningar i CLDC och KVM som är direkt förknippade med säkerhet och programspråket Java finns listade i avsnitt 3.2.2.

4.1.2 Mobile Information Device Profile

Profilen MIDP utökar funktionaliteten genom att lägga till funktioner som inte finns i CLDC. MIDP är inriktat på kommunikation med användaren genom olika grafiska användargränssnitt samt kommunikation med andra enheter genom uppkoppling till Internet.

Eftersom en MIDlet skall kunna exekveras på en mängd olika typer av mobila terminaler där utseendet kan skilja sig kraftigt ställs stora krav på generalitet och funktioner för att kunna beskriva användargränssnitt mer abstrakt. Med detta menas att det är upp till enheten att visa till exempel en inmatningsruta på bästa sätt och det programmeraren behöver göra är att instruera applikationen att visa inmatningsrutan, eventuellt med några parametrar för inställningar. Samma abstraktion används för funktionsknappar: Programmeraren anger vilka funktioner som skall finnas på funktionsknapparna, och det är sedan upp till enheten att möjliggöra val av funktionerna även om antalet knappar är färre än antalet begärda funktioner. [9]

Med MIDP följer även möjligheten att tända och släcka enskilda punkter på skärmen. Denna möjlighet är ämnad främst för utveckling av spel och andra tillämpningar där exakt grafisk presentation är viktig. Dock är variationen av utseende på olika enheters skärmar ett hinder för portabilitet av denna typ av applikationer. [9]

Det finns bara ett sätt för en MIDlet att spara undan information lokalt på den mobila terminalen, nämligen att använda sig av Record Management System (RMS). Detta system låter en applikation spara undan data i beständigt minne som sedan kan läsas in igen vid ett senare tillfälle. RMS öppnar också möjligheten för MIDlets att kommunicera med varandra genom att spara undan information som filer som andra MIDlets kan läsa. Kravet för att två eller flera MIDlets skall kunna dela filer är att de använder samma beständiga minnesområde i enheten. Detta åstadkoms, som tidigare nämnts i avsnitt 3.2, genom att placera dem i en så kallad programsvit. Skälet till att MIDlets som inte ligger i samma programsvit inte delar samma minnesutrymme är att skydda filerna från att bli exponerade och därmed kanske förvanskade eller raderade av andra applikationer. [9]

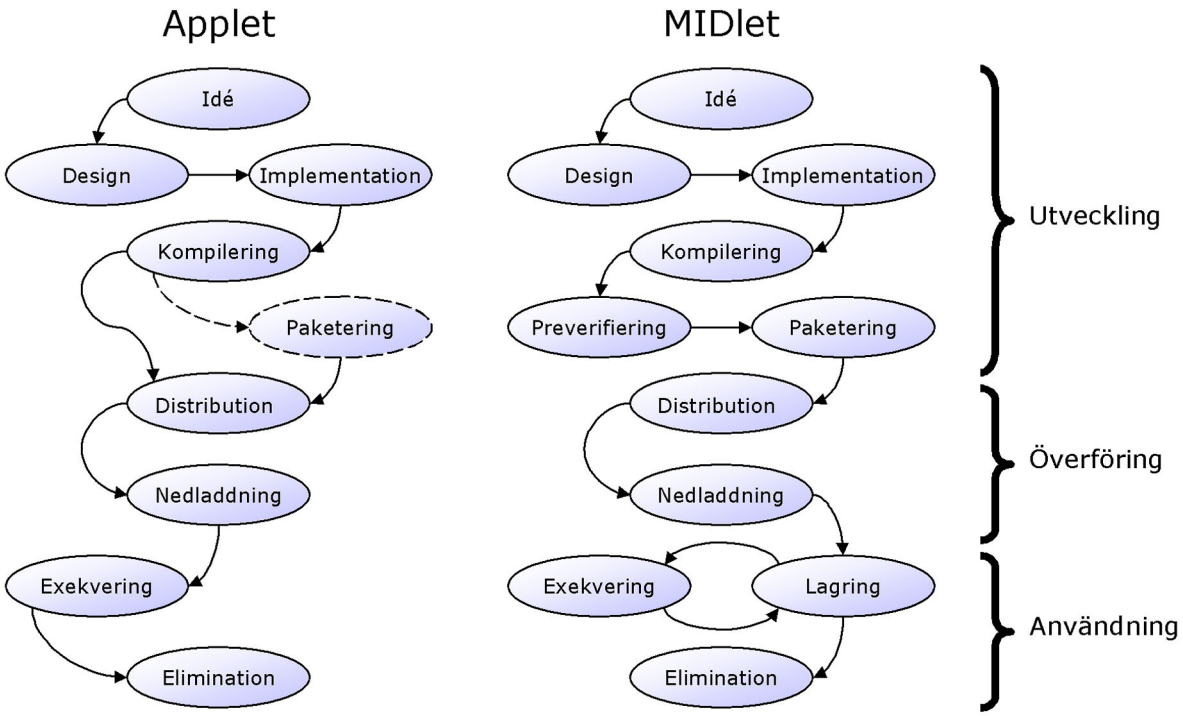
Enligt specifikationen skall MIDP implementera stöd för åtminstone nätverksprotokollet HTTP. Tillverkare av mobila terminaler kan därmed implementera fler protokoll och ändå benämna sina terminaler som kompatibla med MIDP. Nätverksuppkopplingarna upprättas genom CLDCs klass `Connector` på det vis som beskrivs i avsnitt 4.3. [9][37]

4.2 En MIDlets livscykel

Som påpekats på ett flertal ställen tidigare är en MIDlet en applikation som är utvecklad i MIDP. En parallell kan dras till namngivningen av javaapplikationer som exekverar genom en webbläsare, applets. Utvecklingen av MIDlets liknar enligt vår uppfattning utvecklingen av applets. Även exekveringen av MIDlets och applets är likartad. Främst tänker vi då på att

båda exekverar ovanpå virtuella maskiner och använder sig av javaplattformens sandlådemodell, se avsnitt 3.2. Det som skiljer de två typerna av applikationer åt är några extra steg i utvecklingsprocessen för MIDlets och sättet en MIDlet distribueras och laddas ned på. Det faktum att en MIDlet lagras lokalt i en enhets beständiga minne medan en applet inte behöver lagras lokalt medför också vissa skillnader. [6]

I följande avsnitt kommer en MIDlets livscykel presenteras. Skillnader i livscykeln mellan MIDlets och applets tas upp där de förekommer för att förtydliga vad som är speciellt i en MIDlets livscykel. Anledningen till att vi valt att jämföra MIDlets med applets är för att de båda påminner mycket om varandra. Figur 4.1 illustrerar en MIDlets och en applets livscykel från utveckling, via överföring, till användning. Figurens delar behandlas i de följande avsnitten.



Figur 4.1: Livscykeln hos en applet och en MIDlet

4.2.1 Utveckling

Utvecklingen av MIDlets genomgår ett antal faser. Först måste en bärande idé leda till en design som kan realiseras. Realiseringen sker medelst kod skriven i programspråket Java och som använder MIDPs applikationsprogrammeringsgränssnitt. När källkoden till applikationen är färdigskriven kompileras den. Varje skapad klass översätts till bytekod och denna sparas i klassfiler. Som nämnts i avsnitt 3.2.3 måste klassfilerna efter kompileringen genomgå en yttre verifiering kallad preverifiering som förändrar innehållet i dem och förbereder dem för intern

verifiering och exekvering genom att förse dem med extra information som den mobila enhetens virtuella maskin känner igen. Preverifiering sker inte för applets eftersom den verifiering som JVM utför inte kräver någon preverifiering. I avsnitt 6.2.3 och 6.3 behandlar vi preverifiering och den extra information processen tillför klassfiler. [6]

Om applikationen består av flera MIDlets måste de tillhöra samma programsvit. Paketering gör det möjligt att samla en eller flera MIDlets i samma programsvit. Paketeringen komprimerar dessutom alla klassfiler som ingår i programsviten och lägger dem i en arkivfil, kallad Java Archive (JAR). I JAR-filen finns också en så kallad manifestfil som innehåller information om vad som ingår i programsviten, var den kan laddas ned och versionsinformation. Vid sidan av denna fil skapas ytterligare en fil, Java Application Descriptor (JAD), som i princip innehåller samma information som manifestfilen, men inte är inbakad i JAR-filen. Manifestfilen och JAD-filen är läsbara textfiler. [6][28]

En applet består av en enda applikation och ingår därför inte i någon programsvit. Däremot kan även en applet komprimeras och arkiveras i en JAR-fil, men en manifestfil och en JAD-fil skapas då inte eftersom de inte används. [6]

Att utveckla en applikation i MIDP påminner på många vis om utvecklingen av andra typer av applikationer i Java då programspråket i sig är detsamma. Vid programmeringen skapas klasser som eventuellt ärver eller använder andra klasser, och metoder modifieras eller läggs till så att ett förväntat resultat uppnås. Metodernas kod exekveras alltid sekventiellt. För att underlätta utvecklingen av MIDlets kan en utvecklare använda sig av kompletta utvecklingsmiljöer som integrerar kodskrivning, kompilering, preverifiering och paketering. Utvecklingsmiljöerna finns både som kommersiella produkter och som gratisalternativ. Utvecklaren kan även använda sig av en vanlig enkel textredigerare och utföra kompilering, preverifiering och paketering via en kommandotolk. Utvecklingsmiljön vi använt heter Wireless Toolkit och tillhandahålls av Sun Microsystems [30]. I Wireless Toolkit ingår några så kallade emulatorer. En emulator är ett program som emulerar hur applikationen kommer bete sig när den exekverar i den riktiga enheten. Detta innebär att en utvecklare efter små ändringar i sin applikation inte behöver överföra den till en verklig enhet för att provköra den. Vid större ändringar kan det däremot vara en bra idé att använda en verklig enhet eftersom det visat sig att emulatorerna trots allt inte återspeglar verkligheten så bra i alla lägen. [6]

Den fas som återstår efter det att applikationen är färdig och har testats i en emulator eller i en verklig enhet, är att distribuera den till slutanvändaren.

4.2.2 Överföring

En färdig MIDlet distribueras vanligtvis via en server som är åtkomlig via Internet. I kapitel 5 presenteras nedladdningsförfarandet och dess egenskaper. I följande stycke sammanfattas kort hur en utvecklares MIDlet vanligtvis hamnar i användarens mobila terminal:

Användaren ansluter sin enhet till Internet och använder dess internetläsare för att navigera till en domän där en utvecklares MIDlet finns angiven i form av en hyperlänk. Användaren väljer att följa länken varefter nedladdningen av applikationen kan påbörjas. Applikationen sparas i enhetens beständiga minne, och när överföringen är klar finns utvecklarens MIDlet lagrad i användarens enhet, klar att exekvera. [25]

En applet överförs till användarens dator på ett liknande vis. Skillnaden är att en hyperlänk i webbläsaren refererar direkt till applikationens uppstartande klassfil som datorns virtuella maskin börjar ladda ned och tolka, men utan att lagra i det lokala beständiga minnet.

4.2.3 Användning

Det är olika från terminal till terminal hur en applikation lagras och hur en användare gör för att köra den. Är valet väl gjort startas enhetens virtuella maskin ovanpå vilken applikationen exekverar lokalt. När användaren tröttnat på applikationen eller vill frigöra minne för att göra plats för en annan kan den tas bort från terminalen permanent. Eventuella filer som applikationen skapat och lagrat i sin beskärda del av minnet raderas samfällt med applikationen. [6][27]

När en MIDlet exekverar har den virtuella maskinen fullständig kontroll över vad som sker eftersom det är den som tolkar instruktionerna i klassfilerna och instruerar terminalens operativsystem att utföra dem. Enhetens operativsystem är den virtuella maskinen övermäktigt och kan när som helst avsluta eller tillfälligt avbryta en MIDlets exekvering. [16]

Eftersom en applet inte lagras lokalt i beständigt minne så försvinner den i samma ögonblick användaren slutar använda den.

4.3 Applikationsprogrammeringsgränssnitt

Ett applikationsprogrammeringsgränssnitt är ett bibliotek med färdiga klasser och funktioner redo att användas. I detta avsnitt presenteras övergripande utvalda delar av den funktionalitet som utvecklaren erbjuds vid utveckling av MIDlets.

Applikationsprogrammeringsgränssnitten är grupperade i så kallade paket, vilka är bibliotek med innehåll vars funktion riktas till något specifikt område. I paketen finns upp till fyra olika sorters verktyg:

- Klasser (`class`). Detta är en samling av data och operationer utifrån vilka objekt kan instansieras. En klass kan också fungera som förälder till en ny klass vid arv.
- Gränssnitt (`interface`). Ett gränssnitt är en tom mall som klasser kan utnyttja. Det går inte att instansiera objekt av gränssnitt.
- Undantagsklasser (`Exception`). När ett fall som inte kan behandlas eller behöver speciell hantering uppstår kan ett undantag kastas. Exempel på ett undantag är division med noll.
- Felklasser (`Error`). En allvarligare variant av undantag är fel, som inte kan behandlas ens med speciell hantering. Exempel på ett fel är när minnet i en enhet tar slut.

[26]

4.3.1 CLDC

CLDC erbjuder väldigt begränsad funktionalitet. Av de fyra paket som finns tillgängliga utgör tre en liten delmängd av J2SEs applikationsprogrammeringsgränssnitt medan det sista är nytt och riktar sig till målgruppen med resurssnåla enheter. Paketen är:

- `java.lang`. Detta paket erbjuder funktionalitet som ligger nära programspråket: Här finns klasser som utökar egenskaperna för primitiva datatyper och verktyg vid exekvering. Klassen `Thread` och gränssnittet `Runnable` är verktyg som gör det möjligt att skapa parallellt exekverande trådar i en applikation, men vi belyser än en gång att det inte är något krav att enheterna stödjer detta. I paketet finns också undantagsklasser för hantering av indexering utanför arrayer och aritmetiska omöjligheter såsom division med noll. De få felklasser som erbjuds i detta paket är till för minnesbrist och omöjligheter för den virtuella maskinen att fortsätta exekvera.
- `java.util`. Paket som innehåller containerklasser för att lagra olika typer av instansierade objekt och andra användbara tillämpningar såsom slumptal, datum och tid.
- `java.io`. Paketet tillhandahåller funktionalitet för generella in- och utströmmar av data.
- `javax.microedition.io`. Innehåller ett flertal generella gränssnitt för nätverksuppkoppling. Klasser som skapas utifrån ett av dessa gränssnitt instansieras med hjälp av klassen `Connector` vars medlemsmetod `open` returnerar och upprättar ett generellt gränssnitt för nätverksuppkoppling. Som tidigare nämnts i avsnitt 4.1 implementerar dock inte CLDC någon nätverksuppkoppling, utan lämnar den

uppgiften till ovanliggande lager i javaplattformen (MIDP till exempel, som tidigare nämnts i avsnitt 4.1.2).

[26]

4.3.2 MIDP

Eftersom MIDP bygger på CLDC inkluderar dess applikationsprogrammeringsgränssnitt CLDC. Tre paket med funktionalitet för olika områden hos mobila terminaler har tillkommit:

- `javax.microedition.midlet`. Detta paket innehåller klassen `MIDlet` och ett undantag som kan kastas av denna. `MIDlet` är den klass som alla MIDlets huvudklasser ärver från för att kunna köras i de mobila terminalerna.
- `javax.microedition.lcdui`. Här finns klasser för att skapa grafiska användargränssnitt, behandla knapptryckningar samt tända och släcka punkter på skärmen. `TextBox` är en klass som representerar en textruta där text kan visas och matas in. `Canvas` är en klass avsedd att användas för att skapa spel. `Canvas` tillhandahåller metoder för att sköta händelser och knapptryckningar och specificerar en yta som går att rita på. Klassen `Graphics` möjliggör fri modifiering av enskilda punkter på skärmen. Upp till 24 bitars färgdjup är möjligt för de terminaler som stödjer det och ett antal färdiga geometriska figurer som cirklar och rektanglar går att rita ut.
- `javax.microedition.rms`. Klassen `RecordStore` och några gränssnitt och undantagsklasser för hantering av beständig data finns i detta paket. `RecordStore` är den klass som representerar beständig data och klarar av att spara, radera och ta reda på information om filer. Som antyddes i avsnitt 4.1.2 har klassen tillgång endast till sin beskärda del av terminalens beständiga minne.

