HRVATSKA AKADEMSKA I ISTRAŽIVAČKA MREŽA
CROATIAN ACADEMIC AND RESEARCH NETWORK

Sigurnost Java aplikacija

CCERT-PUBDOC-2006-07-163

CARNet CERT u suradnji s LS&S

Sigurnosni problemi u računalnim programima i operativnim sustavima područje je na kojem CARNet CERT kontinuirano radi.

Rezultat toga rada ovaj je dokument, koji je nastao suradnjom CARNet CERT-a i LS&S-a, a za koji se nadamo se da će Vam koristiti u poboljšanju sigurnosti Vašeg sustava.

CARNet CERT, www.cert.hr - nacionalno središte za **sigurnost računalnih mreža** i sustava.

LS&S, www.lss.hr - laboratorij za sustave i signale pri Zavodu za elektroničke sustave i obradbu informacija Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

Sadržaj

1. Uvod

U prosincu 1990. godine Sun Microsystems je započeo razvoj tzv. „Oak“ tehnologije zamišljene kao platforme i programskog jezika, u želji da ponudi alternativu C/C++ programskim jezicima. Rezultati projekta bili su prezentirani 1992. godine kada je javnosti predstavljen osobni pomoćnik razvojnog imena Star7 koji je imao grafičko sučelje i animiranog pomoćnika zvanog „Duke“ kasnije odabranog za zaštitni simbol i maskotu projekta. Sve do 1994. godine projekt nije davao zadovoljavajuće rezultate pa se pojavila potreba za promjenom ciljanog tržišta. U lipnju iste godine dogovoreno je da se ciljano tržište „Oak“ projekta prebaci s dlanovnika i kabelske televizije na sve popularniji Internet. Nakon što je otkriveno kako je postojeće ime projekta „Oak“ već zaštićeno na tržištu, u jednom od mnogobrojnih posjeta cafe baru, razvojno osoblje se odlučilo za ime Java koje po njihovim navodima nema posebno značenje. Kako bi ostavili spomen na cafe bar, odlučeno je da 4 okteta kojim počinju sve klase bude u heksadekatskom sustavu 0xCAFE.

Glavna prednost Java programskog jezika nad postojećim jezicima su prenosivost programskog koda koja se propagira preko svoje filozofije „Napiši jednom, pokreni svuda“ (eng. *Write once, run anywhere*). Navedena metodologija programiranja omogućuje programerima da velike količine koda mogu koristiti u svim svojim projektima, na različitim operacijskim sustavima i tako smanjiti vrijeme potrebno za realizaciju ideja. Način na koji je Java ostvarila te mogućnosti je implementacija svih osobina objektno orijentiranog jezika, kao što su klase, metode, nasljeđivanje, enkapsulacija, apstrakcija, itd. Kako bi dodatno olakšali programiranje i smanjili potencijalne sigurnosne nedostatke izbačeni su pokazivači i bilo kakvo eksplicitno manipuliranje memorijom koje bi moglo rezultirati nestabilnošću koda. Jednostavan memorijski model u kojem su svi objekti sadržani na gomili (eng. *heap*) ne ostavlja puno prostora za pogrešku, stoga memorijska organizacija programa ne zahtijeva puno posla, utoliko manje što je oslobađanje memorije riješeno automatski preko kolektora koji ju oslobodi i stavi na listu slobodnog memorijskog prostora nakon što ju nitko više ne koristi.

Od modernog programskog koda se traži kompatibilnost tj. mogućnost da se isti program može izvršavati na više računalnih arhitektura i operativnih sustava. Postoji više načina kako riješiti navedeni zahtjev, svaki sa svojim prednostima i nedostacima. Sun je riješio ovaj problem na zanimljiv način žrtvujući brzinu izvođenja programa u korist programera i pojednostavljenog programiranja. *Java Virtual Machine* (JVM) je temelj Java platforme zadužen za neovisnost programa o sklopovlju i operativnom sustavu. JVM predstavlja apstraktno računalo koje kao i svako pravo ima svoj skup naredbi i mogućnost upravljanja memorijom.

U nastavku dokumenta opisane su osnove Java platforme, njezini sigurnosni nedostatci i preporuke kako se zaštititi od nedostataka te poboljšati kvalitetu programskog koda.

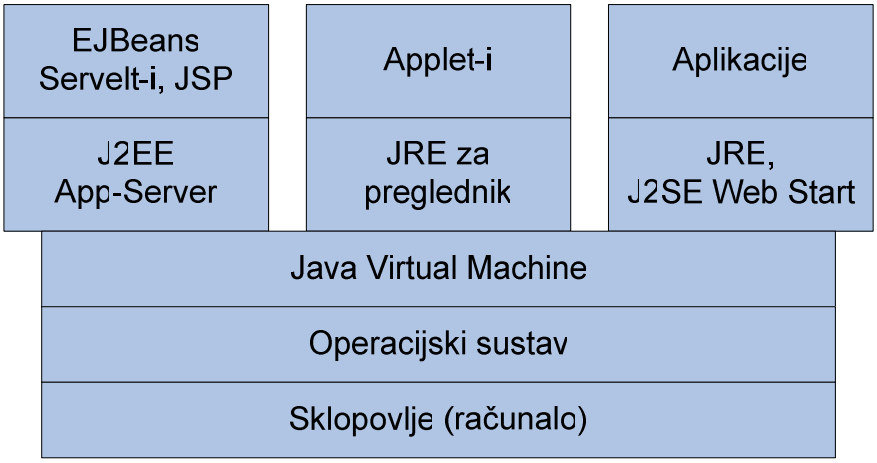
Slika 1: Duke – maskota Java projekta

2. Java platforma

Platforma je naziv za kompleksnu aplikaciju koja omogućuje korisniku nadograđivanje po svojim željama. Primjeri platformi su operacijski sustavi kao Microsoft Windows, Macintosh, UNIX, Linux i mnogi drugi koji omogućuju krajnjem korisniku interakciju sa sklopovljem računala. Kod navedenih platformi, aplikacije moraju biti prevedene (eng. *compiled*) odvojeno za svaku platformu kako bi se mogle pokrenuti. Razlog tome je ovisnost aplikacija o tipu izvršne datoteke i instrukcijskom skupu procesora.