[27]

MIDP skall, som tidigare nämnts i avsnitt 4.1.2, enligt specifikationen implementera åtminstone nätverksprotokollet HTTP. Detta åstadkoms genom att MIDP implementerar gränssnittet `HttpConnection` som ett tillägg i CLDCs paket `javax.microedition.io`. Metoden `open` i klassen `Connector` klarar, som tidigare nämnts, av att upprätta och returnera ett generellt gränssnitt för nätverksuppkoppling och därmed även `HttpConnection`. [9][27]

4.4 Framtid – MIDP Next Generation

Under tiden denna uppsats skapades släpptes en så kallad public review av CLDC 1.1 och MIDP 2.0. En hel del ny funktionalitet har lagts till och det finns nu mer ingående dokument som behandlar hur bland annat nedladdning (kapitel 5) och preverifiering (avsnitt 6.3) går till. En public review är endast en kandidat till den färdiga standarden och det kan hända att vissa delar ändrats när de båda standarderna släpps. Även om CLDC 1.1 och MIDP 2.0 släpps som färdiga standarder inom en relativt snar framtid kommer det att dröja innan mobila enheter som implementerar dem finns på marknaden. Det är först nu, några år efter att CLDC 1.0 och MIDP 1.0 släpptes, som antalet enheter som stödjer dessa standarder ökar och det lär dröja ett tag för tillverkarna att konstruera nya lösningar och hinna testa dem tillräckligt mycket innan CLDC 1.1 och MIDP 2.0 etableras på marknaden. [3][7]

Även om säkerheten i MIDP 1.0 och CLDC 1.0 är mycket god innebär sandlådemodellen en stor begränsning av vilka applikationer som är möjliga att skapa. Med MIDP 2.0 ska certifikat och signering av MIDlets införas. Om en användare kan lita på en applikation kan applikationen i sig ges större befogenheter. Alla implementationer av javaplattformen måste tillåta att MIDlets som användaren inte litar på kan köras. Detta kan låta konstigt men hela konceptet Java bygger mycket på att kunna tillåta sådana applikationer. Systemet med signering och certifikathantering är resurskrävande och det återstår att se om kommande enheter kommer att implementera stöd för detta. [3]

CLDC 1.1 ska stödja en hel del som CLDC 1.0 inte har stöd för: Flyttal, en större delmängd av J2SE samt utökad fel- och undantagshantering. Kravet på tillgängligt minne ökar dessutom från 160 kB till 192 kB. [3]

Under profilen MIDP har ett antal klasser lagts till. Det rör sig om funktionalitet för bland annat spel, media och certifikat. Ett stort antal företag och organisationer har varit med om att specificera MIDP 2.0, däribland Siemens. MIDP 2.0 har utökad funktionalitet för spelutveckling med tydliga drag från det utökade applikationsprogrammeringsgränssnittet i Siemens SL45i, se kapitel 7. För att ytterligare förstärka upplevelsen av en MIDlet har ett mediabibliotek tillkommit. Även om kravet endast är att kunna generera toner under en viss tidsrymd så är det meningen att stöd för MIDI skall implementeras. Detta är ett mycket bra sätt att representera musik i ett litet format. Nya och utökade kommunikationsmöjligheter står också på listan över nyheter. [7][37]

5 Nedladdningsförfaranden

För att standardisera nedladdningen av applikationer har specifikationen Over The Air User Initiated Provisioning (OTA) tagits fram. OTA specificerar hur en applikation kan och bör föras över från en server till den mobila terminalen. Vissa krav finns på nätverksprotokollen, enheterna och mjukvaran, men det mesta är rekommendationer. [25]

I kapitel 8, avsnitt 8.2 kompletterar vi detta kapitel med en analys av möjliga sätt att utnyttja nedladdningsförfarandet på ett för användare och seriösa utvecklare icke önskvärt sätt. Det visar sig att versionsuppgraderingar öppnar ett hål i säkerheten.

5.1 Upptäckt

Till att börja med behöver den mobila terminalen någon mekanism för att låta användaren hitta en MIDlet som denne önskar ladda ned. Detta kan åstadkommas med enhetens motsvarighet till persondatorernas webbläsare, WAP-läsaren. WAP står för Wireless Application Protocol och är en samling nätverksprotokoll som det är vanligt att mobila terminaler använder för att hämta och skicka data över Internet. [25]

OTA specificerar hur utvecklare bör distribuera sina applikationer för att vara kompatibla med WAP. Ett vanligt scenario är att den mobila terminalen och servern som applikationen finns på kommunicerar via en så kallad gateway. Detta eftersom det är vanligt att den mobila terminalen använder WAP och servern TCP/IP. De båda protokollen kan tolkas och översättas i en gateway. [25]

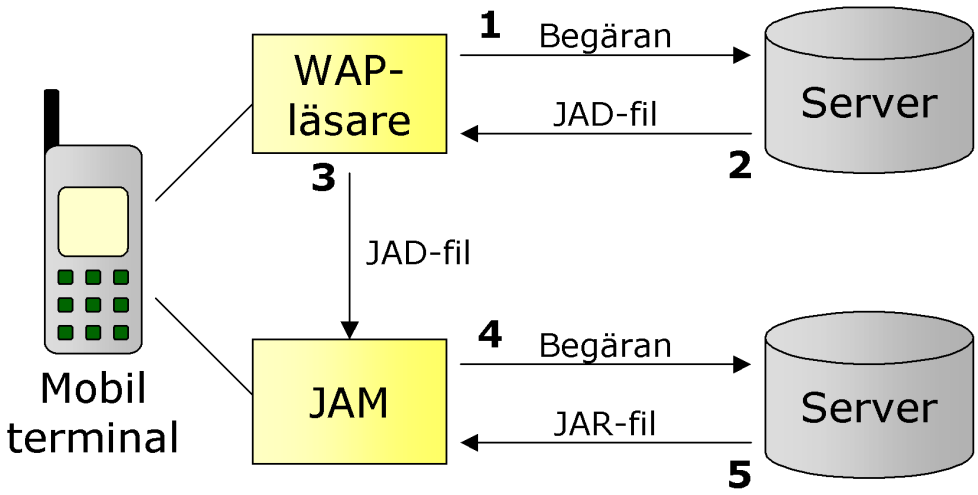
5.2 Nedladdning

Terminaler med WAP-läsare behöver stödja överföring av JAD- och JAR-filer. Dessutom krävs stöd för responsmeddelanden för att servern skall kunna rapporteras om hur installationen gick för att till exempel kunna föra statistik eller ta betalt. Om inget responsmeddelande kommer från klienten antar servern att installationen gick bra. Servern som applikationen laddas ned från måste stödja funktionaliteten i HTTP 1.1. [25]

En färdig MIDlet distribueras vanligtvis på en server som är åtkomlig via Internet. Tre filer är normalt inblandade: JAD-filen, JAR-filen samt en presentationsfil. Presentationsfilen kan

till exempel vara ett WML-dokument, det vill säga ett dokument som enhetens WAP-läsare kan tolka och presentera. [25]

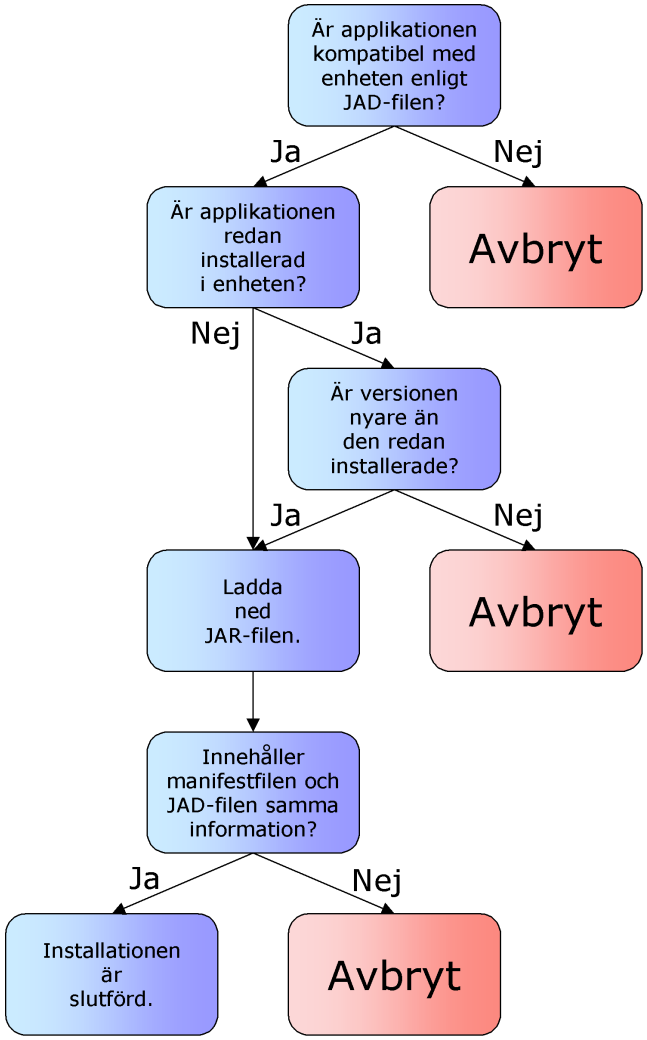
I Figur 5.1 är nedladdningsförfarandet illustrerat från det att WAP-läsaren följer en hyperlänk till en applikation, till det att applikationen laddats ned till terminalen. En användare ansluter sin terminal till Internet och använder dess WAP-läsare för att navigera till en domän där en utvecklares MIDlet finns angiven i form av en hyperlänk. Användaren väljer att ladda ned utvecklarens MIDlet varefter WAP-läsaren följer hyperlänken, vilken refererar till applikationens JAD-fil, se steg 1 i Figur 5.1. WAP-läsaren laddar ned JAD-filen till terminalen, se steg 2 i Figur 5.1, och låter sedan Java Application Manager (JAM) tolka innehållet i filen, se steg 3 i Figur 5.1. JAM är det program i terminalen som hanterar installationen och livscykeln hos en eller flera applikationer i en programsvit. [25]



Figur 5.1: Översikt över hur nedladdningen av en applikation går till

5.3 Installation

JAM börjar installationen genom att försäkra sig om att applikationen enligt JAD-filen kan fungera i terminalen och får plats i minnet. Om applikationen som beskrivs i JAD-filen redan finns installerad skall installationen hanteras som en potentiell uppgradering. Uppgradering sker endast om versionsnumret på den redan installerade applikationen är lägre än det som anges i JAD-filen. Vid uppgradering rekommenderas det att beständig data sparas. Efter dessa kontroller kan nedladdningen av JAR-filen påbörjas, se steg 4 och 5 i Figur 5.1. Om informationen i manifestfilen i JAR-filen inte överensstämmer med den som angivits i JAD-filen så kasseras JAR-filen. En översikt av det beskrivna flödet är illustrerat i Figur 5.2. [25]



Figur 5.2: Översiktligt flödesschema för hur installationen går till

I OTA står det att en versionsuppgradering kan öppna ett säkerhetshål. Detta i och med att versionsuppgraderingen är helt textbaserad. Innehållet i en JAD-fil och en manifestfil är, som tidigare nämnts i avsnitt 4.2.1, läsbar text. Kravet för att en versionsuppgradering ska ske är endast att den upptäckta applikationen har samma namn och att den har ett högre versionsnummer än den redan installerade applikationen. Inget krav ställs på att applikationen måste uppgraderas från samma server som den först laddades ned från. Därför är det fullt möjligt att uppgraderingen inte alls är en nyare version av den applikation som redan är installerad i enheten. Eftersom det är rekommenderat att beständig data sparas för den uppgraderade applikationen kan känslig beständig data exponeras för en falsk uppgradering. [25]

6 Systemnära delar i J2ME

I detta kapitel presenterar vi de grundläggande strukturer som finns i ledet mellan kompilatorn och den virtuella maskinen. När vi undersöker möjligheten till skadlig kod på denna nivå lämnar vi även den traditionella programmeringen. I vanliga fall är det meningen att källkod ska beskriva den färdiga applikationen och omvandlas till bytekod i klassfiler med en kompilator. Genom att modifiera klassfilerna efter kompileringen kan det hända att den virtuella maskinen konfronteras med instruktioner som den egentligen inte är ämnad att behandla. Bytekod är till skillnad från källkod inte lättläst utan består av kombinationer av ettor och nollor som tillsammans bildar symboliska instruktioner till den virtuella maskinen. Därför behövs speciella verktyg för att studera klassfiler. [16]

Detta kapitel ämnar ge en introduktion till hur den virtuella maskinen fungerar och hur klassfilerna som den exekverar är uppbyggda. Detta görs i avsnitt 6.1 och 6.2. Speciell uppmärksamhet ges åt attributet Stackmap som är specifikt för KVM och dess preverifiering som presenteras i avsnitt 6.3. Kunskapen används sedan för att utröna huruvida det går att lura den virtuella maskinen att exekvera kod som inte kan genereras av en giltig kompilator, se kapitel 8, avsnitt 8.3. Samtliga av de modifierade klassfiler som vi prövat att exekvera accepterades inte i testenheten.

6.1 Den virtuella maskinen KVM

Ordet virtuell kan översättas till tänkt. En virtuell maskin existerar inte fysiskt utan är endast ett program i en annan maskin, ett program som agerar som en tänkt maskin. Ett sådant program som tolkar och exekverar klassfiler kompilerade med en kompilator för programspråket Java är en virtuell javamaskin. För en applikation skriven i Java skall plattformen inte vara relevant. Det enda applikationen skall behöva beakta är vad den virtuella maskinen tillhandahåller, inte vad den fysiska maskinen och dess operativsystem tillhandahåller. Genom att tillverka olika virtuella maskiner för olika plattformar uppstår ett gemensamt gränssnitt mot applikationen. En och samma javaapplikation kan därmed köras på olika plattformar, så länge plattformen har en anpassad virtuell javamaskin. En utvecklare behöver alltså inte bry sig om vilken typ av maskin den färdiga applikationen ska exekveras på, utan kan utveckla sin applikation för den virtuella javamaskinen. [5][8][23]

Den virtuella maskinen har flera uppgifter. Den primära är att tolka innehållet i klassfiler och exekvera de instruktioner som finns angivna där i form av bytekod. En applikation som är kompilerad till maskinkod exekveras genom att processorn i den maskin applikationen exekverar ovanpå hämtar instruktioner från maskinkoden, tolkar dessa och ser till att de utförs. Det är på ett liknande sätt en javaapplikation exekverar ovanpå en virtuell maskin: Genom att dess innehåll tolkas och instruktionerna som finns där utförs. Skillnaden är att den virtuella maskinen är just virtuell. Den kan inte själv agera som en fullfjädrad fysisk maskin eftersom den själv endast är ett program som exekverar ovanpå den fysiska maskinen. Därför måste den virtuella maskinen översätta bytekoden i javaapplikationens klassfiler till maskinkod som processorn i den fysiska maskinen förstår. Andra uppgifter som den virtuella maskinen har är att hantera minne, upprätthålla säkerhet mot skadlig kod och hantera parallellt exekverande trådar inom den process som applikationen utgör i den fysiska maskinens system. [19]

Som antytts i avsnitt 2.3.2 existerar det inte bara en Java Virtual Machine (JVM) utan många. Anledningen till detta är att det finns flera olika användningsområden och plattformar som de anpassas till. Deras implementationer kan skilja sig åt en hel del men det mest grundläggande som alla JVM måste stödja finns specificerat i Java Virtual Machine Specification (JVMS). Avancerade virtuella maskiner innehåller mängder av funktioner och kan vara optimerade för olika ändamål. De måste dock kunna exekvera klassfiler skrivna för den virtuella maskin som Sun Microsystems [29] specificerat i JVMS för att få kalla sig javakompatibla. [23]

När J2ME togs fram ville upphovsmännen att även de virtuella maskinerna i resurssnåla enheter skulle följa JVMS så långt som möjligt. Detta för att bibehålla portabiliteten mellan olika plattformar och kompatibiliteten mellan olika delar av javaplattformen. För KVM, den virtuella maskin som enheter i MIDPs målgrupp implementerar, har vissa funktioner som kräver alltför mycket minne eller processorkraft tagits bort. I stort sett är KVM lik JVM. Den största skillnaden är saknaden av stöd för flyttal och en stor mängd inbyggd funktionalitet samt att en yttre verifiering av klassfiler kallad preverifiering måste tillämpas. Verifiering innebär som tidigare nämnts i avsnitt 3.2.3 att klassfiler kontrolleras mot fel och ogiltigt innehåll. KVM har en något annorlunda verifiering än övriga virtuella maskiner i och med preverifieringen som måste göras innan applikationen förs över till en enhet. Mer om verifiering och preverifiering finns att läsa i avsnitt 6.3. [8]

6.2 Javas klassfiler

6.2.1 Javas bytekod

När vi talar om bytekod menar vi en ström av bytes som utgör instruktioner för exekveringsförloppet i ett Javaprogram. Varje instruktion i ett program motsvaras av ett tal mellan 0 och 255 (00 och FF hexadecimalt) och lagras i en byte. En vanlig dator skulle inte kunna exekvera programmet om det inte fanns en virtuell maskin som översatte bytekoden till maskinkod. Bytekod är alltså bytes vars värde pekar ut en speciell instruktion. Ett exempel på en bytekod är talet 0 som tolkas som operationen "no operation". Ett mer avancerat exempel följer nedan där en enkel metod i programspråket Java omvandlas till bytekod: [23]

```
void spin()
{
  int i;
  for (i = 0; i < 100; i++);
}
```

Metoden `spin` definierar en heltalsvariabel och låter dess värde öka från 0 till 99. Funktionskroppen för `spin` motsvaras av följande serie hexadecimala bytekoder:

```
{03, 3C, A7, 00, 08, 84, 01, 01, 1B, 10, 64, A1, 00, 05, B1}
```

Alla koderna är inte rena instruktioner. Inbakat finns data och index som fungerar som pekare till andra instruktioner eller index till den lista av konstanter som finns i klassfilen, se avsnitt 6.2.2. Dessa koder är argument till instruktioner. För att kunna skilja på instruktioner och argument har man satt namn på instruktionerna och ställt upp strukturer för dem i JVMS. [23]

Tabell 6.1 utgörs av ovan nämnda koder översatta till de namn som finns angivna i JVMS tillsammans med kommentarer. Kolumnen Index refererar till den position som instruktionen har i bytekoden, kolumnen Kod visar det hexadecimala värdet, kolumnen Namn visar det namn som givits åt koden och kolumnen Operander visar eventuella argument till instruktionen. Lägg märke till att många instruktioner handlar om att manipulera stacken.

Index	Kod	Namn	Operand(er)	Kommentar
0	03	iconst_0		Lägg heltalsvärdet noll på stacken
1	3C	istore_1		Lagra värdet noll i lokal variabel ett (i)
2	A7	goto	8	Första iterationen, hoppa över inkrementeringen
5	84	iinc	1, 1	Öka lokal variabel ett med ett (i++)
8	1B	iload_1		Lägg lokal variabel 1 på stacken
9	10	bipush	100	Lägg värdet 100 på stacken
11	A1	if_icmplt	5	Hoppa till instruktion vid index 5 om i < 100
14	B1	return		Returnera när funktionen är färdig

Tabell 6.1: Exempel på bytekoder i Java och deras innebörd

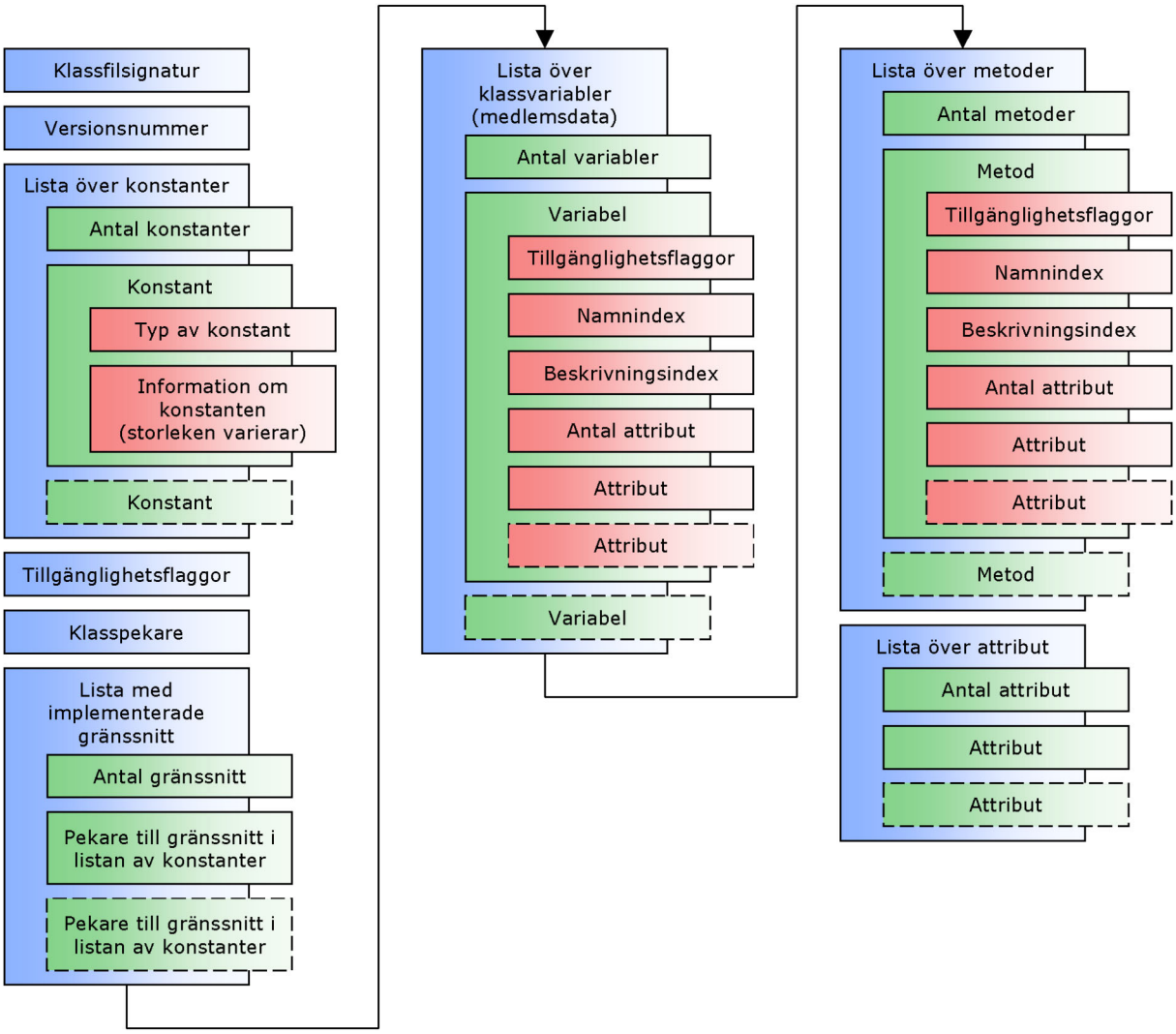
För att läsa och modifiera bytekod har vi använt oss av en så kallad hexredigerare som heter Hackman [31]. Hackman visar varje byte i en fil som ett hexadecimalt värde, och följaktligen också varje bytekod i en klassfil som ett hexadecimalt värde. Detta program är bra för att ändra på enskilda byte, men vi ansåg det inte vara tillräckligt bra för att kunna tolka innehållet i en klassfil. Därför har vi utvecklat ett verktyg som vi valt att kalla ClassToXML, se bilaga B, som strukturerar upp innehållet i en klassfil och presenterar den som ett strukturerat XML-dokument. I XML-dokumentet är det sedan enkelt att hitta en enskild byte, se vad den har för innebörd och få ett index till vilken position den finns på i klassfilen. Därefter används Hackman för att eventuellt modifiera klassfilen.

ClassToXML är ett bra hjälpmedel för att studera något som annars skulle bestå av en lång serie hexadecimala tal. I bilaga B analyseras en enkel klassfil för att illustrera hur ClassToXML fungerar och kan vara till nytta.

6.2.2 Javas klassfilsstruktur

En klassfil genereras för varje klass skriven i programspråket Java. Klassfilen innehåller all nödvändig information som den virtuella maskinen behöver för att kunna skapa objekt av klassen och exekvera den kod som klassen innehåller. Alla implementationer av virtuella maskiner för Java måste stödja det filformat som specificerats i JVMS. [23]

En klassfil byggs upp av en ström av bytes. Om ett värde representeras av fyra bytes, exempelvis data av typen `int`, bryts värdet ner i fyra kvantiteter om vardera en byte och lagras i ordningen ”big endian” där den högsta byten kommer först. Då vi tittar på innehållet i en klassfil och tolkar koden hittar vi en struktur som specificeras i JVMS. Figur 6.1 ger en översikt av strukturen. Två på varandra följande poster med samma namn där den andra har streckad kantlinje anger att det kan finnas flera förekomster av denna post. [23]



Figur 6.1: Klassfilsstruktur

Klassfilens struktur byggs upp enligt posterna i Figur 6.1. Nedan följer en förklarande text till varje post i figuren. När vi i texten skriver ”klassen” så syftar vi på den klass som klassfilen beskriver. En närmare förklaring på vad ett attribut är ges efteråt, i avsnitt 6.2.3.

- Klassfilen inleds med klassfilssignaturen som är ett fyra bytes långt tal innehållande det hexadecimala värdet CAFEBABE. Detta tal fungerar som en identifierare för klassfiler. Filer som inte inleds med detta tal tolkas inte som klassfiler av den virtuella maskinen.
- Det finns två versionsnummer i en klassfil. Dessa nummer visar vilken version av kompilatorn som skapat klassfilen.
- Listan över konstanter är en lista över all konstant data som sparas i klassfilen. Här har ordet konstant inte den betydelse som annars är vanlig inom programspråk.

Konstanterna här är mer tänkta att fungera som behållare av information för förekommande namn på metoder, klasser och andra namngivna entiteter.

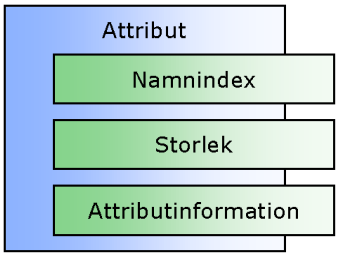
- Med tillgänglighetsflaggor för klassen menas hur klassen är synlig för andra klasser. Här anges om klassen är publik (`public`) eller privat (`private`), slutgiltig (`final`) eller abstrakt (`abstract`) och om den är en basklass (`super`) eller ett gränssnitt (`interface`).
- Posten Klasspekare inkluderar två fält som båda är index till varsin position i listan över konstanter: Det första fältet pekar ut klassen och det andra fältet pekar ut vilken klass klassen ärver från.
- Listan med implementerade gränssnitt är som namnet antyder en lista över vilka gränssnitt (`interface`) klassen implementerar.
- Listan över klassvariabler avser enbart sådana variabler som är deklarerade i klassen och inte variabler deklarerade i metoder. Varje variabel har fälten Tillgänglighetsflaggor, Namnindex, Beskrivningsindex, Antal attribut och Attribut. Tillgänglighetsflaggor anger vilken åtkomst andra klasser har till variabeln, Namnindex pekar ut variabelns namn i listan över konstanter, Beskrivningsindex anger vilken datatyp variabeln har och Attribut är ett eller flera fält för eventuell ytterligare information om variabeln. Attribut behandlas i avsnitt 6.2.3.
- Listan över metoder ser ut på samma sätt som listan över klassvariabler. Varje metod i listan över metoder har ett attribut som benämns Code. Här döljer sig den bytekod som utgör de instruktioner som metoden vid anrop skall utföra. I Code kan underattributet Stackmap förekomma. Attribut behandlas i avsnitt 6.2.3.
- Klassfilen avslutas med ett antal attribut innehållande eventuell ytterligare information om klassen. Typiskt är att det enda attributet här innehåller ett index till listan av konstanter där källkodsfilens namn är angivet. Attribut behandlas i avsnitt 6.2.3.

[23]

Det skiljer inte mycket mellan en klassfil för JVM och en klassfil för KVM. En klassfil för KVM är en giltig klassfil för JVM men inte tvärt om. Det finns dock en skillnad som är betydande. För att en klassfil skall kunna exekveras på en enhet med stöd för MIDP måste klassen genomgå en viktig process: Preverifiering, se avsnitt 6.3. Bytekoden förändras men behåller samma semantiska innebörd. Attributet Stackmap kommer att läggas till som ett underattribut till attributet Code i de moduler i klassfilen där det behövs. En klassfil för JVM och en klassfil för KVM skiljer sig inte förrän klassfilen genomgått preverifieringen. [8][23]

6.2.3 Attributet Stackmap

Ett attribut är ett fält för eventuell ytterligare information om den post i klassfilen som attributet tillhör. En illustration av hur ett attribut är uppbyggt ges i Figur 6.2.



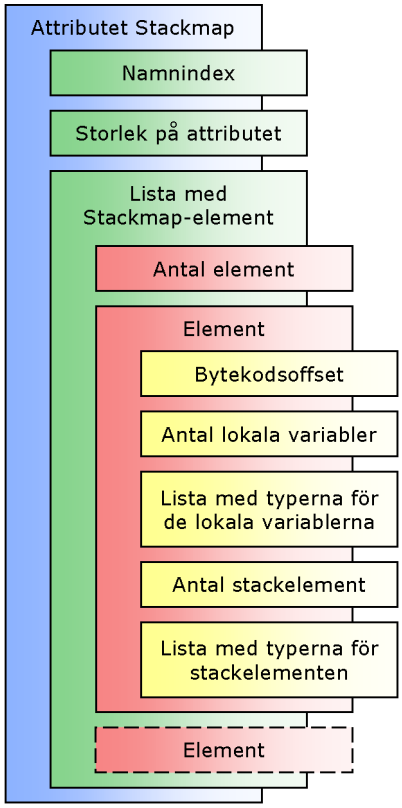
Figur 6.2: Struktur för attribut

Fälten i ett attribut ser ut som följer:

- Namnindex är index till listan över konstanter där attributets namn är angivet. Exempelvis är strängen ”Code” angiven för attributet Code. Detta index talar om för KVM vilket attribut det handlar om.
- Fältet Storlek anger storleken på fältet Attributinformation.
- Det är fältet Attributinformation som innehåller själva informationen som attributet lagrar. Attributinformation är av godtycklig storlek och kan själv innehålla ett godtyckligt antal attribut. Som ett exempel på detta kan nämnas att attributet Stackmap ligger i fältet Attributinformation i attributet Code.

[23]

Attribut är en viktig del i strukturen som bygger upp en klassfil. De fungerar som behållare av information och ofta handlar det om attributet Code som innehåller den faktiska bytekoden. Systemet med attribut möjliggör också att mer information än vad som behövs kan lagras i en klassfil. Detta faktum utnyttjas av klassfiler för KVM. Attributet Stackmap är till för den verifieringsprocess som körs internt i enheten. Syftet är att minska kraven på hårdvaran. När Java körs på en vanlig dator finns det relativt stora resurser för att utföra en analys av programflödet via iterativa algoritmer. Detta kräver resurser som ofta inte finns tillgängliga i en mobil terminal. Genom att lägga in attributet Stackmap kan man alltså underlätta verifieringen i enheten genom att göra processen linjär, det vill säga en resurssnål analys utan hopp och upprepningar. På detta vis görs javaplattformen tillgänglig för fler och mindre kostsamma enheter. En illustration av Stackmap finns i Figur 6.3. [8][23][36]



Figur 6.3: Struktur för attributet Stackmap

Fältet Lista med Stackmap-element är fältet Attributinformation i den generella strukturen för attribut i Figur 6.2. Stackmap inleds som andra attribut med ett index till listan över konstanter vilket pekar på ett fält vars innehåll är strängen ”Stackmap”. Detta talar alltså om för verifieringen i KVM att det handlar om attributet Stackmap. Attributet Stackmap består i huvudsak av ett antal element och efter att deras antal angivits följer elementen ett efter ett. Elementen innehåller avståndsmarkörer i bytekoden, antalet lokala variabler och deras datatyper samt antalet element på stacken och deras datatyper. [23]

Attributet hjälper verifieringen genom att fungera som en referens för två saker: Vilka datatyper lokala variabler har och deras antal samt vilka datatyper operanderna på stacken har och deras antal. Vissa bytekoder i attributet Code tilldelas attributet Stackmap för att möjliggöra en linjär verifiering. Även om det är fullt möjligt att tilldela varje bytekod i Code attributet Stackmap så sker det inte eftersom det inte är nödvändigt. Stackmap skapas endast när KVMs verifiering stöter på instruktioner som kräver det. Exakt vilka bytekoder som tilldelas attributet Stackmap anser vi inte nödvändigt att presentera i uppsatsen. [23][36]

6.3 Preverifiering

Liksom JVM måste KVM kunna välja att inte acceptera ogiltiga klassfiler. Beroende på att verifieringen av klassfilen kräver både kraft och lagringsutrymme har man valt en något annorlunda lösning på problemet. Effektiv verifiering uppnås genom införandet av preverifiering som ligger utanför den mobila terminalen. Detta möjliggör en enklare typ av verifiering i den mobila terminalen. Med andra ord måste en klassfil som skall köras på en mobil terminal genomgå en preverifiering externt för att kunna genomgå en verifiering internt och sedan exekveras i den mobila terminalen. [8]

6.3.1 Preverifiering externt

Preverifiering sker ofta som en del av utvecklingsprocessen. Verktyget för preverifiering följer med Wireless Toolkit [30]. En av dess viktigare uppgifter är att ta bort alla bytekoder av typen ”jump to subroutine” och ”return from subroutine” från klassfilen, alltså anrop till och returer från metoder. Antingen ersätts de med semantiskt ekvivalenta bytekoder, eller så flyttas koden för metoderna till den plats där metoderna skulle ha anropats. Denna modifikation görs för att undvika modifiering av programflödet vid exekvering eftersom det anses vara alltför resurskrävande. [8]

Attributet Stackmap skapas vid behov vid preverifieringen och bidrar till en snabb och effektiv verifiering internt i enheten på det vis som beskrevs i avsnitt 6.2.3. Attributet ignoreras av JVM och gör således klassfilen uppåtkompatibel med J2SE. Nackdelen med preverifieringen är att storleken på klassfilen ökar med ungefär fem procent. [8]

I bilaga B använder vi vårt egenutvecklade verktyg ClassToXML för att strukturera upp en klassfil före och efter preverifiering. Med de XML-dokument som ClassToXML genererar är det enkelt att se de förändringar som sker vid preverifieringen.

6.3.2 Intern verifiering

Då en klassfil modifierats av verktyget för preverifiering kan verifiering ske i den mobila terminalen. Detta innebär först och främst allokering av minne för att kunna lagra all information om alla variabler och objekt som förekommer. Hur mycket minne som skall allokeras avgörs av attributet Stackmap. Efter ytterligare initiering går verifieraren genom alla instruktioner och kontrollerar att de är giltiga. Vissa instruktioner har som tidigare nämnts tilldelats en post i attributet Stackmap. Skulle denna post inte finnas där den behövs eller om innehållet i posten inte är korrekt kommer verifieringen i den mobila terminalen att rapportera ett fel och applikationen kommer inte att köras. [8]

7 Utökade applikationsprogrammeringsgränssnitt

Med CLDC och MIDP ges möjligheten att utveckla applikationer. Specifikationerna definierar funktionalitet i form av ett antal klasser som är tänkta att ge utvecklaren de verktyg som behövs. Antalet klasser är minimalt och därför har somliga tillverkare av enheter utökat CLDC och MIDP med egen terminalspecifik funktionalitet. Syftet är dels att underlätta utvecklingen av applikationer till just den specifika enheten, men främst för att möjliggöra skapandet av mer avancerade applikationer som utnyttjar terminalens egna funktioner. Det har visat sig att det oftast handlar om klasser som tillhandahåller funktionalitet för spelprogrammering eller utökade möjligheter för kommunikation.

Anledningen till att MIDP inte innehåller sådana funktioner från första början är främst på grund av utrymmesskäl. Vi tror även att det kan vara väldigt terminalspecifikt hur ett telefonsamtal upprättas och därmed svårt att göra en generell lösning på detta.

I detta kapitel har vi valt att koncentrera oss på en specifik mobil terminal och presentera dess utökade applikationsprogrammeringsgränssnitt. Siemens SL45i är en mobil terminal med stöd för J2ME genom CLDC och MIDP. Den har terminalspecifik funktionalitet som är åtkomlig via ett utökat applikationsprogrammeringsgränssnitt.

7.1 Egenskaper hos Siemens SL45i

Siemens SL45i är en bra representant för mobila terminaler med stöd för Java. Den implementerar CLDC och MIDP och har ett utökat applikationsprogrammeringsgränssnitt. Terminalen kan anslutas till en persondator för att en användare skall kunna placera javaapplikationer i den, men erbjuder även möjligheten att ladda ned applikationer från Internet på det vis som beskrevs i kapitel 5. Terminalen är också försedd med ett minneskort på 32 MB, där nedladdande applikationer kan sparas. Figur 1.2 i kapitel 1 visar hur Siemens SL45i ser ut. [10][34][35]

7.2 Utökningar i Siemens SL45i

Vårt val att gå närmare in på en specifik mobil terminal grundar sig på de fakta som pekar på att J2ME är säker så länge sandlådemodellen bibehålls. Studerar vi de klasser som erbjuds utöver MIDP och CLDC ser vi att Siemens på grund av säkerhetsaspekter valt att begränsa

deras betydelse, och att kraftfulla funktioner som fri åtkomst till flertalet av telefonens inbyggda program har uteslutits. Siemens vill inte ta några risker och det är förståeligt.