Java platforma je novi tip aplikacijske platforme koja omogućava pokretanje interaktivnih *applet*-a (programi koji se izvršavaju u web preglednicima korisnika), aplikacija i *servlet*-a (programi koji se izvršavaju na poslužitelju) na mrežnom računalu. Ono što razlikuje Java platformu od ostalih je to što se ona nalazi iznad navedenih platformi i omogućuje izvođenje *bytecode*-a, programskog koda srednje razine namijenjenog pokretanju od strane virtualnog računala (eng. VM - *Virtual Machine*). Stoga *bytecode* nije ovisan o platformi na kojoj se. Program napisan u Java jeziku se prevodi u *bytecode* datoteku koja se onda može izvršiti svuda gdje postoji *Java Virtual Machine,* tj. iznad bilo kojeg operativnog sustava ili arhitekture. Iako svaka od platformi na koju se Java nadograđuje ima svoju verziju VM-a, postoji jedinstvena specifikacija *bytecode* datoteke. Zbog te jedinstvenosti, Java platforma može omogućiti jednoznačno sučelje za programiranje *applet*-a i aplikacija na bilo kojem sklopovlju. Iz navedenog je moguće zaključiti kako je Java platforma idealna za Internet gdje kod pokretanja aplikacija nije moguće znati kakvo računalo pojedini korisnik posjeduje već se samo želi osigurati ispravno funkcioniranje aplikacije u svim uvjetima.

Java platforma se sastoji od dva osnovna dijela. Prvi je *Java Virtual Machine* zadužen za pokretanje *bytecode*-a, a drugi je Java API (eng. *Application Program Interface*) koji omogućuje korištenje gotovih funkcija i klasa.



Slika 2: Java platforma

Kao što je na prethodnoj slici prikazano, zahvaljujući *Java Virtual Machine*, moguće je pokretanje različitih oblika programa:

- aplikacije – izvršavaju se na računalu na kojem je i JVM, a za njihovo ispravno funkcioniranje potrebni su JRE (eng. *Java Runtime Environment*) potreban za izvršavanje koda te J2SE (eng. *Java 2 Platform Standard Edition*) za razvoj i nadogradnju aplikacija,
- *applet*-i – programi koji se izvršavaju u web pregledniku, a za njihovo pokretanje potreban je JRE za preglednike, te
- *servlet*-i u što su uključeni i JSP (eng. *JavaServer Pages*) dinamičke web stranice – programi namijenjeni izvršavanju na poslužitelju, a za njihovo izvođenje potreban je J2EE (eng. *Java 2 Platform Enterprise Edition*) aplikacijski poslužitelj.

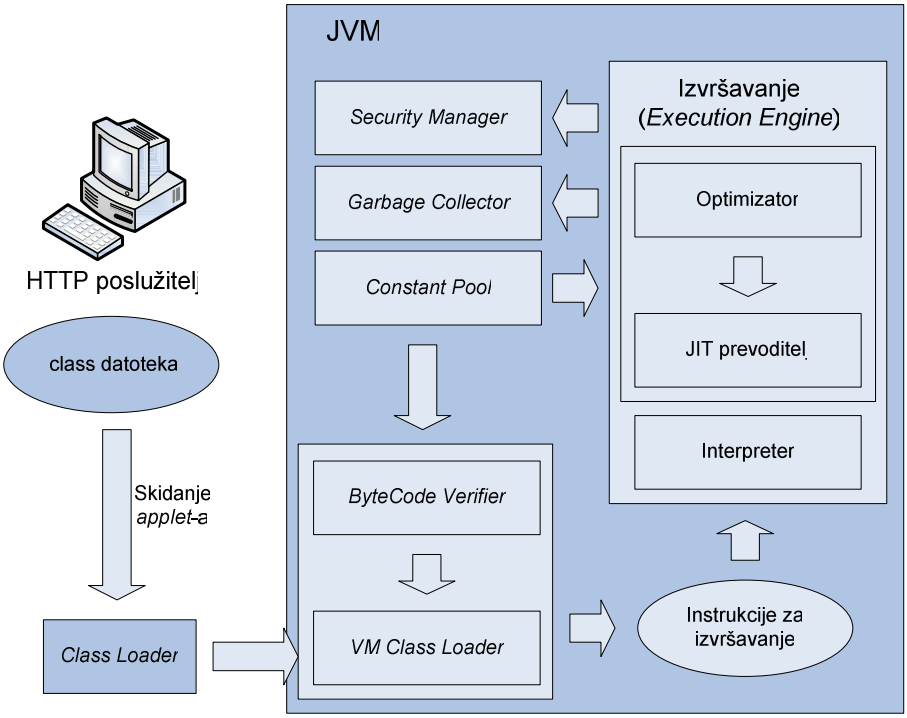
2.1. *Java Virtual Machine*

Java Virtual Machine (JVM) je okruženje za izvršavanje *bytecode*-a koje se sastoji od nekoliko komponenti koje osiguravaju arhitekturnu neovisnost Java jezika, sigurnost i pokretnost. JVM je apstraktno računalo koje može učitavati i izvršavati Java programe. S aspekta sigurnosti, najzanimljiviji su Java Class datotečni format, Java *bytecode* instrukcijski set i JVM *bytecode Verifier*.

Java Class datotečni format definira načine zapisa Java klasa (programskog koda) u zapis nezavisan od platforme. Pojedinačne Class datoteke uvijek sadržavaju zapis jedne Java klase. Osim osnovnih karakteristika i informacija o klasi koja je spremljena unutar Class datoteke, u zapisu se uvijek mogu pronaći i informacije o dozvolama, polja koja su definirana i sučelja koja klasa ostvaruje. Programski kod svake metode iz klase je također spremljen unutar Class datoteke. Ovo je ostvareno korištenjem odgovarajućih programskih atributa, koji sadrže Java *bytecode* instrukcije neke metode. Također, unutar zapisa je moguće saznati informacije o korištenju stoga (eng. *stack*), lokalnih varijabli, broju registara i ostale informacije koje mogu pomoći JVM-u da što brže prevede zapis u izvršni kod. Sigurnosni mehanizam JVM-a je zadužen da provjeri jesu li svi podaci o klasama unutar Class datoteke ispravno zapisani.
Java *bytecode* je programski jezik koji JVM prevodi i interpretira u izvršni kod. To je zapis Java programskog jezika nakon što je program prošao kompilaciju. Kako bi JVM mogao što brže izvršiti zadani zapis, potrebno je da on bude što jednostavnijeg oblika i što bliži standardnom jeziku procesora. Osim logičkih, aritmetičkih, registarskih i stogovnih instrukcija koje su slične procesorskim, postoje i Java specifične instrukcije koje služe kreiranju, manipuliranju i pristupanju podacima unutar klasa. Interpretiranje *bytecode*-a se može prikazati pomoću petlje koja linearno dohvaća instrukcije te ukoliko postoji zapis te instrukcije, odradi zadani posao te nastavlja sve dok ne bude više instrukcija. Pseudo kod bi izgledao ovako:

```
ponavljaj {
  dohvati instrukciju;
  ako (instrukcija) dohvati izvršni kod;
  {   odradi zadano;  }
} dok (postoji još instrukcija);
```