Det utökade applikationsprogrammeringsgränssnittet i Siemens SL45i har stöd för spelutveckling och ljud, möjliggör sändning av korta textmeddelanden genom SMS (Short Message Service), tillåter upprättande av telefonsamtal samt tillåter begränsad användning av telefonboken. Nedan följer en kort presentation av de tre paket som utökar funktionaliteten i CLDC och MIDP:

- Paketet `com.siemens.mp.io` innehåller utökade möjligheter för in- och utmatning av data. En ny filhanteringsklass `File` som är ett alternativ till RMS finns här. `File` möjliggör mappstrukturer, men tillåter inte en applikation att byta mapp till nivåer högre än den nivå som applikationen finns i. Klassen `Connection` möjliggör sändning av data till andra enheter och gränssnittet `ConnectionListener` möjliggör mottagning av data från andra enheter.
- I paketet `com.siemens.mp.gsm` finns telefonifunktioner. Klassen `Call` kan upprätta telefonsamtal till ett angivet nummer, och om en javaapplikation lyckas göra detta så avslutas den och låter terminalen hantera telefonsamtalet. Klassen `SMS` kan sända SMS och klassen `PhoneBook` kommer åt missade samtal.
- Paketet `com.siemens.mp.game` tillhandahåller ett större antal klasser för att underlätta utveckling av spel. Det handlar om lättare hantering av grafik och ljud. Möjligheter att sätta på den inbyggda vibratorn och skärmens bakgrundsbelysning finns också.

[37]

8 Analys av möjligheten att utveckla och sprida skadlig kod

I detta kapitel analyserar vi möjligheten att utveckla och sprida skadlig kod. Kapitlet behandlar de områden som vi presenterat tidigare i uppsatsen. I avsnitt 8.1 analyserar vi möjligheten att utveckla skadlig kod med den funktionalitet som finns i CLDC och MIDP. Analysen är baserad på den presentation vi gjort i kapitel 4 och vi utreder de möjligheter som finns för en utvecklare att använda giltig kod, det vill säga kod som godkänns av en kompilator, för att skapa skadlig kod. Avsnitt 8.2 innehåller en analys av olika sätt att sprida skadlig kod. Analysen är baserad på presentationen av nedladdningsförfarandet i kapitel 5 och finns här eftersom skadlig kod inte är skadlig förrän den hamnat i en användares enhet. I avsnitt 8.3 analyseras möjligheten att utveckla skadlig kod genom att utnyttja de systemnära delar av J2ME som beskrivits i kapitel 6. Här tar vi upp möjligheterna att använda ogiltig kod, det vill säga kod som egentligen inte skall passera vare sig en kompilator eller en verifiering i en virtuell maskin, för att skapa skadlig kod. Slutligen analyserar vi i avsnitt 8.4 vilka möjligheter utökningarna i Siemens SL45i ger för att utveckla skadlig kod. Analysen baseras på kapitel 7 och utförs för att undersöka om utökningar av MIDP öppnar luckor i säkerheten.

8.1 Möjligheten att utveckla skadlig kod med funktionalitet i MIDP

Vår uppsats utreder möjligheterna att utveckla skadlig kod för mobila enheter. Säkerheten i javaplattformen är bra och de delar som kan orsaka säkerhetshål är som tidigare nämnts i kapitel 3 eliminerade i J2ME. Men eftersom J2ME är en relativt ny standard som tagits fram snabbt kan det hända att vissa områden förbisetts eller helt sonika glömts bort. Därför är det intressant att se till funktionaliteten i CLDC och MIDP och utreda om det finns möjlighet att genom den förstöra för eller irritera användaren eller rent av utvecklaren.

8.1.1 Att utveckla skadlig kod genom implementationsfel

Specifikationen av CLDC och MIDP lämnar inga uppenbara hål i säkerheten. Att en tillverkare av en mobil terminal kan avvika från specifikationen och hitta på egna lösningar ser vi inte som något hål i säkerheten för CLDC och MIDP. Om en tillverkare går från specifikationen litar vi på att denne kan sin sak och inte låter sin alternativa lösning öppna säkerhetsluckor. Skulle en säkerhetslucka mot förmodan öppnas så är den terminalspecifik och faller därför utanför uppsatsens avgränsning. Att leta efter implementationsfel i

standardbiblioteken genom att testa alla möjliga kombinationer av CLDCs och MIDPs möjligheter anser vi vara en alltför tidskrävande process. Därför undersöker vi inte om sådana förekommer.

8.1.2 Att utveckla skadlig kod med Record Management System

Enhetens filsystem Record Management System (RMS) är uppdelat så att varje programsvit har sin beskärda isolerade del av det. Detta innebär att en illasinnad utvecklare endast kan förvanska, radera eller läsa känslig information från filer som dennes egen applikation skapat. Om det går att komma åt filer utanför programsviten öppnas ett gigantiskt säkerhetshål. En intressant iakttagelse är att restriktionen med åtkomst endast till programsvitens RMS-filer saknas i emulatorerna som medföljer Sun Microsystems Wireless Toolkit [30]. Det är fullt möjligt för en programsvit att komma åt RMS-filer från en annan programsvit. Frågan är om denna restriktion saknas i någon verklig enhet. I avsnitt 9.1 testar vi detta på en mobil terminal av märket Siemens SL45i.

8.1.3 Att utveckla skadlig kod med MIDPs HTTP-uppkoppling

MIDP kan ansluta enheten till Internet med en HTTP-uppkoppling. Detta innebär att enheten har kontakt med någon annan enhet eller person. En tanke som uppstår är att enheten då kanske kan kontrolleras av någon annan är huvudanvändaren. Denna möjlighet finns dock bara om programmet i enheten själv tolkar vilka kommandon den illasinnade personen ber den utföra och utför dem. Därför måste en applikation ha programmerats för just det ändamålet. Om en användare har en sådan applikation i sin enhet kan den illasinnade utvecklaren över Internet radera programsvitens filer eller ändra skärmens innehåll för att irritera användaren. Frågan är vari nyttan ligger att styra enheten över en HTTP-uppkoppling istället för att bygga in störmoment i applikationen från början när det ändå är den illasinnade personen som måste skapa den.

HTTP-uppkopplingen möjliggör också för enheten att skicka känslig information, kanske även utan att användaren vet om det. Om det går att lura användaren att trycka sin PIN-kod eller sitt bankkortsnummer är det fullt möjligt att bakom ryggen på denne sända informationen till en illasinnad persons HTTP-server. Ett exempel kan vara en trovärdig applikation som utger sig för att ta reda på saldot på användarens bankkonto. Detta kan vid en första anblick tyckas vara en användbar applikation. När applikationen frågar användaren om dennes bankkortsnummer är det inte otänkbart att användaren skriver in det i tron att det kommer användas för att ta reda på saldot. I terminalens skärm står sedan att en uppkoppling

håller på att ske och användaren är övertygad om att enheten ansluter till bankens datorer. I själva verket har bankkortsnumret sänts till den illasinnade upphovsmannens HTTP-server. Denna form av skadlig kod är inte unik för applikationer utvecklade för MIDP. Dock visar det på en funktionalitet i en mobil terminal som skulle kunna verka skadligt för användaren.

8.1.4 Att utveckla skadlig kod med hjälp av det grafiska användargränssnittet

Ett försök att lura eller förvilla användaren kan göras genom att simulera enhetens grafiska gränssnitt. MIDPs applikationsprogrammeringsgränssnitt tillhandahåller funktionalitet som gör det möjligt att återskapa en miljö som är grafiskt identisk med någon del av enhetens grafiska gränssnitt. Exempelvis kan det gå att få användaren att mata in en kod genom att en situation där enheten normalt frågar efter en kod simuleras. På så vis kan applikationen få reda på koden och sända den till en illasinnad persons server vid tillfälle.

Ett grafiskt användargränssnitt kan också utformas så att enheten enligt användaren beter sig konstigt. Alternativet ”ja” kan motsvaras av ”nej” och vice versa eller också kan en följd av meningslösa frågor med dessa två svarsalternativ konfrontera användaren. Om en följd av flera sådana frågor börjar irritera användaren kan det hända att denne tröttnar och börjar trycka ”ja” oavbrutet för att bli av med de eviga frågorna som verkar sakna relevans. Om applikationen bland dessa frågor ger enheten en instruktion som det begärs bekräftelse på från användaren, till exempel bekräftning av uppkoppling eller sändning av data, kan det hända att användaren bekräftar något som egentligen inte skulle ha bekräftats.

Att simulera det grafiska användargränssnittet får användaren att tro att enheten beter sig konstigt, inte applikationen. Men om användaren startat en applikation har denne ingen anledning att tro att det är enheten som beter sig konstigt. Därför bör applikationen börja med att lura användaren att tro att applikationen inte körs, utan att det är enheten verkliga gränssnitt som visas på skärmen. Detta kan åstadkommas genom att applikationen börjar med att visa ett felmeddelande om att den inte kunde köras. Om användaren tror på detta är det sannolikt att denne även tror att det är enhetens verkliga gränssnitt som visas.

I avsnitt 9.2 kombinerar vi möjligheten att sända data över HTTP med möjligheten att simulera det grafiska användargränssnittet genom att skapa en enkel applikation som ber användaren om dennes PIN-kod och sedan sänder den till en server.

8.1.5 Att utveckla skadlig kod med en applikation utan innehåll

En intressant tanke vi fick medan vi testade funktionalitet i MIDP var hur en enhet beter sig när den skall ladda en applikation som är en giltig MIDlet, men som inte innehåller någonting

i övrigt. En tom MIDlet har ingen information att presentera, inga grafiska objekt att rita ut och inget sätt att kommunicera med användaren. I avsnitt 9.3 utför vi denna test.

8.1.6 Att utveckla skadlig kod med överanvändning av resurskrävande funktionalitet

Många av de mobila enheterna har och kommer att ha stöd för parallell exekvering genom användande av trådar, vilket är en funktion som erbjuds i CLDC. Ökad parallellitet ger ökad belastning på enheten. Enhetens system kan belastas genom att starta massor av parallellt exekverande trådar, öppna massor av filer samt köra beräkningsintensiva loopar där nya objekt ideligen instansieras. Det är möjligt att en enhet börjar uppträda på ett icke önskvärt sätt vid en sådan belastning. I avsnitt 9.4 testar vi hur Siemens SL45i beter sig vid ett antal olika former av belastning.

8.1.7 Förslag på tester av skadlig kod med funktionalitet i CLDC och MIDP

I kapitel 9 kommer följande tester att utföras baserad på analysen i detta avsnitt:

- Avsnitt 9.1: Åtkomst till andra programsviters RMS.
- Avsnitt 9.2: Lura en användare med grafiskt gränssnitt och HTTP-uppkoppling.
- Avsnitt 9.3: Applikation utan innehåll.
- Avsnitt 9.4: Belastning av enheten.

8.2 Möjligheten att sprida skadlig kod med nedladdningsförfarandet

Vilka möjligheter finns att utnyttja nedladdningsförfarandet för att få en användare att ladda ned en applikation med icke önskvärt innehåll? Det verkar finnas delar av nedladdningsförfarandet som inte är helt bra och detta anser vi vara värt att titta närmare på.

8.2.1 Mellanhand i nedladdningen

En användare ansluten till ett seriöst företags server med seriösa applikationer förväntar sig att kunna ladda ned dessa applikationer. När nedladdningen påbörjas finns dock möjligheten att en illasinnad datorkunnig person avlyssnar överföringen och modifierar den. Exempelvis skulle det kunna tänkas att personen själv tar emot den seriösa applikationen och låter användaren ladda ned en annan förmodligen icke önskvärd applikation. Både företaget och applikationen tror att överföringen gått bra, trots att så inte är fallet. Detta är ett exempel på en mellanhand i nedladdningen och kommer inte tas upp eller testas i uppsatsen då det faller utanför dess avgränsning.

8.2.2 Lura en användare att ladda ned icke önskvärda applikationer

För att en applikation över huvud taget ska hamna i en användares enhet måste användaren självmant välja att ladda ned den. För att en skadlig applikation ska hamna i en intet ont anande användares enhet måste med andra ord applikationen verka lockande för att bli nedladdad.

Falsk reklam kan vara ett bra sätt att göra sin applikation åtråvärd. Genom att utge applikationen för att göra något som en användare kan tänkas vara intresserad av kan illasinnade utvecklare locka användare att ladda ned skadliga applikationer. Ett lockande namn som exempelvis ”FreeSMS” (Gratis SMS) kan också tänkas få en användare att ladda ned en icke önskvärd applikation.

Med de nyaste terminalerna är det möjligt att skicka SMS med hyperlänkar i. Den användare som får ett SMS med en hyperlänk till en lockande applikation är då bara en knapptryckning från att ladda ned applikationen. Att göra det bekvämt för användaren att komma åt en applikation tror vi kan bidra en hel del till dennes beslut om att ladda ned applikationen eller ej.

Vi tror inte att en användare behöver vara särskilt lättlurad eller ovan vid nya teknologier för att bli lurad att använda applikationer som inte är önskvärda ur säkerhetssynpunkt. Att få en användare att ladda ned en applikation är för en illasinnad utvecklare minst lika viktigt som att lyckas skapa en skadlig applikation. Även om utvecklaren lyckas skapa en skadlig applikation så är den inte särskilt skadlig om ingen använder den.

8.2.3 Injicera en skadlig MIDlet i en redan installerad programsvit

Om det går att injicera en skadlig MIDlet i en redan installerad programsvit så innebär det en stor brist i säkerheten. Med att injicera en MIDlet i en redan installerad programsvit menar vi att lägga till en klass som ärver klassen `MIDlet` i en JAR-fil som redan finns i enheten. Eftersom en programsvit är en sluten entitet och alla MIDlets som ingår i den installeras och tas bort samtidigt går det dock inte att realisera med nedladdningsförfarandet. Kvarstår gör möjligheten att använda en anslutning till en persondator och modifiera JAR-filen den vägen. Eftersom vi testat detta och funnit att det går redovisar vi testet i avsnitt 9.6. Vi anser det dock inte öppna någon säkerhetslucka eftersom användaren själv måste ansluta sin enhet till persondatorn och injicera en MIDlet i en befintlig JAR-fil.

8.2.4 Injicera en skadlig MIDlet genom versionsuppgradering

Som nämnts i avsnitt 5.3 kan versionsuppgraderingen öppna ett säkerhetshål eftersom den är helt textbaserad och endast använder applikationens namn och versionsnummer. Ett seriöst SMS med en hyperlänk till senaste uppgraderingen av en redan installerad applikation kan kanske i själva verket komma från en illasinnad utvecklare. Om versionsuppgraderingen lyckas och programsvitens filer i RMS sparats kan den nya skadliga applikationen radera filerna eller skicka deras innehåll till utvecklarens server. På så vis kan känslig data raderas eller exponeras. Detta är givetvis inte bra, varför vi finner det intressant att testa. Testet är dokumenterat i avsnitt 9.7.

8.2.5 Förslag på tester av skadlig kod med nedladdningsförfarandet

I kapitel 9 kommer följande tester att utföras baserad på analysen i detta avsnitt:

- Avsnitt 9.6: Injicera en skadlig MIDlet i en redan installerad programsvit.
- Avsnitt 9.7: Versionsuppgradering till ett program med skadlig kod.

8.3 Möjligheten att utveckla skadlig kod med systemnära delar av J2ME

Eftersom klassfilsstrukturen modifierats något för J2ME, de virtuella maskinerna anpassats för resurssnåla enheter och verifieringen delats upp är det intressant att undersöka om dessa lösningar öppnar hål i säkerheten. Även om kompilatorerna inte kan producera skadlig kod kan det hända att en modifiering av klassfilernas bytekod efter såväl kompilering som preverifiering kan innebära att skadlig kod blir möjlig att utveckla.

Vi menar att javaplattformen och med den också J2ME har visat sig vara säker, eftersom den funnits så pass länge och allvarliga fel i specifikationen borde ha visat sig. Det är på den systemnära nivån brister borde uppstå om varje enskild tillverkare av mobila enheter inte följt de angivna specifikationerna i JVMS till punkt och pricka.

8.3.1 Att utveckla skadlig kod genom att modifiera den virtuella maskinen

En tanke är att ändra på KVM i en mobil terminal så den exekverar applikationer på ett icke önskvärt sätt eller exponerar otillåtna minnesområden för såväl läsning som skrivning för en utvecklare. Denna tanke förkastade vi dock ganska snart. Det kan gå att modifiera en virtuell maskin så den agerar skadligt, men vem skall använda den? Varje terminal levereras ju med en egen kopia av KVM och om en illasinnad utvecklare förändrar KVM i en av dessa terminaler så förändras givetvis inte KVM i de andra terminalerna.

En terminaltillverkare kan däremot själv ha gjort en miss vid implementationen av KVM som öppnar möjligheter för illasinnade utvecklare att skapa skadlig kod. I så fall kan det tänkas att samtliga terminaler vid leverans innehåller en icke önskvärd KVM. Detta är ett terminalspecifikt problem som vi inte undersöker närmare i den här uppsatsen.

8.3.2 Att utveckla skadlig kod genom att utnyttja preverifieringen

En tanke som legat i bakhuvudet ända sedan vi hörde talas om preverifiering var att det verkade som om denna öppnade ett hål i säkerheten. Vi föreställde oss preverifiering som en process som granskade en klassfil och försåg den med någon sorts signering som skulle indikera om den kunde fungera i en mobil terminal med stöd för MIDP. Vi tänkte oss att det kanske gick att lura KVM i en mobil terminal att en klassfil var preverifierad fast den i själva verket inte var det genom att manuellt lägga till denna signering. På så sätt kanske KVM kunde luras att exekvera ogiltig kod. När vi studerat vad preverifiering verkligen var förkastades denna tanke. Detta av följande anledningar:

Preverifiering är kort sagt till för att underlätta verifieringen i enheten som annars kan vara en relativt resurskrävande process. Verktyget för preverifiering bygger om bytekoden i en klassfil och ersätter den med semantiskt ekvivalent bytekod som KVM kan tolka och exekvera. Eftersom verifieringen i enheten inte skall acceptera sådan bytekod som preverifieringen byter ut bör en sådan klassfil inte köras. Vid preverifieringen införs också attributet Stackmap, och även det är till för att underlätta verifieringen i enheten. Om Stackmap behövs kommer verifieringen hänvisa till det. Om klassfilen inte genomgått preverifiering finns inte attributet Stackmap där det behövs. Då skall inte enheten acceptera klassfilen. Slutsatsen vi drar är att preverifiering inte öppnar några hål i säkerheten, utan endast är till för att underlätta den verifiering som av säkerhetsskäl måste ske i enheten.

Däremot kan det vara intressant att modifiera bytekoder som preverifieringen orsakat. Poster i attributet Stackmap kan vara intressanta att modifiera för att se hur verifieringen i enheten reagerar på detta. Ett test av detta sker i avsnitt 9.8

8.3.3 Att utveckla skadlig kod genom modifiering av klassfiler

Efter kompilering och preverifiering är det fullt möjligt att gå in i en klassfil och modifiera den innan den förs över till en enhet för att exekvera. Detta är givetvis möjligt att göra även för klassfiler i J2SE och J2EE, men eftersom verifieringen i J2ME är reducerad finner vi det intressant att undersöka om modifieringar i klassfilen upptäcks eller möjliggör exekvering av otillåten kod. Det finns mängder av ställen i en klassfil som kan vara intressanta att modifiera.

Detta anser vi dock vara en alltför tidskrävande process. Därför kommer vi att välja några angreppspunkter i en enkel klassfil som vi antar ger samma resultat för liknande modifikationer. Följande punkter verkar vara intressanta testfall:

1. Förvanska strukturen genom att ändra kritiska värden som till exempel antalet metoder, antalet klassvariabler, antalet element i listan över konstanter eller antalet attribut i någon attributlista. Det är intressant att se om en klassfil modifierad på detta vis passerar den interna verifieringen, och i så fall om applikationen klarar av att referera till alla metoder, variabler, konstanter och attribut.
2. Ändra en instruktion i bytekoden till en instruktion som KVM inte skall kunna exekvera. Till exempel en flyttalsinstruktion eller en instruktion som inte finns. Det är intressant att se om en sådan instruktion ens passerar den interna verifieringen, och i så fall vad som händer när KVM försöker exekvera instruktionen.
3. Pröva hur viktigt det är med klassens arv och hur verifieringsprocessen hanterar otillåtna kombinationer av tillgänglighetsflaggorna (till exempel `final` och `abstract`). Går det att skapa ett cirkulärt beroende som får enheten att bete sig på ett icke önskvärt sätt genom att låta en klass ärva sig själv? Passerar en sådan klassfil ens den interna verifieringen?
4. Modifiera attributet Stackmap för att studera både hur den mobila enheten hanterar attributet och hur verifieringen används för att underlätta den interna verifieringen. Detta test föreslogs i analysen i avsnitt 8.3.2.
5. Förändra versionsnumren för vilken kompilator klassfilen kompilerats med. Vi frågar oss om KVM behandlar klassfiler kompilerade med olika kompilatorer på olika sätt och om klassfilerna kan passera den interna verifieringen om versionsnumren är orealistiska.

Dessa områden är endast en liten del av mängden möjliga modifieringar, men de utgör den del vi funnit vara mest potentiell att manipulera för att orsaka upphov till skadlig kod. Med ett antal tester inom dessa områden gjorda kan vi givetvis inte dra en heltäckande slutsats om möjligheten att utveckla skadlig kod med systemnära delar i J2ME. Dock anser vi oss ha undersökt en tillräcklig del för att kunna anta att en sådan slutsats är rimlig. Testerna är dokumenterade i avsnitt 9.8.

8.3.4 Förslag på tester av skadlig kod med systemnära delar av J2ME

I kapitel 9 kommer följande test att utföras baserad på analysen i detta avsnitt:

- Avsnitt 9.8: Klassfilsmodifiering.

8.4 Möjligheten att utveckla skadlig kod med utökningar i Siemens SL45i

Utökade applikationsprogrammeringsgränssnitt ökar möjligheterna att utveckla applikationer till en viss terminal. Frågan vi ställer oss är om möjligheterna att utveckla skadlig kod också ökar. Siemens SL45i har några utökningar som verkar intressanta att undersöka närmare. Ett område som vi valt att helt utesluta är det paket som underlättar utveckling av spel. Detta för att det vid en närmare studie inte verkar finnas några utökningar som medför några aspekter utöver de vi tagit upp i avsnitt 8.1.1 och 8.1.4.

8.4.1 Att utveckla skadlig kod genom åtkomst till otillåtna delar av filsystemet

Siemens SL45 i erbjuder ett alternativ till RMS som åstadkoms med klassen `File`. Anledningen till att vi valde att testa om RMS möjliggjorde åtkomst till andra programsviters filer i avsnitt 8.1.2 var att vi av en händelse upptäckte att emulatorerna i Wireless Toolkit [30] möjliggjorde det. Då det klart och tydligt står i dokumentationen för `File` att klassen endast kan komma åt filer i den mapp som applikationen finns placerad i anser vi det inte finnas någon anledning att testa detta.

8.4.2 Att utveckla skadlig kod med hjälp av terminalspecifika telefonifunktioner

Klassen `PhoneBook` möjliggör åtkomst till terminalens inbyggda telefonbok, men endast delen med missade samtal. Posterna där går bara att läsa, inte förändra eller radera. Dock medför det faktum att de är åtkomliga en möjlig attack i syfte att kränka användarens personliga integritet eftersom en applikation kan se vem som ringt till denne. Numren kan sändas till en illasinnad persons server på det vis vi kom fram till i avsnitt 8.1.3.

`Call` är en klass som kan sätta upp ett samtal. När en applikation gör en sådan begäran tillfrågas användaren att bekräfta varefter applikationen avslutas. Med denna funktion kan en applikation som ringer upp dyra betaltjänster tillverkas, men användaren måste bekräfta att samtalet får ske. En metod för att lura användaren att gå med på att samtalet sätts upp beskrevs i avsnitt 8.1.4.

Klassen `SMS` klarar av att skicka SMS. Ett SMS skickas utan att applikationen avslutas, men kräver fortfarande bekräftelse från användaren för att skickas. Genom att kombinera möjligheten att komma åt telefonboken, metoden att lura användaren som beskrevs i avsnitt 8.1.4 och möjligheten att skicka SMS kan det gå att få en användare att skicka ett SMS till någon som ringt till användaren med innehåll som utvecklaren bestämmer. Innehållet skulle kunna vara av hotande eller kränkande karaktär. I avsnitt 9.9 beskriver vi en applikation som kombinerar möjligheten att skicka SMS och metoden att lura en användare med ett lockande

namn (FreeSMS) som beskrevs i avsnitt 8.2.2. Det kan bli en dyr historia för användaren om denne på allvar tror att applikationen skickar SMS gratis.

8.4.3 Att utveckla skadlig kod med terminalspecifik sändning och mottagning av data

Klassen `Connection` möjliggör sändning av data och gränssnittet `ConnectionListener` möjliggör mottagning av data. Eftersom det går att skicka och ta emot data av typen ”SMS” är det intressant att undersöka om en applikation som implementerar `ConnectionListener` kan ta emot ”vanliga” SMS som skickats till enhetens inbyggda program för mottagning av SMS. Om det går kan programmet ta emot SMS som inte är menade för applikationen, läsa innehållet och skicka detta till en illasinnad persons enhet eller server. Detta kränker den personliga integriteten. I avsnitt 9.10 testar vi om det går att utföra en sådan attack.

8.4.4 Förslag på tester av skadlig kod med utökningar i Siemens SL45i

I kapitel 9 kommer följande tester att utföras baserad på analysen i detta avsnitt:

- Avsnitt 9.9: Lura användaren med telefonifunktioner i Siemens SL45i
- Avsnitt 9.10: Avlyssning av inkommande SMS med Siemens SL45i.

9 Test av möjligheten att utveckla och sprida skadlig kod

I detta kapitel presenterar vi de tester som analysen i kapitel 8 gett upphov till. Från testernas resultat drar vi slutsatser om säkerheten inom det område testet utförts. Slutsatserna kan ge upphov till ytterligare tester som också presenteras här. Bilaga A innehåller dokumentationer av samtliga tester.

Då vi haft stora problem med nedladdning och överföring av applikationer till Motorola Accompli 008 har de flesta tester endast kunnat utföras på Siemens SL45i.

9.1 Åtkomst till andra programsviters RMS

Med klassen `RecordStore` i MIDPs applikationsprogrammeringsgränssnitt går det att skapa och radera filer. Enligt specifikationen skall åtkomst till dessa filer endast kunna fås av applikationer som ingår i den programsvit vars RMS filerna skapats i. Som nämnts i avsnitt 8.1.2 har vi gjort en intressant iakttagelse gällande emulatorerna som medföljer Wireless Toolkit [30]: I dessa saknas restriktionen med åtkomst till andra programsviters RMS-filer. Om det går att komma åt filer i en annan programsvits RMS och radera dem i en verklig enhet så har enheten ett stort hål i säkerheten. Vårt mål är att undersöka huruvida detta är fallet.

9.1.1 Metod

Klassen `RecordStore` innehåller de båda metoderna `listRecordStores` och `deleteRecordStores`. `listRecordStores` returnerar en strängarray innehållande namn på RMS-filerna i den programsvit som applikationen ingår i. `deleteRecordStores` raderar en fil med angivet namn. Genom att kombinera dessa båda metoder kan vi radera samtliga filer i en programsvits RMS. Om `listRecordStores` till följd av ett implementationsfel returnerar namn på filer utanför programsvitens RMS kommer försök att radera dessa utföras. Källkoden till applikationen är dokumenterad bilaga A.1.

9.1.2 Resultat

Varken Motorola Accompli 008 eller Siemens SL45i tillåter en applikation att få åtkomst till andra programsviters RMS. Dessa båda terminaler fungerar alltså enligt specifikationen. Som tidigare nämnts ges en applikation i emulatorerna som följer med Wireless Toolkit access till filer utanför programsvitens RMS.

9.1.3 Slutsats

Eftersom emulatorer skall spegla verkliga enheter ska en utvecklare kunna provköra sin applikation i en emulator och förvänta sig samma resultat i den verkliga enheten. En utvecklare kan skapa en hel applikation som bygger på åtkomst till andra programsviters RMS och få den att exekvera som den ska i emulatorerna. När utvecklaren sedan för över sin applikation till en verklig enhet och upptäcker att den inte fungerar är redan många timmars arbete förgäves. För ett företag som investerat stora pengar i applikationen är det ekonomiska bakslaget klart. Därför anser vi att emulatorerna som medföljer Wireless Toolkit inte är helt bra. Vad gäller säkerheten i de båda enheter vi testat kan vi bara konstatera att den är bra och att implementationen av javaplattformen följer specifikationen, åtminstone inom detta område.

9.2 Lura en användare med grafiskt gränssnitt och HTTP-uppkoppling

Enligt specifikationen skall MIDP implementera en HTTP-uppkoppling. Via uppkopplingen kan information skickas till en server. Vidare har MIDP stöd för att presentera grafiska användargränssnitt, både med färdiga komponenter som exempelvis textrutor och genom manipulation av enskilda punkter på skärmen. Målet är att undersöka hur osynlig en uppkoppling är för användaren samt påvisa möjligheten att skicka känslig information till servern genom att lura användaren med ett iscensatt grafiskt användargränssnitt.

9.2.1 Metod

I bilaga A.2 finns källkoden till applikationerna JFakePIN och JServer. JFakePIN är en applikation som utnyttjar en textruta (klassen `TextBox`) i MIDPs grafiska användargränssnitt för att lura användaren att mata in dennes PIN-kod. Applikationen kopplar därefter upp enheten mot en server och sänder iväg det användaren matat in till den. JServer är det program på servern som tar emot den information som JFakePIN skickat.

JFakePIN är mycket enkel: Applikationen består endast av en textruta som kan visa upp till åtta siffror representerade av asterisker, och ett bekräftningskommando på en av funktionsknapparna. När användaren bekräftar inmatningen kopplas enheten upp mot servern med inmatat värde som bifogad parameter till JServer och därefter avslutas applikationen.

9.2.2 Resultat

Exekveringen av JFakePIN i Siemens SL45i fungerar som den ska: Användaren ombeds mata in sin PIN-kod, inmatningen bekräftas, enheten kopplar upp sig mot servern och på den skärm

som är kopplad till servern skrivs PIN-koden ut. Terminalen tar lite tid på sig att koppla upp mot servern och under tiden visas ett meddelande om att detta håller på att ske. Meddelandet visas av själva enheten och inte av vår applikation.

9.2.3 Slutsats

JFakePIN kan modifieras med den del av MIDPs applikationsprogrammeringsgränssnitt som tillåter manipulering av varje enskild punkt på enhetens skärm. På så vis går det att bygga upp en bild som är identisk med den som enheten själv visar upp när användaren ombeds mata in sin PIN-kod. JFakePIN kan naturligtvis även modifieras för att lura användaren att mata in annan information än sin PIN-kod, exempelvis ett bankkortsnummer enligt det scenario som beskrevs i avsnitt 8.1.3.

Fortfarande handlar det då om att lura användaren att mata in känslig information, fast med en mer trovärdig metod. Genom att förfina användargränssnittet anser vi att det kan gå att lura en användare att inte reflektera över meddelandet om uppkoppling som terminalen automatiskt visar då den kopplar upp sig mot servern. Ett exempel på detta gavs i avsnitt 8.1.3 där en användare angett sitt bankkortsnummer och tror att enheten ansluter sig till en banks dator. Denna form av attack ligger inom ramarna för javaplattformens säkerhetsmodell och utnyttjar fullt giltig kod för att orsaka skadlig verkan för användaren. Detta är ett säkerhetshål som förmodligen inte går att göra något åt eftersom lösningen skulle innebära begränsningar av alltför mycket funktionalitet.

9.3 Applikation utan innehåll

Alla klasser som ärver klassen `MIDlet` blir när de kompileras applikationer som JAM i en enhet kan starta. I avsnitt 8.1.5 ställde vi oss frågan hur en enhet beter sig när en klass som inte innehåller någonting utöver informationen att den ärver klassen `MIDlet` skall laddas och köras. Målet med testet är att undersöka detta och testet utförs på Siemens SL45i.

9.3.1 Metod

Vi har skapat en klass som ärver klassen `MIDlet`, och som endast innehåller tomma funktionskroppar för de metoder som måste implementeras. Källkoden finns dokumenterad i bilaga A.3.

9.3.2 Resultat

Ibland kan de enklaste medel visa sig vara fullt tillräckliga för att skapa skadlig kod. Den mobila terminalen Siemens SL45i är ett bra exempel på detta: En MIDlet som inte innehåller någon kod alls laddas nämligen i all evighet, eller åtminstone tills batteriet tar slut eller användaren väljer att avbryta laddningen. Vad detta beror på vet vi inte men det verkar troligt att det är ett terminalspecifikt problem som vi bara kan konstatera.

9.3.3 Slutsats

Detta faktum, att en giltig applikation inte kan köras, kan vara skadligt för distributören. Om en användare märker att en MIDlet från en distributör bara laddas utan att börja exekvera kan det hända att han väljer att aldrig mer ladda ned applikationer från just den distributören. På så vis kan en illasinnad utvecklare med sin egentligen ganska harmlösa applikation reducera antalet kunder hos en distributör och samtidigt se till att användaren går miste om eventuella applikationer som denne kunde ha haft nytta av.

Det är troligt att fenomenet är ett implementationsfel som vi anser att Siemens borde ha tänkt på.

9.4 Belastning av enheten

Enligt specifikationen för CLDC behöver inte en enhet stödja parallell exekvering av trådar inom en process. Siemens SL45i gör dock det. Vårt mål är att undersöka hur denna terminal beter sig när antalet parallellt exekverande trådar blir alltför stort. Vidare undersöker vi dess beteende då antalet öppnade RMS-filer och instansierade objekt ökar.

9.4.1 Metod

En klass som ärver klassen `Thread` blir implicit ett objekt som kan köras som en tråd. Metoden `start` startar tråden och metoden `run` innehåller de instruktioner som tråden skall exekvera. Klassen `RecordStore` innehåller metoden `openRecordStore` vilken öppnar och skapar en fil med angivet namn i programsvitens RMS.

9.4.2 Testfall

Vi har implementerat några enkla testapplikationer som kombinerar belastningsmetoderna på följande sätt:

1. En tråd som när den exekverar skapar en ny tråd av samma typ och därefter avslutas. Detta program har alltid minst en tråd som exekverar, och det är intressant att se vad

som händer när antalet påbörjade och avslutade trådar ökar och vad som händer om den virtuella maskinens skräpsamlare (garbage collector) inte hinner rensa bort de förbrukade trådarna.

2. En tråd som ideligen skapar nya trådar av samma typ utan att någonsin avslutas. Detta program är en mer intensiv variant av programmet i föregående testfall. Här är det i stället intressant att se hur terminalen beter sig då antalet aktiva trådar blir stort.
3. En tråd som ideligen skapar och öppnar nya filer med olika namn. Detta program testar hur terminalen beter sig när antalet öppnade filer blir stort, hur lång tid det tar för den att skapa och öppna en fil under dessa omständigheter och hur många filer som kan skapas och hållas öppnade samtidigt.
4. En tråd som ideligen skapar nya filer med olika namn och en tråd som skapar nya trådar av samma typ. Detta program belastar enheten både med trådar och filer och vi anser att det är intressant att se hur terminalen beter sig.

Gemensamt för alla testapplikationer är att de innehåller en textruta (klassen `TextBox`) som ger användaren en förklarande text till vad applikationen gör för tillfället, det vill säga om en tråd eller en fil skapas. I textrutan går det även att mata in egen text. Detta för att kunna testa om enheten fortfarande svarar eller ej. Källkoden till samtliga testfall är dokumenterad i bilaga A.4.

9.4.3 Resultat

De olika metoderna resulterade i följande:

1. När applikationen körts ett tag går det inte längre att mata in text i textrutan. Istället hörs en gäll ton från terminalen då någon knapp trycks ned. Ytterligare en intressant iakttagelse är denna: Om applikationen avbryts manuellt och ett försök att åter starta JAM (programmet som hanterar exekvering och installation av javaapplikationer i en enhet) görs, så misslyckas detta och terminalen ger ifrån sig samma ljud som tidigare vilken knapp som än trycks ned. När terminalen startats om är allt som vanligt igen. Värt att nämna är att emulatorerna i Wireless Toolkit [30] också slutade svara efter en stund.
2. När programmet exekverat ett tag levererar terminalen meddelandet ”Out of heap memory”. Detta innebär att antalet trådar blivit för stort och minnet tagit slut. Ytterligare en intressant iakttagelse är denna: Om applikationen avbryts manuellt innan meddelandet levererats och JAM startas om, så ges felmeddelandet ändå och därefter syns markören från textrutan i bakgrunden på terminalens skärm. Värt att

nämna är att emulatorerna i Wireless Toolkit [30] efter en liten stund levererade felmeddelandet ”Out of heap memory”.