Važna zadaća JVM-a je zaštita računala od potencijalno zlonamjernog ili neispravnog Java koda. Dio JVM-a zadužen za provjeru sintakse i logičke cjeline programa je *Bytecode Verifier*. On sadrži skup pravila koje program mora zadovoljavati kako bi se dozvolilo njegovo pokretanje. Pravila se odnose na ispravnost strukture datoteke koja se pokreće, provjere jesu li zadovoljeni svi zahtjevi za programskim bibliotekama, provjere o konfliktima atributa te ispravnost zapisa instrukcija.
Bytecode Verifier nije jedini sigurnosni mehanizam JVM-a koji štiti Java platformu od sigurnosnih napada. Ostale komponente jednako su važne i zajedno tvore mrežu koja uklanja sigurnosne propuste i pokušaje njihova iskorištavanja. Detaljan opis koraka koji se poduzimaju kako bi se zaštitila Java platforma biti će opisani na primjeru pokretanja Java *applet*-a.
Prije nego što se može pokrenuti Java *applet* unutar JVM-a, on mora proći proces kompilacije u `.class` datotečni format. Ukoliko programski kod koristi više od jedne `.class` datoteke, one mogu biti proizvoljno pakirane u `.jar` ili `.zip` arhivu. Ako korisnik Web preglednika posjeti internet stranicu koja koristi `<APPLET>` prefiks, nova instanca JVM-a će biti pokrenuta od strane preglednika. Ukoliko je instanca uspješno pokrenuta, JVM započinje proces dobavljanja *applet*-a. Prilikom toga se poziva *Class Loader* objekt koji uspostavlja vezu s poslužiteljem, navedenim unutar `CODE` atributa nakon `<APPLET>` prefiksa, kako bi dohvatio `.class` datoteku unutar koje se nalazi definicija *applet*-a koji se učitava. Dobivena definicija *applet* klase biva registrirana unutar JVM-a pozivom jedne od osnovnih metoda VM *Class Loader* objekta. Prije nego nova klasa može biti registrirana, mora proći odgovarajuće provjere *Class Loader*-a koji štiti od nekih vrsta napada kao što su lažiranje ili redefiniranje klase. Ukoliko su provjere uspješno završene, *Class Loader* zove JVM metodu kako bi odredila zadanu klasu unutar *appleta*. Prilikom tog poziva JVM poziva *Bytecode Verifier* kako bi izveo novi skup provjera koje moraju biti zadovoljene kako bi se uspješno nastavio proces registracije unutar Java izvršnog (eng. *runtime*) procesa.



Slika 3: Grafički prikaz puta *applet*-a prilikom učitavanja

Sljedeći korak je kreiranje nove instance objekta unutar Java *runtime* procesa i pokretanja *start()* metode novo kreirane *applet* klase. Poziv bilo kojoj Java objekt metodi zahtijeva pokretanje odgovarajućeg Java *bytecode* instrukcijskog niza, za koji je zadužena komponenta *Execution Engine*. Ova komponenta je najčešće implementirana jednim od dva načina: kao interpreter ili prevoditelj. U slučaju interpreterske implementacije, programski kod se izvršava čitanjem *bytecode*-a instrukciju po instrukciju i izvršava linearno, za razliku od prevoditeljske implementacije kada se prvo prevede Java *bytecode* u izvršni kod platforme na kojoj se nalazi Java platforma te se pokrene kao obična izvršna datoteka. Izvršavanjem koda na bilo koji od dva moguća načina, programski kod često poziva Java komponente kao *Garbage Collector* i *Security Manager*. Prva je zadužena za organizaciju memorije, te poslove poput rezervacije i oslobađanja memorijskog prostora. A *Security Manager* mehanizam se poziva od strane JVM-a ukoliko postoji zahtjev za odvijanjem akcije koja zahtijeva posebne ovlasti. Njegova uloga je provjeravanje da li takva akcija može povrijediti standardne sigurnosne postavke *applet*-a i njegovog zaštitnog modela.

Za vrijeme izvršavanja Java koda zapisuju se različite informacijske Java *runtime* strukture. Ovo se odnosi na imena funkcija, konstantne vrijednosti te opise klasa i metoda. Sve ove podatke organizira i čuva JVM u specijalno za to dizajniranom *Constant Pool* području. Ove podatke koriste neke od već navedenih Java komponenti kao što su *Bytecode Verifier* i *Execution Engine*.

Opisan proces dobavljanja i pokretanja *applet*-a ukratko predstavlja putovanje Java klase unutar JVM-a. Iz navedenog je potrebno primijetiti kako su *Class Loader*, *Bytecode Verifier* i *Security Manager* komponente JVM-a koje na sebi imaju važnu ulogu očuvanja sigurnosti Java platforme i iz tog razloga predstavljaju sigurnosno rizične komponente.

2.2. Java programski jezik

Java programski jezik sintaktički jako sliči C++ programskom jeziku. Zamišljen je kao objektno orijentiran jezik bez podrške za pokazivače i direktne manipulacije memorijom. Veliki broj gotovih klasa za često korištene operacije uvelike olakšava programiranje.

Slijedi jednostavan Java programski kod koji ispisuje zadane riječi pretvarajući mala u velika slova:

```
public class Java_primjer
{
        public static void main(String[] args)
        {
```

```
                String[] String1 = {"Ovo","je","primjer","java","koda"};
                for (int i = 0; i < String1.length; i++)
                {
                        new Java_primjer(String1[i]);
                }
        }
        public Java_primjer(String a)
        {
                System.out.println(a.toUpperCase());
        }
}
```

Kako bi se pokrenuo program prvo ga je potrebno pomoću Java prevoditelja pretvoriti u `.class` datotečni format.

```
$ javac Java_primjer.java
```

Ukoliko je prevođenje uspješno izvršeno, neće se pojaviti upozorenja o neispravnosti koda te će biti kreirana `Java_primjer.class` datoteka koja u sebi sadrži *bytecode*. Program se izvršava pokretanjem *Java Virtual Manager*-a i navođenjem imena klase koju je potrebno pokrenuti.

```
$ java Java_primjer
OVO
JE
PRIMJER
JAVA
KODA
```