3. Efter ungefär en minuts exekverande har nio filer skapats och därefter händer inget på en lång stund. Vi lät applikationen exekvera i ytterligare fem minuter och då inget nytt hänt avbröt vi programmet. I övrigt hände inget underligt med enheten. Värt att nämna är att emulatorerna i Wireless Toolkit [30] kan skapa och öppna betydligt fler filer. Vi avbröt emuleringen då över tusen filer skapats, och då hade hastigheten för skapandet av nya filer minskat betydligt jämfört med hastigheten vid applikationens uppstart.
4. Tråden som skapar nya trådar dominerar och den tråd som skapar nya filer lyckas inte skapa en enda fil. I övrigt beter sig terminalen på exakt samma vis som i resultatet av testfall 1.

9.4.4 Slutsats

Följande slutsatser har dragits för de fyra testfallens resultat:

1. Vi drar slutsatsen att terminalen slutar svara då antalet påbörjade och avslutade trådar blir alltför stort. Att terminalen beter sig underligt även efter att applikationen avbrutits antar vi beror på att Siemens SL45i har dålig minneshantering för minne på heapen. Heapen är ett minnesområde där instansierade objekt hamnar, däribland instanser av klassen `Thread`. [33]
2. För testfall 2 drar vi slutsatsen att minnet i terminalen tar slut då antalet aktiva trådar blir för stort. Det underliga beteende som terminalens uppvisar då applikationen avbrutits antar vi beror på att Siemens SL45i har dålig minneshantering för minne på heapen. [33]
3. Eftersom endast nio filer lyckades skapas drar vi slutsatsen att Siemens SL45i inte kan hålla fler än nio filer öppna samtidigt.
4. Från resultatet av testfall 3 drar vi slutsatsen att den tråd som skapar nya filer kräver mycket resurser. Eftersom den tråd som skapar nya trådar dominerade lyckades därför inte en enda fil skapas.

Det råder inget tvivel om att överanvändning av trådar kan få terminalen att bete sig underligt. När terminalen startas om verkar dock terminalen fungera normalt igen. Detta är dock inte hela sanningen. Efter att vi slutfört samtliga fyra testfall och startat om terminalen anslöt vi den till en persondator och undersökte dess innehåll. Det visade sig att inte mindre än fem kopior gjorts av den mapp som innehåller javaapplikationer i terminalen. Det verkar troligt att dessa skapats då försök att starta om JAM gjorts efter att exekvering av

testapplikationerna avbrutits manuellt. Varför detta skett faller dock utanför uppsatsens avgränsning och är något vi endast konstaterar och inte undersöker närmare.

Med tanke på att meddelandet ”Out of heap memory” levereras trots att applikationen avbrutits i testfall 2 och 4, verkar det som vissa eller alla trådar som skapats i en applikation fortsätter att exekvera efter att applikationen avbrutits manuellt. Enligt specifikationen tillåter inte CLDC så kallade döda trådar, det vill säga trådar som exekverar efter att applikationen som de tillhör har avslutats. Detta kan bero på att Siemens SL45i inte hanterar avslutade applikationer på samma vis som avbrutna, eller på att terminalen har dålig hantering av minne på heapen. Det kan också vara ett fel i implementationen på Siemens SL45i. Huruvida så är fallet faller också utanför uppsatsens avgränsning och terminalens underliga beteende är därför något vi enbart konstaterar. [8][33]

Det faktum att ett grafiskt fel inträffar som beskrivits i resultatet för testfall 2 kan tyda på att oavslutade trådar i den manuellt avbrutna applikationen faktiskt skriver till en minnesadress som applikationen inte längre skall ha åtkomst till. Detta är intressant, och med en tråd som egentligen inte längre skall kunna exekvera kanske skadlig kod kan skapas. I avsnitt 9.5 testas återigen möjligheten till åtkomst av andra programsviters RMS, fast denna gång med dessa oavslutade trådar. Om en tråd från en avbruten applikation fortfarande körs i bakgrunden kan det tänkas att den får åtkomst till en nystartad applikations minnesområde, och därmed också dess RMS-filer.

9.4.5 Förslag på tester av skadlig kod som slutsatserna i detta avsnitt gett upphov till

- Avsnitt 9.5: Åtkomst till andra programsviters RMS med oavslutade trådar

9.5 Åtkomst till andra programsviters RMS med oavslutade trådar

I avsnitt 9.1 gjordes en undersökning som visade att en applikation inte hade åtkomst till andra programsviters RMS-filer med den funktionalitet som finns i MIDP. I detta avsnitt ska vi använda oss av en något annorlunda metod för att undersöka möjligheterna att komma åt andra programsviters RMS. Metoden bygger på resultaten och slutsatserna i avsnitt 9.4.

9.5.1 Metod

Resultaten och slutsatserna från avsnitt 9.4 visar tendenser till att massor av trådar kanske kan exekvera i bakgrunden trots att en applikation avbrutits. Den applikation vi ska implementera använder samma metoder som de som beskrivs i avsnitt 9.1.1 och 9.4.1, samt fungerar som applikationen beskriven i testfall 2 i avsnitt 9.4.2 med några modifikationer.

Applikationen skapar en tråd som har som uppgift att först radera samtliga filer i programsvitens RMS och därefter skapa en ny tråd av samma typ. Tråden avslutas inte, utan upprepar samma förfarande i en oändlig slinga. Skapandet av nya trådar på detta vis används för att återskapa effekten av att trådarna ligger kvar i bakgrunden och exekverar trots att applikationen avbrutits så som resultaten och slutsatserna i avsnitt 9.4 gav. Metoden för att komma åt andra programsviters RMS-filer är att manuellt avbryta applikationen och snabbt starta en annan applikation. Källkoden till vår applikation är dokumenterad i bilaga A.5.

9.5.2 Resultat

Vår applikation, låt oss kalla den A, verkar ligga kvar i bakgrunden av terminalen och exekvera en stund efter att den avbrutits. Efter att manuellt ha avbrutit applikation A startar vi en annan javaapplikation (B). Då visas för ett kort ögonblick applikation A i skärmen innan ett felmeddelande om att applikation B inte kunde startas levereras av terminalen. Görs därefter omedelbart eller efter en liten stund ett nytt försök att starta applikation B går det alldeles utmärkt.

Genom att ansluta terminalen till en persondator kunde vi konstatera att applikation A inte lyckades radera filer i den programsvit som applikation B ingår i.

9.5.3 Slutsats

Terminalens underliga beteende vid uppstart av en mångfald av trådar möjliggör inte åtkomst till filer i andra programsviters RMS. Därmed är det inte sagt att detta underliga beteende kan ha andra effekter och öppna andra, i så fall, terminalspecifika säkerhetsluckor. Huruvida så är fallet undersöker vi inte närmare i denna uppsats eftersom det faller utanför dess avgränsning.

9.6 Injicera en skadlig MIDlet i en redan installerad programsvit

Som nämndes i analysen i avsnitt 8.2.3 är det intressant att undersöka om det går att injicera en MIDlet i en redan installerad JAR-fil och få enheten att låta denna MIDlet exekvera. Om detta är möjligt är det troligt att denna injicerade MIDlet får tillgång till programsvitens RMS-filer. Målet med detta test är att undersöka om det går att utföra en sådan injicering genom att ansluta enheten till en persondator, och i så fall om det går att starta en injicerad MIDlet och om denna har tillgång till programsvitens RMS-filer.

9.6.1 Metod

Vi ansluter en mobil terminal av märket Siemens SL45i till en persondator med programvara installerad för att kunna kommunicera med terminalen. För att kunna öppna, läsa och ändra en JAR-fil använder vi oss av programmet WinZip [39]. Istället för att skapa en egen applikation och installera den använder vi en redan befintlig applikation i terminalen. Själva processen försöker vi utföra enligt följande: Först kompileras den MIDlet vi skall injicera och därefter försöker vi med hjälp av WinZip lägga till den klassfil som genereras vid kompileringen i JAR-filen. Sedan modifieras JAD-filen och manifestfilen där vi anger att ytterligare en MIDlet finns i programsviten, samt dess namn. Källkoden till den applikation vi skall injicera är dokumenterad i bilaga A.6.

9.6.2 Resultat

Vi upptäckte att det går att lägga in klassfiler som ärver `MIDlet` i en JAR-fil och få dem synliga som MIDlets i enheten efteråt. Emellertid är det inte JAR-filen i den mobila terminalen som modifieras utan en öppnad kopia som skapats genom att WinZip läst in JAR-filen i datorns primärminne. När vi är färdiga med modifikationen och avslutar WinZip sparas den öppnade kopian ned till terminalen. Vad som hänt är alltså att JAR-filen i terminalen ersatts med en annan JAR-fil med snarlikt innehåll. Eventuella RMS-filer tillhörande den programsvit som den gamla JAR-filen utgjorde är intakta och åtkomliga från den injicerade MIDleten.

9.6.3 Slutsats

Vi har visat att det går att injicera en MIDlet i en redan installerad programsvit, men också att detta är detsamma som att ersätta den befintliga JAR-filen med en ny och ändå behålla eventuella RMS-filer. Detta anser vi dock inte utgöra något hål i säkerheten eftersom användaren själv måste ansluta sin enhet till en dator och injicera en skadlig MIDlet eller byta ut den redan installerade JAR-filen mot en annan som innehåller en skadlig MIDlet. Om det i framtiden däremot skulle gå att modifiera JAR-filer genom funktionalitet i Java är det inte omöjligt att en säkerhetslucka uppstår på detta område. En annan installerad applikation kan i så fall kopiera in sig själv i en viss programsvit och på så vis få tillgång till dess RMS-filer. Detta är dock bara spekulationer.

9.7 Versionsuppgradering till ett program med skadlig kod

En användare vill i regel hålla applikationerna uppgraderade. Mekanismer för detta finns inbyggda i OTA-specifikationen och nedladdningsförfarandet. Vårt mål är att undersöka huruvida en mobil terminal behåller RMS-filer vid en uppgradering eftersom detta enligt OTA-specifikationen är valfritt. Om detta är fallet har vi visat tidigare att en applikation kan exponera data via till exempel sändning över HTTP. [25]

9.7.1 Metod

För att genomföra versionsuppgradering behövs två MIDlets med samma namn där den ena har lägre versionsnummer än den andra. Version 1.0 av MIDleten JPassword är en seriös, icke skadlig MIDlet som skulle kunna vara ett program som lagrar lösenord åt användaren. Det viktiga är att MIDleten skapar RMS-filer. Version 1.1 av JPassword är en skadlig MIDlet som har till uppgift att lista eventuellt befintliga RMS-filer för att sedan radera dem. Eftersom denna version har ett högre versionsnummer skall uppgradering vara möjlig. JAR- och JAD-filer till version 1.0 och version 1.1 av JPassword finns dokumenterade i bilaga A.7.

9.7.2 Resultat

I Siemens SL45i installerades först version 1.0 av JPassword och exekverades för att generera en RMS-fil. Version 1.1 laddades därefter ned och installerades utan några problem. Anmärkningsvärt är att användaren måste bekräfta att skriva över den äldre versionen, inte bekräfta att uppgradera den. Vid körning av version 1.1 bekräftades att RMS-filen var intakt och den togs bort.

9.7.3 Slutsats

En säkerhetslucka uppstår genom proceduren med uppgradering av MIDlets. OTA-specifikationen tar upp denna säkerhetslucka och där står uttryckligen att det är en säkerhetsrisk. Om en användare kan luras att ladda ned en falsk uppgradering av ett känt program, en uppgradering som en illvillig person släppt fri på Internet, riskerar användaren att exponera känslig data. Åtminstone om denne använder Siemens SL45i eftersom den behåller RMS-filer vid uppgraderingen.

Om en mobil terminal inte behåller RMS-filer är det tveksamt om det skall kallas uppgradering. Vad som händer då är att hela programsviten raderas och en annan med samma namn installeras. Samma effekt kan åstadkommas utan versionsuppgradering om användaren själv raderar en applikation innan en annan med samma namn och högre versionsnummer laddas ned och installeras. Vi anser att det är bra att Siemens SL45i behåller RMS-filer vid en

uppgradering, men att det är dåligt att en säkerhetslucka uppstår på grund av det undermåliga sätt uppgraderingen sker på.

En spekulation vi har är att om det i framtiden går att installera enskilda MIDlets i en redan installerad programsvit, alltså lägga till MIDlets i programsviter, så innebär det en säkerhetsrisk på samma sätt som vi beskrev i avsnitt 9.6.3. En injicerad MIDlet kommer åt programsvitens RMS-filer.

Värt att nämna är också att en applikation inte behöver skapa RMS-filer med känslig information för att vara mål för en falsk versionsuppgradering. Ett exempel på detta är den applikation vi nämnde i avsnitt 8.1.3 som kontrollerar saldot på användarens bankkonto med hjälp av numret på det bankkort som är kopplat till kontot. Antag att det faktiskt finns en sådan applikation som banken själv tillverkat. Denna applikation kommunicerar med bankens server och returnerar saldot på det konto som är kopplat till angivet kortnummer. En illasinnad utvecklare skapar en applikation identisk med bankens, med det undantaget att applikationen kommunicerar med utvecklarens server istället för bankens. Den illasinnade utvecklaren skickar sedan ett SMS till användare av bankens saldoapplikation. I detta SMS står det att banken gjort en ny, gratis version av sin applikation och att det bara är att följa den hyperlänk som är angiven för att ladda ned applikationen. Länken leder i själva verket till utvecklarens applikation som vid installation ersätter bankens applikation. Eftersom den illasinnade utvecklarens applikation på ytan är identisk med bankens, har inte de användare som väljer att uppgradera någon anledning att misstro den. Användarna kommer av denna anledning exponera sitt bankkortsnummer nästa gång de vill veta sitt saldo.

9.8 Klassfilsmodifiering

Vi vill undersöka vilka effekter som uppkommer i den mobila terminalen Siemens SL45i vid modifiering av klassfiler. Målet är att modifiera strukturen i en klassfil samt ändra den bytekod som finns lagrad i attributen Code och Stackmap. Testapplikationen vars klassfil vi modifierar utför ingenting av värde. Dess enda uppgift är att fungera som en giltig MIDlet.

Anledningen till att vi utför testerna är för att undersöka om den virtuella maskinen klarar av att upptäcka ogiltiga bytekoder, eller om den exekverar dem på ett för användaren icke önskvärt sätt. Om en tillverkare följer specifikationen för KVM skall testerna endast resultera i att MIDleten inte accepteras vid inladdning eller att den inte går att köra.

9.8.1 Metod

För att spara tid och minska komplexiteten i sökandet efter värden och strukturer använder vi oss av vårt verktyg ClassToXML. ClassToXML är skrivet i Java med funktionalitet från J2SE och källkoden finns dokumenterad tillsammans med en demonstration av hur det används för att analysera en klassfil i bilaga B. ClassToXML tar en kompilerad klassfil som inargument och genererar ett XML-dokument med en beskrivning av innehållet. XML-dokumentet kan visas i till exempel Microsoft Internet Explorer 5 [40]. För att ytterligare underlätta analysen av klassfilen infogar ClassToXML vissa offsetvärden så det går snabbt att i en hexredigerare leta upp precis den byte som avses. Den hexredigerare vi använt oss av heter Hackman [31]. XML är ett språk för uppmärkning av data och genom att använda sig av språket synliggörs klassfilsstrukturen och möjliggörs en enkel traversering av den. [32]

9.8.2 Testfall

I avsnitt 8.3.3 listades de strukturer i en klassfil som vi ansåg särskilt intressanta att modifiera. Följande modifieringar kommer göras i dessa strukturer:

1. Klassfilen innehåller ett antal konstanter i listan över konstanter, se avsnitt 6.2.2. I början av denna lista finns ett värde som anger hur många konstanter listan innehåller. Vi kommer att ändra värdet så att antalet konstanter ökas med ett. Efter att detta testats går vi vidare med att modifiera antalet element som maximalt får finnas på stacken när en viss metod exekveras. Först ändrar vi värdet så att det blir mindre än ursprunget och sedan ökar vi det så att det blir större än ursprunget. Till sist prövar vi att ange det maximala värdet.
2. I vårt andra test modifierar vi den bytekod som utgör de instruktioner som skall exekveras. Denna ligger lagrad i attributet Code. En instruktion som i normala fall hanterar vanliga heltal byts ut mot en likartad instruktion som istället hanterar flyttal. Därefter ersätter vi samma instruktion med en reserverad instruktion. Tills slut prövar vi att ersätta instruktionen med en instruktion som har ett värde som inte är definierat i JVMS.
3. I vårt tredje testfall använder vi oss av de fält i klassfilen som pekar ut från vilken klass klassen ärver och vad klassen själv heter. I första deltestet låter vi klassen ärva sig själv och i det andra deltestet låter vi klassen vara den klass den ärver. Om vi kallar klassfilens klass för A och den klass som klassfilens klass ärver från för B så gör vi alltså följande ändringar: Först sätter vi As namn som klassnamn på både A och B och därefter Bs namn som klassnamn på både A och B.

4. Vårt fjärde testfall utgörs av tre deltester som modifierar attributet Stackmap. Först ändrar vi antalet element i listan över datatyper så att det är mindre än tidigare. Sedan ändrar vi det så att det blir större än det ursprungliga. Slutligen testar vi att helt ta bort ett element i listan. Vi vill med dessa tester undersöka hur terminalen reagerar på en korrupt Stackmap. Eftersom Stackmap är ett attribut som vid behov tillkommer när preverifiering utförs kan det hända att den interna verifieringen inte vet hur den ska hantera en korrupt Stackmap. Detta eftersom det inte finns någon inbyggd funktionalitet för att skapa detta attribut.
5. De versionsnummer som finns angivna i klassfilen anger vilken kompilatorversion som använts för att kompilera klassfilen. Vi byter ut dessa värden mot andra godtyckliga värden och undersöker om verifieringen och KVM överhuvudtaget använder denna information, och i så fall om exekveringen blir olika med olika versionsnummer.