Bytecode reprezentaciju izvršnog programa moguće je vidjeti korištenjem Java *dissasembler*-a:

```
$ javap -c Java_primjer
Compiled from "Java_primjer.java"
public class Java_primjer extends java.lang.Object{
public static void main(java.lang.String[]);
  Code:
   0:   iconst_5
   1:   anewarray       #1; //class String
   4:   dup
   5:   iconst_0
   6:   ldc     #2; //String Ovo
   8:   aastore
   9:   dup
   10:  iconst_1
   11:  ldc     #3; //String je
   13:  aastore
   14:  dup
   15:  iconst_2
   16:  ldc     #4; //String primjer
   18:  aastore
   19:  dup
   20:  iconst_3
   21:  ldc     #5; //String java
   23:  aastore
   24:  dup
   25:  iconst_4
   26:  ldc     #6; //String koda
   28:  aastore
   29:  astore_1
   30:  iconst_0
   31:  istore_2
   32:  iload_2
```

```
   33:  aload_1
   34:  arraylength
   35:  if_icmpge       55
   38:  new     #7; //class Java_primjer
   41:  dup
   42:  aload_1
   43:  iload_2
   44:  aaload
   45:  invokespecial   #8; //Method "<init>":(Ljava/lang/String;)V
   48:  pop
   49:  iinc    2, 1
   52:  goto    32
   55:  return
public Java_primjer(java.lang.String);
  Code:
   0:   aload_0
   1:   invokespecial   #9; //Method java/lang/Object."<init>":()V
   4:   getstatic       #10; //Field
                             java/lang/System.out:Ljava/io/PrintStream;
   7:   aload_1
   8:   invokevirtual   #11; //Method
                          java/lang/String.toUpperCase:()Ljava/lang/String;
   11:  invokevirtual   #12; //Method
                          java/io/PrintStream.println:(Ljava/lang/String;)V
   14:  return

}
```

Iako je Sun napravio puno toga kako bi olakšao programiranje u Java programskom jeziku, bez dobrog razumijevanja programskog jezika, njegove sintakse i specifičnosti, programeri nikad neće primijetiti i ukloniti propuste. Uz dobro razumijevanje strukture zadane platforme, zlonamjerni napadač može iskoristiti naizgled neprobojne zaštite sve dok programer ostavlja malo mjesta za izvođenje napada. Stoga naizgled jednostavni programi pisani u Java programskom jeziku mogu dati neočekivane rezultate, što je vidljivo iz narednih nekoliko kratkih programa koji pokazuju koliko je mala razlika između ispravnog i neispravnog programskog koda.
Kao prvi primjer odabran je programski kod koji ispisuje parnost cijelog broja:

```
public static boolean neparan(int i) {
    return i % 2 == 1;
}
```

Navedeni primjer je jedan izuzetno jednostavan program koji bi trebao uvijek raditi. Ipak, ako se kod malo bolje promotri, moguće je primijetiti da je ulazni parametar funkcije varijabla tipa *integer*. Ukoliko nije eksplicitno navedeno je li varijabla pozitivna ili negativna, prevoditelj pretpostavlja kako je varijabla predznačnog tipa. Kao posljedica te pretpostavke navedeni kod će u 25% slučajeva vratiti krivi rezultat prilikom modulo 2 operacije nad negativnim brojem jer je ostatak -1, a ne 1 kao kod pozitivnih brojeva. Rješenje ovog problema je više nego jednostavno - ne provjerava se ukoliko je ostatak 1 već 0:

```
public static boolean neparan(int i) {
    return i % 2 != 0;
}
```

Slijedeći primjer opisuje nedostatak razumijevanja naredbi uspoređivanja:

```
public class usporedba {
    public static void main(String[] args) {
        char a = 'X';
        int i = 0;
        System.out.print(true  ? a : 0);
```

```
        System.out.print(false ? i : a);
    }
}
```

Program se sastoji od dvije varijable, jedan cijeli broj (0) i jedan znak (X). Pitanje je što će se ispisati dvije funkcije za ispis? Iako bi većina programera pretpostavila „XX“, to nažalost nije točan odgovor. Zbog kompliciranih uvjeta implicitnog pretpostavljanja ispisivanja brojeva, program će ispisati „X88“. Naime, znak „X“ je u dekadskom ASCII sustavu zapisan kao 88. Prilikom drugog ispisivanja, ponuđen je ispis broja i znaka. Pošto je broj prvi naveden, prilikom ispisa drugi znak se ispisuje poput broja.
Prezentirani programi predstavljaju samo mali broj često krivo tumačenih izraza za koje se smatra da se podrazumijevaju, a stvarnost je katkad puno drugačija od očekivanoga. Navedeni problemi ne predstavljaju ozbiljne sigurnosne propuste već programerske greške, ali ukoliko se takvim propustima ne prida dovoljno velika pozornost, oni mogu dovesti do ozbiljnih sigurnosnih propusta.

3. Sigurnost Java aplikacija

Java programski jezik, za razliku od C/C++ programskih jezika, nema direktan pristup memoriji i operacijskom sustavu. Zbog te činjenice sigurnosni propusti Java aplikacije se donekle razlikuju od onih prisutnih u jezicima koji su prevedeni za arhitekturu na kojoj se izvršavaju. Kako Java programski jezik ne podržava pokazivače i eksplicitno kopiranje memorije, veliki dio učestalih sigurnosnih propusta prisutnih unutar C/C++ programskih jezika i njima sličnima, nije prisutan. Iako Java nema navedene probleme, zbog svoje organizacijske arhitekture ona uvodi nekoliko novih sigurnosnih tipova propusta. Jedan od najvećih problema je onaj koji pogađa sigurnost *applet*-a koji se izvršavaju na računalu osobe koja pregledava web stranicu. Kako bi se spriječile potencijalne zlonamjerne aktivnosti, *applet* se nalazi u tzv. „*sandbox*“ okruženju koje mu ne dozvoljava pozivanje funkcija i klasa za pristup resursima računala na kojem se *applet* izvodi. Ukoliko programer napiše program poštujući sva pravila Java sigurnosnih mehanizama i slijedeći programersku etiku, još uvijek je moguće da program predstavlja sigurnosnu prijetnju za korisnika. To se događa u slučaju kada se koriste klase Java programskog jezika ili biblioteke iz nekih nepouzdanih izvora. Ukoliko korišteni kod sadrži sigurnosne propuste, ti propusti će postati sastavni dio naizgled sigurnog koda. Zbog ove činjenice, osim sigurnosnih propusta aplikacije, postoji potreba i za obraćanjem pozornosti na sigurnost koda koji se implicitno uključuje u razvijane aplikacije.