I bilaga A.8 har vi dokumenterat källkoden, klassfilen och XML-dokumentet som ClassToXML genererar för vår testapplikation JSimple. Exakt vilka bytekoder som ändrats, vilken position dessa har i klassfilen, samt vilka värden vi ändrat dem till är också angivna. Även de modifierade klassfilerna samt de XML-dokument som ClassToXML genererat av dessa är dokumenterade i bilagan.

9.8.3 Resultat

1. Vid ändring av antalet klasskonstanter passerade inte klassfilen verifieringen. Detsamma gäller ändringen till ett färre antal element på stacken för en metod. Den klassfil som modifierats för att ha ett högre antal element på stacken än det ursprungliga passerade däremot verifieringen och exekverade normalt. Det verkar finnas en övre gräns på detta värde eftersom terminalen fick slut på minne då vi angav det maximala värdet. Exakt vilket värde detta är anser vi irrelevant att ta reda på och alltför tidskrävande att undersöka eftersom värdet ligger i ett stort intervall innefattande tiotusentals värden.
2. Alla tre deltester med modifiering av bytekod resulterade i att verifieringen inte accepterade klassfilen. Terminalen visade ett meddelande om att ett fel uppstod när applikationen verifierades.
3. Att försöka få klassen att ärva sig själv resulterade i ett meddelande i terminalen om att klassen var cirkulärt definierad. Då vi vill få klassen att vara den klass som klassen

ärver rapporterar terminalen att klassen JSimple inte kunde hittas och applikationen accepteras inte.

4. Vid ändring av antalet element i listan över datatyper till ett högre värde rapporterar enheten att det är fel storlek på attributet. När vi angav ett lägre värde än originalet så fick vi ett felmeddelande om att attributet Stackmap var fel. Även vid borttagandet av ett element i listan får vi meddelandet att storleken är fel.
5. Verifieringen lyckades i samtliga deltester, men inga av modifieringarna resulterade i förändringar i exekveringen.

9.8.4 Slutsats

Vi kan bara konstatera att KVM kontrollerar innehållet i en klassfil mycket noga. Vi är inte säkra på om det endast är KVM i Siemens SL45i som är utomordentligt bra när det gäller verifiering men vår uppfattning är att KVM klarar verifieringen galant. Det verkar därför troligt att enheten reagerar på ett likartat sätt på andra typer av förändringar i klassfilen även om ett definitivt bevis inte kan ges.

9.9 Lura användaren med telefonifunktioner i Siemens SL45i

Vi vill undersöka om det utökade applikationsprogrammeringsgränssnittet i Siemens SL45i ger en utvecklare möjlighet att skapa skadlig kod. Vi har valt att rikta in oss på funktionen för att med en MIDlet skicka ett SMS till en annan mobiltelefon. Vi vill testa hur terminalen hanterar SMS via MIDlets samt undersöka om den upplyser om kostnaden och om den kräver att användaren bekräftar att meddelandet skickas.

9.9.1 Metod

Vi har skapat en applikation som vi valt att kalla FreeSMS som kombinerar möjligheten att skicka SMS och metoden att lura en användare med ett lockande namn som beskrevs i avsnitt 8.2.2. FreeSMS skickar inte SMS gratis, utan heter så endast för att lura användaren att tro att så är fallet. Till sin uppgift har applikationen att skicka ett SMS till ett visst nummer. Numret är lagrat i koden till applikationen för att förenkla utvecklingen. För att få tillgång till mobiltelefonens SMS-funktion används de javaklasser som följer med utvecklingspaketet för Siemens SL45i. Källkoden är dokumenterad i bilaga A.9. [37]

9.9.2 Resultat

Innan Siemens SL45i skickar meddelandet frågar den användaren om lov. Numret som meddelandet skall skickas till visas och möjlighet ges att avbryta proceduren. Ingen kostnadsuppgift ges.

9.9.3 Slutsats

Namnet FreeSMS och det faktum att ingen kostnadsuppgift ges till användaren anser vi tillräckligt för att lura vanliga användare att applikationen skickar gratis SMS. Applikationen skulle kunna modifieras så att den meddelar användaren om att kostnaden för SMSet är noll kronor. På så sätt kanske applikationen blir än mer trovärdig. En snarlik applikation kan tänkas sända andra texter än de inmatade till andra nummer än de angivna. Vi vill påpeka att det krävs en mänsklig faktor för att dessa typer av applikationer skall få genomslagskraft, men också att vi tror att det inte krävs mycket för att lura en naiv användare.

9.10 Avlyssning av inkommande SMS med Siemens SL45i

I avsnitt 8.4.3 utredde vi möjligheten att med gränssnittet `ConnectionListener` ta emot SMS menade för enhetens program för mottagning av SMS. Målet med detta test är att undersöka om vi med hjälp av denna metod kan avlyssna inkommande SMS och på så sätt få tillgång till data som inte är avsedd för applikationen.

9.10.1 Metod

Genom att implementera gränssnittet `ConnectionListener` kopplar den MIDlet vi skapat upp sig mot den port på enheten där data tas emot. Applikationen implementerar metoden `receiveData` i gränssnittet `ConnectionListener`. MIDleten ligger och väntar på inkommande SMS. Om ett sådant kommer är det meningen att innehållet i meddelandet skall skrivas ut på skärmen. Vi prövade att skicka ett SMS till den mobila terminalen från en annan mobiltelefon och studerade vad som hände.

I bilaga A.10 är källkoden till programmet dokumenterad.

9.10.2 Resultat

När ett SMS kom medan vår MIDlet exekverade reagerade Siemens SL45i på ett normalt sätt. Det vill säga att den för en kort stund visade ett meddelande om att ett SMS mottagits för att sedan återvända till exekveringen av vår MIDlet. Ingen text skrevs ut i skärmen.

9.10.3 Slutsats

Vi kunde inte inhämta information från inkommande SMS trots att dokumentationen antyder att det går att skicka data av typen ”SMS”. Misslyckandet kan bero på någon detalj i vår implementation eller på att data av typen ”SMS” inte är samma sak som ett SMS. Vi tror att Siemens tänkt på detta säkerhetshål och inte tillåter en MIDlet att ta emot SMS menade för enhetens program för mottagning av SMS. Vi har inte lyckats hitta någon källa som tar upp hur och om en distinktionen av dessa båda typer av SMS görs.

10 Resultat och rekommendationer

Kapitel 8 innehåller analys av olika områden inom möjligheterna att skapa skadlig kod och vissa delar av denna analys resulterade i tester som är dokumenterade i kapitel 9. Detaljerade resultat och slutsatser finns angivna i samband med testerna och analysen. För att undvika onödig redundans kommer vi här ta upp resultaten och slutsatserna från testerna och analysen översiktligt och hänvisar istället till kapitel 8 och kapitel 9 för detaljer. Vi avslutar med rekommendationer till företag med verksamhet involverande teknologi som vi behandlat i denna uppsats.

Våra tester har inte resulterat i upptäckten av några säkerhetshål som skadar enheternas system på något vis. Däremot har vi funnit andra typer av attacker som är skadliga främst för användaren, men även för utvecklaren och distributören. Ser vi till funktionaliteten i MIDP upptäckte vi att emulatorerna i Wireless Toolkit [30] inte följer specifikationen till hundra procent. Detta eftersom de ger en applikation tillgång till RMS-filer utanför applikationens programsvit.

Möjligheten att koppla upp en enhet till Internet via HTTP innebär inga tekniska säkerhetsrisker. Möjligheten för en illasinnad person att kontrollera en enhet genom uppkopplingen anser vi obefintlig och möjligheten att sända känslig information är inte unik för MIDP och mobila terminaler. De enheter vi testat att skapa uppkopplingar på visar ett meddelande om att detta sker. Det kan gå att få en naiv användare att förbise meddelandet genom att få det att verka som en naturlig del av applikationen. MIDPs grafiska gränssnitt möjliggör manipulering av enskilda punkter på skärmen vilket skulle kunna användas för att skapa bilder som ser ut som och agerar på samma sätt som enhetens verkliga grafiska gränssnitt. På så vis går det att lura användaren att mata in känslig information som exempelvis kan sändas till en illasinnad persons server. Återigen är inte detta unikt för MIDP och mobila terminaler.

Belastning av en enhet skulle kunna få den att fungera på ett icke önskvärt sätt. Vi kom fram till att terminalen Siemens SL45i beter sig underligt då antalet aktiva eller förbrukade trådar blir stort. Eftersom vi upptäckte att trådarna verkade fortsätta exekvera trots att en applikation avbrutits gjorde vi ett test för att se om dessa trådar kunde få tillgång till otillåtet minne. Våra tester resulterade i grafiska fel i enhetens skärm, men gav ingen åtkomst till otillåtna filer.

Att en tillverkare skulle ha implementationsfel i sina terminaler ser vi inte som omöjligt, men alltför tidskrävande att undersöka. Dessutom är det i så fall ett terminalspecifikt problem som faller utanför uppsatsens avgränsning. Att modifiera den virtuella maskinen i en enda terminal så att exekvering av otillåten kod möjliggörs är också lönlöst eftersom det inte finns något sätt att sprida den virtuella maskinen till fler terminaler. Däremot visade ett enkelt test på terminalen Siemens SL45i att applikationer som inte innehåller någonting laddas i evighet. Detta verkar vara en miss i implementationen och något vi anser att Siemens borde ha tänkt på.

Det sätt på vilket en applikation normalt hamnar i en terminal är via nedladdning från Internet. Vi har diskuterat olika sätt för en illasinnad person att placera sin skadliga kod i en användares terminal, till exempel med falsk reklam och lockande applikationsnamn. Ingen av dessa metoder är specifik för MIDP och mobila terminaler. Däremot har vi funnit att det sätt en applikation uppgraderas på möjliggör exponering av känslig information för en skadlig applikation i Siemens SL45i. RMS-filer finns nämligen kvar efter en uppgradering av programsviten och användaren tillfrågas inte om filerna skall vara kvar eller tas bort. En seriös MIDlet som användaren anförtror känslig information kan uppgraderas till en skadlig MIDlet. Det är då troligt att användaren även anförtror den skadliga uppgraderingen känslig information. Versionsuppgraderingen anser vi vara den största bristen i säkerheten vi hittat.

Möjligheten att injicera en skadlig applikation i en redan installerad programsvit för att komma åt programsvitens RMS-filer finns om användaren själv ansluter terminalen till en persondator och injicerar applikationen. Vanlig nedladdning möjliggör nämligen inte injicering av applikationer i redan installerade programsviter. Detta utgör därför inget säkerhetshot.

Vi prövade också att ändra på innehållet i klassfiler. Resultaten i de flesta fall var att klassfilen inte accepterades av enheten eftersom den inte godkändes i den interna verifieringen. Vi kom också fram till att preverifiering inte öppnar några säkerhetshål eftersom processen endast spelar en hjälpande roll i verifieringsprocessen. Vi konstaterar därför att klassfilsstrukturen är mycket säker och inte utgör ett mål för attacker av illasinnade personer.

För att illustrera terminalspecifika funktioner som utökar MIDP testade vi hur Siemens SL45i hanterar sändning av SMS via en MIDlet. Att skicka SMS kostar pengar och testen visade att terminalerna inte visar kostnaden för detta. Vår MIDlet FreeSMS kombinerar terminalspecifik funktionalitet med möjligheten att lura användaren att köra applikationen genom ett lockande namn. Applikationen kan få genomslagskraft eftersom den lurar

användaren att tro att SMS skickas gratis. Ytterligare ett test med SMS visade att en applikation inte kan ta emot SMS menade för det inbyggda program i Siemens SL45i som hanterar mottagning av SMS.

Då J2ME och den funktionalitet som erbjuds genom MIDP har en så bra och genomarbetad säkerhetsmodell uppstår inte säkerhetsbrister om inte en terminaltillverkare gör ett misstag vid implementeringen. Sannolikheten att säkerhetsbrister uppstår i terminalspecifik funktionalitet som utökar MIDP är då större. Förutom att implementationsfel kan förekomma kan den utökade funktionaliteten redan innan den implementeras innehålla ej genomtänkta delar som innebär bristande säkerhet. Utökad funktionalitet innebär också fler möjligheter för en illasinnad utvecklare att lura användaren så som vi visade att sändning av SMS med utökningarna i Siemens SL45i gjorde. Vi anser därför att risken att säkerhetshål uppstår i utökningar av MIDP är större än risken att säkerhetshål uppstår i MIDP. Vidare anser vi att risken för stora säkerhetshål även den är större i utökningar av MIDP än den är i MIDP. Utökningarna möjliggör åtkomst till delar av terminalerna som inte är åtkomliga från MIDP.

Vi vill rekommendera MIDP till de företag som letar efter en plattform för distribution av applikationer till små resurssnåla enheter. Visserligen kan MIDP i sin nuvarande form vara en flaskhals vid utveckling av applikationer eftersom den är så begränsad, men vi påstår att säkerheten är så pass god att fördelarna överväger. Den största bristen med MIDP är sättet versionsuppgraderingar sker på. Säkerhetsbrister i de övriga områdena vi undersökt är inte specifika för MIDP och ofrånkomliga även i många andra plattformar eftersom de ofta styrs av den mänskliga faktorn. En ny generation av MIDP kommer inom en snar framtid och den löser problemet med versionsuppgraderingen, men tillför kanske nya säkerhetshål.

Vi anser också att utvecklingen av applikationer bör innefatta tester på flera enheter för att undvika terminalspecifika problem och säkerhetsbrister. I alla fall bör företag ha en lista över enheter som klarar av MIDP och ha information om eventuella implementationsspecifika säkerhetsbrister i dem. Det är bra att känna till hur enheterna uppför sig i vissa extrema situationer. Som exempel kan nämnas den mobila terminalen Siemens SL45i och hur den betedde sig vid användning av alltför många trådar.

11 Slutsatser

Efter vårt arbete med studier, analys och tester har vi bildat oss en uppfattning om säkerheten i mobila terminaler som implementerar javaplattformen. Enligt oss är det mycket svårt, för att inte säga omöjligt, att utveckla skadlig kod som skadar enhetens system på något sätt. Att utveckla och sprida applikationer som är skadliga för användaren är däremot fullt möjligt och betydligt enklare. Skadlig kod är i det här fallet fullt giltig kod som används i skadligt syfte. För att nyansera det hela lite mer är javaplattformen i mobila terminaler lika säker som sin svagaste länk och i detta fall är den svagaste länken människan.

Så länge det existerar människor som använder applikationer utan att sätta sig in i säkerhetsfrågor kommer det alltid finnas utrymme för applikationer som utnyttjar detta faktum och lurar användaren. Som ett exempel nämner vi det ”virus” som var i omlopp medan denna uppsats skapades. Viruset fanns egentligen inte, utan utgjordes av användaren själv och utnyttjade dennes oro för att drabbas av ett riktigt virus:

Det börjar med att en användare får e-post från en bekant. I brevet påstås att användaren drabbats av ett mycket farligt datorvirus och att det enda som hjälper är att ta bort vissa filer som utgör det påstådda viruset. Sedan uppmanas användaren att skicka vidare brevet till alla denne känner. Självklart existerar inget riktigt datorvirus men hela processen fungerar exakt på ett sådant sätt som kännetecknar virus. Först förstör viruset för användaren genom att göra dennes system obrukbart eftersom filerna som tas bort är kritiska för systemets funktionalitet. Sedan sprider sig viruset vidare genom att användaren vill vara hjälpsam och varna sina vänner på samma sätt som användaren själv blivit varnad.

Även om inget lika drastiskt skulle kunna göras i mobila terminaler eftersom Java är helt isolerat från det övriga systemet, så skulle liknande scenarion åtminstone kunna exponera känslig information. Vi har kommit fram till att javaplattformen i mobila terminaler är mycket säker och att det endast är genom att lura användaren som illasinnade utvecklare kan orsaka skada.

I framtiden kommer mer funktionalitet att läggas till javaplattformen för mobila terminaler. Även om varje funktion är övervägd ur säkerhetssynpunkt kommer oundvikligen plattformen bli mer attraktiv för attacker. Fler funktioner ger avancerade applikationer som behandlar mer data. Populära applikationer från stora programvaruföretag kommer förmodligen att bli mer utsatta för attacker, precis som vi ser har skett i persondatorvärlden.

Vi har under skapandet av vår uppsats införskaffat oss kunskaper om Java i mobila terminaler. Viktiga delområden inom detta är hur utvecklingen går till, hur säkerhetsmodellen ser ut samt hur ett javaprogram är strukturerat både före och efter att det kompilerats. Vi fann det något svårt att hitta källmaterial av tillräckligt hög kvalitet eftersom det saknas bra böcker som behandlar ämnet. Det har inte varit några problem att uppnå målet med uppsatsen. Det har varit någorlunda lätt att komma med uppslag för analys och de flesta av testerna kunde genomföras utan problem. Vi har följt tidsplanen nästan fullt ut och den enda större avvikelsen var avsnittet som behandlade utökad terminalspecifik funktionalitet. Vi upptäckte att detta område var smalare än vi först föreställt oss. Om möjligheten fanns att göra om uppsatsen skulle det för oss inte innebära några större förändringar i upplägget eller innehållet. Vi är nöjda med vårt arbete och känner att uppgiften var både lärorik och utmanade.

Referenser

[1] Sun Microsystems, Introduction to Wireless Java, http://wireless.java.sun.com/getstart, 2002-02-18.

[2] Sun Microsystems, Martin Hardee, Why Wireless needs Java 2 technology, http://java.sun.com/features/2000/07/wireless.print.html, 2002-02-18.

[3] Sun Microsystems, Connected Limited Device Configuration – Public review, http://jcp.org/aboutJava/communityprocess/review/jsr139/index.html, 2002-04-29.

[4] Sun Microsystems, Qusay Mahmoud, Wireless Java security, http://wireless.java.sun.com/midp/articles/security, 2002-02-18.