3.1. Propusti neprovjeravanja ulaznih parametara

Jedan od najčešćih razloga zbog kojeg dolazi do sigurnosnih nedostataka je nedovoljno razumijevanje programskog jezika i svih njegovih osobina. Unutar *Java SE Development Kit* 1.3.1 biblioteke, otkriveno je više sigurnosnih propusta, koji prilikom pozivanja određenih funkcija s NULL argumentima rezultiraju rušenjem Java platforme. Koliko je jednostavno iskoristiti navedeni propust može se vidjeti u sljedećih nekoliko linija koda:

```
package crashtest;
import sun.dc.pr.PathDasher;
public class CrashTest
{
    public CrashTest()
    {
        PathDasher d = new PathDasher(null);
    }

    public static void main(String args[])
    {
        CrashTest crashTest1 = new CrashTest();
    }
}
```

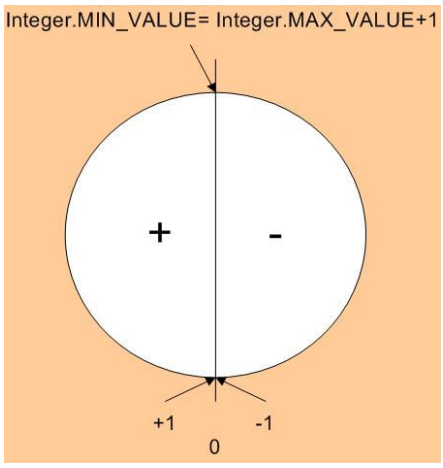
Rezultat pokretanja navedenog programa je rušenje Java platforme:

```
Unexpected Signal : EXCEPTION_ACCESS_VIOLATION occurred at PC=0x6d443081
Function name=JVM_DisableCompiler
Library=H:\java_test\JB6\jdk1.3.1\jre\bin\hotspot\jvm.dll

Current Java thread:
       at sun.dc.pr.PathDasher.cInitialize(Native Method)
       at sun.dc.pr.PathDasher.<init>(PathDasher.java:48)
       at crashtest.CrashTest.<init>(CrashTest.java:91)
       at crashtest.CrashTest.main(CrashTest.java:98)

Dynamic libraries:
0x00400000 - 0x00405000    H:\ java_test \JB6\jdk1.3.1\bin\javaw.exe
0x77F40000 - 0x77FF0000    C:\WINDOWS\System32\ntdll.dll
....
0x76BB0000 - 0x76BBB000    C:\WINDOWS\System32\PSAPI.DLL

Local Time = Sun Jun 23 14:00:38 2002
Elapsed Time = 0
#
# HotSpot Virtual Machine Error : EXCEPTION_ACCESS_VIOLATION
# Error ID : 4F530E43505002CC
# Please report this error at
# http://java.sun.com/cgi-bin/bugreport.cgi
#
# Java VM: Java HotSpot(TM) Client VM (1.3.1-b24 mixed mode)
#
# An error report file has been saved as hs_err_pid3888.log.
# Please refer to the file for further information.
#
```

Navedeni primjer ilustrira kako nepažnja programera i sigurnosni propusti unutar Java biblioteka mogu za posljedicu imati ozbiljan sigurnosni propust. Česta programerska greška je neprovjeravanje ulaznih parametara funkcije što može imati sigurnosne posljedice ako su funkcije koje se pozivaju dio sistemskog okruženja.

3.2. Propusti cjelobrojnog zaokruživanja

Još jedan učestali tip sigurnosnog propusta prisutan i kod C/C++ programskih jezika su propusti cjelobrojnog zaokruživanja varijabli. Ukoliko program ne posvećuje posebnu pozornost prilikom množenja, dijeljenja, zbrajanja i oduzimanja, mogu se dobiti neočekivani rezultati koji se kose sa zakonima matematike. Razlog tome je binarni prikaz vrijednosti broja unutar varijable, koji daje izgled kružnog oblika promjene vrijednosti. Ako se prijeđe maksimalna vrijednost varijable, prelazi se u najmanju negativnu vrijednost, a vrijedi i obratno.



Slika 4: Grafički prikaz promjene vrijednost varijable

Koliko je ozbiljan problem nepravilnog računanja sa cjelobrojnim varijablama najbolje pokazuje broj ranjivih Java funkcija i klasa. Unutar verzije JDK 1.4.1 biblioteke, koja sadrži funkcije vezane uz komprimiranje datoteka, više od 6 funkcija je ranjivo na opisani napad.
Ukoliko se pozove funkcija `update()` na način da prilikom zbrajanja zadnja dva argumenta dođe do prelaska najvećeg mogućeg pozitivnog broja pa se tako prijeđe u najmanji negativni, dolazi do rušenja JVM-a:

```
CRC32 c = new java.util.zip.CRC32 ();
c.update (new byte [0] ,4 ,Integer. MAX_VALUE -3);

H:\ > java CRCCrash
An unexpected exception has been detected in native code outside the VM.
Unexpected Signal : EXCEPTION_ACCESS_VIOLATION occurred at PC=0 x6D3220A4
Function= Java_java_util_zip_ZipEntry_initFields+0x288
Library=H:\java_test\1.4.1\01\jre\bin\zip.dll
Current Java thread :
at java.util.zip.CRC32.updateBytes(Native Method )
at java.util.zip.CRC32.update(CRC32.java:53)
at CRCCrash.main(CRCCrash.java :3)
Dynamic libraries:
0x00400000 - 0x00406000 h:\java_test\1.4.1\01\jre\bin\java.exe
[... lines omitted ...]
0x76BB0000 - 0x76BBB000 C:\WINDOWS\System32\PSAPI.DLL
Local Time = Mon Mar 17 14:57:47 2003
Elapsed Time = 3
#
# The exception above was detected in native code outside the VM
#
# Java VM : Java HotSpot(TM ) Client VM (1.4.1_01 -b01 mixed mode)
#
```

Ranjivi dio funkcije `update()` je nepotpuna usporedba ulaznih parametara koja neće generirati iznimku u nekim slučajevima kada bi trebala:

```
public void update(byte[] b, int off, int len)
{
    if (b == null)
    {
       throw new NullPointerException();
    }
    if (off < 0 || len < 0 || off + len > b.length)
    {
      throw new ArrayIndexOutOfBoundsException();
    }
    crc = updateBytes(crc, b, off, len);
}
```

Kako bi zaobišli provjeru vrijednosti indeksa kojoj je potrebno pristupiti, uvjet (`off + len > b.length`) ne smije biti zadovoljen, a u isto vrijeme bi trebao biti neispravan. Kako bi se postigla takva situacija, može se izbaciti zbroj (`off + len`) izvan granica pozitivnih brojeva i doći u negativno područje. Kako je negativan broj uvijek manji od nekog pozitivnog (uključujući nulu), program neće izbaciti iznimku, već će pozvati `updateBytes()` funkciju s neispravnim parametrima. Posljedica toga je rušenje JVM-a i moguće izvršavanje DoS napada.
Ispravka navedenog sigurnosnog propusta slična je ispravci prvog primjera neispravnog Java koda. Dovoljno je zamijeniti poziciju `len` varijable na desnu stranu znaka nejednakosti kako bi se izbjegao sigurnosni propust. Ispravljeni kod:

```
public void update(byte[] b, int off, int len)
{
       if (b == null)
```

```
        {
            throw new NullPointerException();
        }
        if (off < 0 || len < 0 || off > b.length - len)
        {
            throw new ArrayIndexOutOfBoundsException();
        }
        crc = updateBytes(crc, b, off, len);
}
```

3.3. Propusti vezani uz vidljivost varijable

Kod inačice 1.4 JDK programskog paketa uočena je zanimljiva pojava uobičajenog ophođenja s privatnim varijablama. Ukoliko nije eksplicitno naveden parametar `-verify` prilikom prevođenja klasa, prevoditelj neće provjeriti programski kod u potrazi za nedozvoljenim pristupima privatnim varijablama. Posljedica toga je moguć pristup sadržaju privatnih varijabli i mijenjanje njihovih varijabli.

```
public class PrivatniString2
{
       private String str = "Tajni podatak";
       public void test()
       {
              System.out.println("Test2: " + str);
       }
}

public class PrivatniString
{
       public static void main(String[] args)
       {
              PrivatniString2 t2 = new PrivatniString2();
              System.out.println("Test1: " + t2.str);
              t2.test();

              t2.str = "promijenjen";
              System.out.println("Test1: " + t2.str);
              System.exit(0);
       }
}
```

Ukoliko se ovaj program pokrene na ranjivim Java programskim paketima, dobiva se sljedeći izlaz:

```
$ java PrivatniString
Test1: Tajni podatak
Test2: Tajni podatak
Test1: promijenjen
```

Ukoliko se isti program pokuša prevesti na inačici 1.5, pokušaj neće uspjeti iz razloga što novije verzije i u općenitom slučaju provjeravaju pristup privatnim varijablama:

```
$ javac PrivatniString.java
PrivatniString.java:6: str has private access in PrivatniString2
               System.out.println("Test1: " + t2.str);
                                                 ^
PrivatniString.java:8: str has private access in PrivatniString2
               t2.str = "promijenjen";
                 ^
PrivatniString.java:9: str has private access in PrivatniString2
               System.out.println("Test1: " + t2.str);
                                                 ^
```

```
3 errors
```

Iako je navedeni propust ispravljen, isti prikazuje kako programeri ne mogu uvijek biti sigurni da su deklaracije atributa varijabli ispravne i sigurne kako su to oni zamislili. Pravilno postavljanje atributa varijabli je jedan od preduvjeta onemogućavanja raznih sigurnosnih napada na aplikaciju. Jedan od važnih atributa varijable je i ispravno postavljanje vidljivosti (eng. *scope*) koja onemogućuje pristupanje varijabli izvan okvira za to predviđenog koda.

3.4. Propusti privilegiranog koda

Dozvola pristupa nekom objektu ili izvršenju nekog zadatka dodijeljena je samo ukoliko su sve pojedinačne domene zaštite uspješno izvršene te su dozvolile željenu akciju. Ukoliko iz nekog razloga programer želi zaobići neke od zaštita, potrebno je koristiti `doPrivileged` funkciju koja omogućuje izvršavanje dijela koda u privilegiranom okruženju. Navedenu tehniku je najčešće potrebno koristiti ukoliko dozvole na aplikacijskom nivou ne odgovaraju potrebnim dozvolama na sistemskom nivou za uspješno izvođenje željene operacije.
Napadač može zloupotrijebiti navedenu situaciju kako bi uvećao svoje ovlasti ili izbjegao zaštićeno okruženje (kao što je *applet sandbox*). Ukoliko postoji potreba za korištenjem `doPrivileged` funkcije, korisno je provjeriti jesu li svi korisnički predani parametri ispravnog oblika kako ne bi došlo do zaobilaženja zabrana, curenje povjerljivih informacija ili izvođenja neželjenih modifikacija programskog koda.

3.5. Propusti *static* varijabli koje nisu *final*

Ukoliko *static* varijabla nije *final*, zlonamjerni korisnik može ugroziti integritet aplikacije mijenjanjem vrijednosti varijable. Općenito se preporučuje postavljane prefiksa *final* na sve elemente (npr. klase, *static* varijable, itd.). Razlog postavljanja *final* prefiksa je u tome što obične *static* varijable mogu biti promijenjene od strane drugih *applet-a* i tako promijeniti normalan tijek izvršavanja aplikacije.
Kod Java *plug-in*-a 1.4.2 postojao je sigurnosni propust (*Cross-Site Java breaks Sandbox Isolation for Unsigned Applets*) kod kojega se nepotpisani (eng. *unsigned*) *applet* mogao ubaciti između komunikacije dva potpisana (eng. *signed*) *appleta* i promijeniti vrijednost *static* varijable. Time je *unsigned applet* zaobišao Java *sandbox* zaštitu te omogućio daljnju izgradnju napada. Ovo je samo jedan od mnogo sigurnosnih propusta vezanih uz *static* varijable koje nisu *final*, a koji omogućuju izvršavanje različitih vrsta napada na ranjive aplikacije.

4. Sigurnosne preporuke

Uz prethodno opisane preporuke vezane uz uobičajene sigurnosne propuste (neprovjeravanje varijabli, cjelobrojno zaokruživanje, vidljivost varijable, privilegirani kod, *static* varijable koje nisu *final*), u nastavku ovog poglavlja raspoložive su i neke druge preporuke koje znatno podižu razinu sigurnosti razvijanih aplikacija.