[5] Catapult Systems, The nature of Java in small things, http://www.ctimn.com/documents/j2me.doc, 2002-02-18.

[6] Sun Microsystems, Qusay Mahmoud, Wireless application programming – MIDP programming and packaging basics, http://wireless.java.sun.com/midp/articles/getstart, 2002-02-18.

[7] Sun Microsystems, JSR-000118 Mobile Information Device Profile – Public review, http://jcp.org/aboutJava/communityprocess/review/jsr118/index.html, 2002-04-29.

[8] Sun Microsystems, JSR-000030 J2ME Connected, Limited Device Configuration, http://jcp.org/aboutJava/communityprocess/final/jsr030, 2000-05-19.

[9] Sun Microsystems, JSR-000037 Mobile Information Device Profile (MIDP), http://jcp.org/aboutJava/communityprocess/final/jsr037/index.html, 2000-12-15.

[10] Siemens, Siemens Mobile Partners, http://www2.siemens.fi/developers.jsp, 2002-02-18.

[11] Sun Microsystems, Java 2 platform, Enterprise Edition authorized Java licensees of J2EE, http://java.sun.com/j2ee/licensees.html, 2002-02-18.

[12] Center for Information Technology, Java history, http://tuts.cit.nih.gov/java_evans/prelims/history.htm, 2002-04-29.

[13] Sun Microsystems, Java Hotspot technology, http://java.sun.com/products/hotspot/index.html, 2002-02-19.

[14] Sun Microsystems, Sun Labs Java Technology Research Group, “Kick butt; have fun; eat lunch; pick up the garbage!”, http://research.sun.com/jtech, 2002-02-19.

[15] Javaworld, Bill Venners, Java's security architecture, http://www.javaworld.com/javaworld/jw-08-1997/jw-08-hood_p.html, 2002-02-19.

[16] Wiley, C. Enrique Ortiz och Eric Giguère, Mobile Information Device Profile for Java 2 Micro Edition, 2001.

[17] Wireless Developer Network, Michael Nygard, A brief look at Java 2 Micro Edition, http://www.wirelessdevnet.com/channels/java/features/j2me.html, 2002-02-19.

[18] Sun Microsystems, Qusay Mohmaud, The J2ME Platform, Which APIs come from the J2SE platform?, http://wireless.java.sun.com/midp/articles/api, 2002-04-29.

[19] Sun Microsystems, The K virtual machine datasheet, http://java.sun.com/products/cldc/ds, 2002-04-29.

[20] Sun Microsystems, Frank Yellin, Low level security in Java, http://java.sun.com/sfaq/verifier.html, 2002-04-29.

[21] F-Secure: Security Information Center, Virus descriptions of viruses in the wild, http://www.f-secure.com/virus-info/wild.shtml, 2002-04-29.

[22] Addison-Wesley Publishing, James F. Kurose, Keith W. Ross, Computer networking: A top-down approach featuring the Internet, 2000.

[23] Sun Microsystems, Tim Lindholm, Frank Yellin, Java Virtual Machine Specification, http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html, 2002-04-29.

[24] Sun Microsystems, Java 2 Platform Micro Edition (J2ME) technology for creating mobile devices, http://java.sun.com/products/cldc/wp/KVMwp.pdf, 2002-04-29.

[25] Sun Microsystems, Over The Air User Initiated Provisioning, recommended practice for the Mobile Information Device Profile, http://java.sun.com/products/midp/OTAProvisioning-1.0.pdf, 2002-04-29.

[26] Sun Microsystems, J2ME CLDC API 1.0, http://java.sun.com/j2me/docs/zip/cldcapi.zip, 2002-04-29.

[27] Sun Microsystems, J2ME MIDP API 1.0, http://jcp.org/aboutJava/communityprocess/final/jsr037/index.html, 2002-04-29.

[28] Apress, Jonathan Knudsen, Wireless Java: Developing with Java 2 Micro Edition, 2001.

[29] Sun Microsystems, The source for Java technology, http://java.sun.com, 2002-04-29.

[30] Sun Microsystems, Java 2 Platform Micro Edition, Wireless Toolkit, http://java.sun.com/products/j2mewtoolkit, 2002-04-29.

[31] TechnoLogismiki, Hackman, http://www.technologismiki.com/hackman, 2002-04-29.

[32] Adnome, XML - grunder för systemutvecklare, http://www.adnome.se/utbildning/kursdetalj_print.asp?KursID=63, 2002-05-02.

[33] Reqwireless, Reqwireless WebViewer readme, http://www.reqwireless.com/WebViewer-1.0/README.html, 2002-05-03.

[34] Cellular news, Siemens SL45i, http://www.cellular-news.com/cellphones/Siemens_SL45i.shtml, 2002-05-03.

[35] JavaMobiles, Siemens SL45i, http://www.javamobiles.com/siemens/sl45i, 2002-05-03.

[36] Alan Kaminsky, Attack Code in J2ME CLDC, http://www.cs.rit.edu/~ark/lectures/sec/04_02_02.html, 2002-05-03.

[37] Siemens, Mobile Interface Device Profile (MIDP) and its Siemens extensions, http://www2.siemens.fi/developers.jsp, 2002-05-03.

[38] Sun Microsystems, Java 2 Platform, Enterprise Edition, v1.3.1, API Specification, http://java.sun.com/j2ee/sdk_1.3/techdocs/api, 2002-05-06.

[39] WinZip Computing, Inc. WinZip, http://www.winzip.com, 2002-05-08.

[40] Microsoft Corp., Internet Explorer home page, http://www.microsoft.com/windows/ie/default.asp, 2002-05-13.

A Dokumentation av tester

A.1 Åtkomst till andra programsviters RMS

Källkoden till applikationen JRMSApplication finns på den bifogade CD-skivan i följande fil:

- `Bilaga A\A.1\JRMSApplication.java`

A.2 Lura en användare med grafiskt gränssnitt och HTTP-uppkoppling

Källkoden till JFakePIN och JServer finns på den bifogade CD-skivan i följande filer: (Observera att JServer är implementerad med funktionalitet från J2EE.) [38]

- `Bilaga A\A.2\JFakePIN.java`
- `Bilaga A\A.2\JServer.java`

A.3 Applikation utan innehåll

Källkoden till applikationen JEmpty finns på den bifogade CD-skivan i följande fil:

- `Bilaga A\A.3\JEmpty.java`

A.4 Belastning av enheten

Källkoden till de fyra testfallen av applikationen JCB finns på den bifogade CD-skivan i följande filer:

- `Bilaga A\A.4\Testfall 1\JCB.java`
- `Bilaga A\A.4\Testfall 2\JCB.java`
- `Bilaga A\A.4\Testfall 3\JCB.java`
- `Bilaga A\A.4\Testfall 4\JCB.java`

A.5 Åtkomst till andra programsviters RMS med oavslutade trådar

Källkod till applikationen JCB finns på den bifogade CD-skivan i följande fil:

- `Bilaga A\A.5\JCB.java`

A.6 Injicera en skadlig MIDlet i en redan installerad programsvit

Källkoden till applikationen JRMSApplication finns på den bifogade CD-skivan i följande fil:

- `Bilaga A\A.6\JRMSApplication.java`

A.7 Versionsuppgradering till ett program med skadlig kod

Källkoden till applikationen JPassword version 1.0 och version 1.1 finns på den bifogade CD-skivan i följande filer:

- `Bilaga A\A.7\Version 1.0\JPassword.java`
- `Bilaga A\A.7\Version 1.1\JPassword.java`

JAR- och JAD-filerna till version 1.0 och version 1.1 av applikationen JPassword finns på den bifogade CD-skivan i följande filer:

- `Bilaga A\A.7\Version 1.0\JPassword.jar`
- `Bilaga A\A.7\Version 1.0\JPassword.jad`
- `Bilaga A\A.7\Version 1.1\JPassword.jar`
- `Bilaga A\A.7\Version 1.1\JPassword.jad`

A.8 Klassfilsmodifiering

I följande filer på den bifogade CD-skivan finns källkoden till klassen JSimple (`JSimple.java`), klassfilen som kompilatorn genererat (`JSimple.class`), en textfil från programmet Hackman innehållande klassfilen i hexadecimal form (`JSimple.class.txt`), samt ett XML-dokument som vår egenutvecklade applikation ClassToXML genererat (`JSimple.class.xml`):

- `Bilaga A\A.8\JSimple\JSimple.java`
- `Bilaga A\A.8\JSimple\JSimple.class`
- `Bilaga A\A.8\JSimple\JSimple.class.txt`
- `Bilaga A\A.8\JSimple\JSimple.class.xml`

I samtliga testfall i avsnitt 9.8 är det klassen JSimple som modifierats. XML-dokumentet är till för att lättare leta upp en viss struktur i klassfilen och få reda på vilken position den finns på i klassfilen. På den bifogade CD-skivan finns de modifierade klassfilerna, textfiler från Hackman och, i de fall det var möjligt att generera, XML-dokument från ClassToXML. Filerna för testfall 1.1 ligger i mappen `Bilaga A\A.8\Testfall 1.1\`, filerna för testfall 1.2 i mappen `Bilaga A\A.8\Testfall 1.2\` och så vidare.

Tabell A.1 visar var i klassfilen en ändring skett för respektive testfall. Kolumnen Testfall visar vilket testfall det är fråga om och kolumnen Offset visar vilken position den första byten har i klassfilen. Kolumnen Ursprungligt värde anger vad det fanns för värde på positionen innan vi modifierade det till det värde som är angivet i kolumnen Nytt värde. Samtliga värden är hexadecimala.

Testfall	Offset	Ursprungligt värde	Nytt värde
1.1	0000:0009	0077	0078
1.2	0000:0651	0006	0003
1.3	0000:0651	0006	0009
1.4	0000:0651	0006	FFFF
2.1	0000:06A7	06	0D
2.2	0000:06A7	06	CA
2.3	0000:06A7	06	E5
3.1	0000:05D9	0021	0020
3.2	0000:05D7	0020	0021
4.1	0000:075F	0003	0004
4.2	0000:075F	0003	0002
4.3	*Här tog vi bort ett element från attributet Stackmap*		
5	0000:0007	032D	0000

Tabell A.1: Lista över ändrade bytekoder i testfallen i avsnitt 9.8

I testfall 4.3 ändrade vi samma bytes som i testfall 4.2 plus att vi tog bort ett av de element som fanns i attributet Stackmap.

A.9 Lura användaren med telefonifunktioner i Siemens SL45i

Källkoden till applikationen FreeSMS finns på den bifogade CD-skivan i följande fil:

- `Bilaga A\A.9\FreeSMS.java`

A.10 Avlyssna inkommande SMS

Källkoden till applikationen JSMSListener finns på den bifogade CD-skivan i följande fil:

- `Bilaga A\A.10\JSMSListener.java`

B Analys av klassfil med ClassToXML

B.1 Inledning

I denna bilaga illustrerar vi hur vårt egenutvecklade verktyg ClassToXML kan användas för att underlätta analys och modifiering av en klassfil. Som exempelprogram för en enkel analys av en klassfil väljer vi samma MIDlet som användes i testerna för klassfilsmanipulering i avsnitt 9.8 och bilaga A.8. JSimple är en enkel MIDlet, men innehåller de element som gör den till en fungerande applikation. Vi skapar analysmaterial av JSimple både före och efter att den preverifierats för att studera den förändring som sker vid preverifieringen.

B.2 ClassToXML

De sex källkodsfilerna till ClassToXML finns i mappen `Bilaga B\B.2\` på den bifogade CD-skivan. I samma mapp finns även motsvarande klassfiler. Klassfilen som skall startas upp för at köra ClassToXML heter `ClassToXML.class`. Verktyget tar namnet på en klassfil som inargument och skapar ett strukturerat XML-dokument av innehållet i klassfilen. I XML-dokumentet är det angivet vilken struktur i klassfilen varje enskild byte tillhör och vilken position densamma har i klassfilen.

B.3 JSimple

Källkoden till applikationen JSimple finns på den bifogade CD-skivan i följande fil:

- `Bilaga B\B.3\JSimple.java`

B.4 Tillvägagångssätt

Källkoden till JSimple kompileras först till en icke preverifierad klassfil med kommandot Build i Wireless Toolkit [30]. Den kompilator som används är faktiskt samma som den som används för applikationer skrivna för J2SE. För att få en preverifierad klassfil använder vi sedan kommandot Package. Detta startar ett antal processer för att förbereda en överföring från den PC utvecklingen sker på till en mobil terminal med stöd för MIDP. Bland annat

skapas en JAR-fil som utgör programsviten och en JAD-fil som innehåller information om programsviten. I detta fall kommer endast en MIDlet, JSimple, att finnas i programsviten.

Efter paketeringen finns det två klassfiler: En preverifierad och en icke preverifierad. Nu är allt förarbete som krävs för analysen klar och det är dags att undersöka klassfilerna med lämpliga verktyg. När omvandlingen mellan källkodsfil och klassfil sker så försvinner även läsligheten. Genom att öppna klassfilen i programmet Hackman [31] som är en hexredigerare kan följande inledande rader utläsas: (En textfil från Hackman innehållande hela klassfilens information finns i bilaga B.5.)

```
CA FE BA BE 00 03 00 2D 00 77 0A 00 21 00 3F 08
00 40 09 00 20 00 41 08 00 42 09 00 20 00 43 08
00 44 09 00 20 00 45 08 00 46 09 00 20 00 47 09
00 20 00 48 09 00 20 00 49 09 00 20 00 4A 0A 00
4B 00 4C 09 00 20 00 4D 07 00 4E 0A 00 0F 00 4F
...
```

Datan som visas innehåller information om klassfilen men det är svårt att tolka den. Det enda som en människa direkt kan se är ordet CAFEBABE. Detta är klassfilssignaturen som anger att det rör sig om en klassfil i Java. Sedan följer en mängd mer svårtolkad data. För att kunna gå vidare med analysen måste kunskap om hur en klassfil byggs upp inhämtas. Strukturen finns förklarad i JVMS men för att snabba upp processen med att tolka hexadecimala värden använder vi verktyget ClassToXML.

B.5 Utdata från Hackman och ClassToXML

Vi har bifogat fyra filer med utdata: Två textfiler från programmet Hackman [31] innehållande klassfilen i hexadecimal form före och efter preverifiering, och två XML-dokument från ClassToXML, också dessa genererade före och efter preverifiering. Utdatan finns i mappen `Bilaga B\B.5\` på den bifogade CD-skivan.

Nämnvärda skillnader som kan ses i XML-dokumenten är förekomster av attributet Stackmap i det XML-dokument som skapades efter preverifiering. Sist i listan över konstanter finns en post innehållande strängen ”Stackmap” och i den sista metoden i listan över metoder finns i attributet Code ett underattribut med en bakåtreferens till denna post.

C Förkortningslista med förklaringar

Card VM	Card Virtual Machine; en virtuell maskin som används för applikationer skrivna i Java för smart cards
CDC	Connected Device Configuration; konfiguration som används i samband med enheter som har snäppet högre prestanda än de enheter som implementerar CLDC
CLDC	Connected Limited Device Configuration; konfiguration som används i samband med små resurssnåla enheter
CVM	Compact Virtual Machine; virtuell maskin avsedd att köras på en enhet med snäppet högre prestanda än enheter som kör KVM
FTP	File Transfer Protocol; nätverksprotokoll för filöverföring
HTTP	Hyper Text Transfer Protocol; nätverksprotokoll
IP	Internet Protocol; standardprotokoll för sändning av data över Internet
J2EE	Java 2 Enterprise Edition; javastandard mer inriktad på stora resurskrävande tillämpningar som exempelvis serverapplikationer
J2ME	Java 2 Micro Edition; den del av javaplattformen som riktar sig till små enheter med begränsade resurser
J2SE	Java 2 Standard Edition; den kanske mest vanliga javastandarden, främst använd i samband med applets på vanliga persondatorer
JAD	Java Application Descriptor; en fil innehållande information angående den programsvit som JAD-filen är kopplad till
JAM	Java Application Manager; benämningen på den inbyggda applikation som har till uppgift att ladda ned och installera programsviter samt starta upp MIDlets
JAR	Java Archive; ett komprimerat bibliotek av en eller flera filer tillhörande en eller flera javaapplikationer
JIT	Just In Time; kompilering efter behov
JVM	Java Virtual Machine; den ursprungliga virtuella maskinen
JVMS	Java Virtual Machine Specification; ett dokument som tar upp hur standardimplementationen av den virtuella maskinen fungerar och beskriver klassfilsformatet

kB, MB	Kilobyte, Megabyte; 1 MB = 1 024 kB = 1 048 576 byte, byte är enheten för lagringsutrymme
KVM	Kilo Virtual Machine; en liten, kompakt virtuell maskin för enheter som implementerar CLDC och MIDP
MIDI	Musical Instrument Digital Interface; en serie digitala instruktioner till en ljudenhet
MIDP	Mobile Information Device Profile; profil för resurssnåla mobila enheter som exempelvis mobiltelefoner
OTA	Over The Air User Initiated Provisioning; ett begrepp som används när nedladdning av applikationer eller data till exempelvis en mobiltelefon sker trådlöst
PDA	Personal Digital Assistant; en enhet tänkt att fungera som en digital filofax men även som en handburen dator
PDA Profile	Personal Digital Assistant Profile; profil för PDAer
PIN	Personal Identification Number; en kod, eller ett lösenord, som utgörs av ett nummer
RMS	Record Management System; en komponent i MIDP som möjliggör lagring av beständig data
RSA	Första bokstaven i efternamnen på de tre utvecklarna av RSA: Ronald Rivest, Adi Shamir, Leonard Adleman; krypteringsstandard
SMS	Short Message Service; ett upp till 160 tecken långt textmeddelande
SSH	Secure Shell; nätverksprotokoll för säker överföring av data
SSL	Secure Sockets Layer; nätverksprotokoll för säker överföring av data
TCP	Transmission Control Protocol; standardprotokoll för sändning av data över Internet
WAP	Wireless Application Protocol; samling av nätverksprotokoll för trådlös kommunikation
WML	Wireless Markup Language; språk för uppmärkning av data
XML	eXtensible Markup Language; språk för uppmärkning av data