4.1. Korištenje *Security Manager* mehanizma

Korištenjem `SecurityManager` klase omogućeno je implementiranje sigurnosnih politika unutar razvijanih aplikacija. Ona omogućava, prije pokretanja potencijalno nesigurnih ili osjetljivih operacija, analizu tih operacija pri čemu se analiziraju njihovi sigurnosni efekti. Aplikacija pri tome može dozvoliti ili zabraniti analiziranu operaciju.
`SecurityManager` klasa sadrži brojne metode čiji nazivi započinju s riječju *check* (provjeri). Te metode se pozivaju od strane drugih različitih metoda u Java bibliotekama prije nego li te metode izvedu određene osjetljive operacije. Pri tome *Security Manager* mehanizam predstavlja mogućnost za onemogućavanje izvođenja analiziranih operacija pri čemu se baca određena iznimka. Poziv tih metoda provjere uobičajeno izgleda ovako:

```
SecurityManager security = System.getSecurityManager();
if (security != null) {
    security.checkXXX(argument,  . . . );
}
```

Pozvana metoda provjere jednostavno odgovara da li je analizirana operacija dozvoljena, a ako nije onda baca `SecurityException` iznimku. Jedino odstupanje od ovog pravila je za `checkTopLevelWindow` metodu koja vraća `false` ukoliko pozvana dretva nije povjerljiva za pozivanje novog prozora.
Security Manager mehanizam se uobičajeno koristi na *applet*-ima. Ipak, to nije njegova jedina primjena pa tako određene aplikacije kao što su npr. RMI (eng. *Remote Method Invocation*) poslužitelji za distribuirane aplikacije, zahtijevaju korištenje *Security Manager* mehanizma kako bi radili ispravno. Čak i ako korištenje *Security Manager* mehanizma nije zahtijevano za normalan rad, ipak se preporuča korištenje istog, ponajprije kod korištenja tuđih programskih biblioteka.

4.2. Korištenje sigurnosnih ekstenzija

Postoje tri ekstenzije važne za sigurnosnu infrastrukturu Java programskog jezika:

- JCE (eng. *Java Cryptography Extension*) – omogućava razvijanje sigurnijeg koda koji izvodi snažnije kriptiranje pri čemu se koriste različiti kriptografski algoritmi (DES, 3DES, Blowfish, …), ali i nekih drugih stvari poput sigurne razmjene tajnih ključeva,
- JSSE (eng. *Java Secure Sockets Extension*) – pruža sučelje za SSL (eng. *Secure Socket Layer*), te
- JAAS (eng. *Java Authentication and Authorization Service*) – omogućava autentikaciju krajnjih korisnika koji izvršavaju programski kod te autoriziranje samo određenih krajnjih korisnika za izvršavanje određenih operacija.

Za programere korištenje ekstenzija omogućava primjenu sigurnijih aplikacijskih sučelja (API). Naime, osnovni API elementi omogućavaju kreiranje digitalnih potpisa i *digest* dodatka na porukama, ali programeri su u osnovi više zainteresirani za kriptiranje podataka i autentikaciju korisnika što je raspoloživo kroz ekstenzije.

4.3. Osiguravanje sigurnog korištenja baze podataka

Uobičajeno je da aplikacije trebaju pristup na određenu bazu podataka. Pristup na bazu podatka osigurava se kroz JDBC (eng. *Java Database Connectivity*) aplikacijsko sučelje. Isporučitelji baza podataka najčešće posjeduju svoje implementacije JDBC specifikacije. Te različite JDBC implementacije posjeduju različite razine sigurnosti. Preporuča se odabir onih JDBC implementacija od pouzdanih isporučitelja, kao i onih koje pružaju takvu razinu sigurnosti koja je zahtijevana sigurnosnim i aplikacijskim politikama.
Prilikom korištenja baze podataka preporuča se izbjegavati direktan pristup na bazu podataka velikom broju klijenata. U tu svrhu potrebno je koristiti posredni aplikacijski poslužitelj koji će kreirati konekcije na bazu podataka, što je mnogo bolje nego kada veći broj korisnika samostalno upravlja svojim konekcijama na bazu podataka. Time se uz povećanje performansi istodobno postižu i sigurnosne prednosti. Autentikacija je pojednostavljena jer se na bazu podataka spaja samo „jedan“ klijent – posredni aplikacijski poslužitelj pa je time nadzor djelomično pojednostavljen. Pošto se na bazu podataka spaja samo posredni aplikacijski poslužitelj, tada mrežni segment između njih može biti bolje osiguran od neautoriziranih korisnika. Naravno, ukoliko se između njih nalazi Internet, to uglavnom nije moguće. Ipak, čak i u slučajevima kad se posredni aplikacijski poslužitelj i baza podataka nalaze na istom zaštićenom mrežnom segmentu, preporuča se odabir JDBC implementacije koja posjeduje mogućnosti kriptiranja podataka.
Prije implementiranja pristupnog modela korištenja baze podatka, nužno je izraditi sigurnosnu shemu baze podatka i njenih komponenata. Administratori baze podatka bi trebali davati samo one ovlasti nad tablicama i operacijama koje su potrebne za ispravan rad aplikacije. U slučajevima kad više aplikacija pristupa bazi podataka, potrebno je osigurati da svaka pri tome posjeduje vlastitu konekciju, korisničko ime i zaporku.

4.4. Pisanje sigurnijeg programskog koda

Najteži dio kod kreiranja sigurnijih Java aplikacija je upravo pisanje sigurnog programskog koda. Programeri ne samo da moraju naučiti pisati aplikacije u Java programskom jeziku, već uz to moraju naučiti pisati sigurne Java aplikacije. Pogreške kod pisanja programskog koda ne mogu uzrokovati prepisivanje spremnika (eng. *buffer overflow*), osim ako se isti nalazi u *Java Virtual Machine* (JVM).

Naime, za razliku od C i C++ aplikacija koje u osnovi imaju gotovo neograničen pristup memoriji, Java nema izravan pristup memoriji već njime upravlja JVM unutar kojeg se aplikacije i izvršavaju.
Unutar ovog poglavlja nabrojene su određene preporuke koje mogu dodatno podići razinu sigurnosti razvijanog programskog koda:

1. Provjera inicijalizacije objekta
Mnogi Java programeri smatraju kako nije moguće alocirati objekt bez da se pokrene definirani konstruktor. Ali to nije istina jer postoji više načina za alociranje ne-inicijaliziranog objekta. Najlakši način zaštite od ovog problema je pisanjem klasa kod kojih se, prije nego što ikoji objekt napravi išta, provjerava da li je objekt ispravno inicijaliziran. To je moguće napraviti na sljedeći način:

- definiranjem svih varijabli da budu privatne – ako je potrebno omogućiti vanjskom kodu pristupanje pojedinim varijablama, tada je to potrebno izvesti korištenjem *get* i *set* metoda koje vraćaju i postavljaju vrijednosti definiranih varijabli,
- dodavanjem *private boolean* varijable inicijalizacije (`initialized`) u svaki objekt,
- predefiniranjem svih varijabli u konstruktoru, te
- postavljanjem u svim metodama procedure provjeravanja ukoliko varijabla inicijalizacije (`initialized`) ima vrijednost `true` (izuzeće od toga je moguće u konstruktoru).

2. Ograničavanje pristupa klasama, metodama i varijablama
Svaka klasa, metoda i varijabla koja nije definirana kao privatna (*private*) pruža potencijalnu pristupnu točku napadačima. Prilikom pisanja programskog koda, svi elementi bi trebali biti *private*. Ukoliko je nešto potrebno ostaviti da ne bude *private*, tada se preporuča dokumentiranje razloga zbog kojeg je to tako učinjeno.

3. Nastojanje korištenja *final* karakteristike
Ukoliko klasa ili metoda nisu konačne (*final*), napadač ih može pokušati proširiti na opasan i nepovjerljiv način. Zbog toga bi ti elementi trebali biti definirani kao *final*. Svi elementi koji ne mogu biti *final* – trebali bi biti dokumentirani s objašnjenjem zašto to nije slučaj.
Ovim zahtjevom od programera se traži ograničavanje jedne važne osobine objektno-orijentiranog programiranja – proširivosti, tj. višestruke iskoristivosti. To je korisna osobina, ali kad se nastoji pisati siguran kod, ta osobina napadačima pruža različite nedozvoljene mogućnosti.

4. Ne oslanjati se na sigurnost pruženu paketom
Klase, metode i varijable koje nisu eksplicitno definirane kao *public*, *private* i *protected*, dostupne su unutar istog paketa što nije dovoljno dobra sigurnosna mjera. Klase u Java programskom jeziku nisu zatvorene pa napadač može unijeti nove klase u paket koje potom koriste elemente koje se nastojalo sakriti. Većina JVM-ova ne posjeduje mehanizam kojim bi onemogućio unošenje novog koda u tuđe pakete (iznimka je `java.lang` paket) pa je jedini način onemogućavanja toga tijekom razvoja.

5. Izbjegavati korištenje *inner* klasa
Pogrešno je mišljenje kako se unutarnjim (*inner*) klasama može pristupati samo iz vanjskih (*outer*) klasa. Java *bytecode* ne razumije koncept *inner* klasa pa su tako definirane klase prevedene od strane prevodioca u obične klase koje mogu biti dostupne od strane bilo kojeg koda u istom paketu za što prethodno pravilo kaže kako nije dovoljno sigurno.
Također, unutarnje klase imaju pristup elementima pripadne vanjske klase iako ti elementi mogu biti definirani kao *private*. Da bi se to omogućilo, pošto je unutarnja klasa prevedena kao obična klasa, prevodilac mijenja status definiranih *private* elemenata iz vanjske klase da bi isti bili dostupni iz cijelog paketa.

6. Definiranje ne-klonirajućih klasa
Mehanizam kloniranja objekata unutar Java jezika može omogućiti napadačima kreiranje novih instanci definiranih klasi bez pokretanja niti jednog od definiranih konstruktora. Čak i ukoliko definirana klasa nije klonirajuća (eng. *clonable*), napadač može definirati pod-klasu implementirajući `java.lang.Clonable` klasu. Time napadač može kreirati nove instance početne klase. Nove instance se kreiraju kopiranjem memorijskih preslika postojećih objekata što često nije prihvatljiv način kreiranja novih objekata. Stoga je potrebno izričito definirati objekte ne-klonirajućim što je moguće izvesti sljedećom metodom koja baca iznimku:

```
public final void clone() throws java.lang.CloneNotSupportedException {
        throw new java.lang.CloneNotSupportedException();
}
```

Ako je nužno imati klasu koja je klonirajuća, i dalje postoji način na koji je moguće zaštiti klasu. Naime, prilikom definiranja metode kloniranja, potrebno ju je definirati kao *final*. U slučaju kad definirana metoda kloniranja nasljeđuje višu klasu, metoda `clone()` može ovako izgledati:

```
public final void clone() throws java.lang.CloneNotSupportedException {
        super.clone();
}
```

Prethodno navedena metoda onemogućava mijenjanje iste.

7. Definiranje ne-serijabilnih klasa

Serijalizacija se koristi za zapisivanje unutarnjeg stanja objekta. Stoga ona može biti opasna kod programiranja jer ista omogućava nekim vanjskim elementima otkrivanje unutarnjeg stanja definiranih objekata. Objekti se mogu serijalizirati u niz okteta koji potom može biti jednostavno pročitan. Time je omogućena inspekcija ukupnog unutarnjeg stanja objekta, uključujući sve elemente koji su označeni kao *private* te unutarnje stanje svih objekata na koje se oni referenciraju.
Da bi se onemogućila serijalizacija objekata potrebno je definirati `writeObject()` metodu koja je tipa *final* kako ju metode definirane u pod-klasama ne bi mogle pregaziti:

```
private final void writeObject(ObjectOutputStream out)
throws java.io.IOException {
        throw new java.io.IOException("Objekt se ne može serijalizirati");
}
```

8. Onemogućavanje de-serijalizacije

Čak i kad klasu nije moguće serijalizirati, postoji mogućnost da ju je moguće de-serijalizirati. Time neki vanjski elementi mogu kreirati niz okteta koji se de-serijalizira u instancu definirane klase. To nije poželjno jer aplikacija nema kontrolu nad stanjem u kojem se nalazi taj de-serijalizirani objekt. Stoga se de-serijalizacija može promatrati kao jedan oblik konstruktora koji se otežano kontrolira.
Napadi ovakvog tipa se mogu onemogućiti na način da se onemogući de-serijalizacija niza okteta u instancu definirane klase. To je moguće izvesti deklariranjem `readObject()` metode na sljedeći način:

```
private final void readObject(ObjectInputStream in)
throws java.io.IOException {
       throw new java.io.IOException("Klasa se ne može de-serijalizirati");
}
```

9. Izbjegavati uspoređivanje klasa po imenu

Ponekad je potrebno usporediti klase dva objekta kako bi se saznalo ukoliko su oni iste klase ili ukoliko određeni objekt pripada nekoj specifičnoj klasi. Prilikom izvršavanja ovakvih usporedbi, potrebno je imati na umu postojanje više klasa istog imena u JVM-u. Stoga je uspoređivanje klasa po imenu neispravno jer različite klase mogu imati isto ime. Bolje je rješenje kad se uspoređuju objekti direktno. Npr. za dana dva objekta A i B, usporedba bi trebala izgledati ovako:

```
if(a.getClass() == b.getClass()){
        // objekti a i b su iste klase
}else{
        // objekti a i b nisu iste klase
}
```

Prethodni primjer opisuje slučaj gdje se za dva objekta uspoređuju njihove klase. Ponekad je potrebno usporediti objekt s određenim nazivom klase. U sljedećem primjeru prikazana je usporedba koja se ne preporuča između objekta A i naziva klase „Klasa“:

```
if(a.getClass().getName().equals("Klasa")){ // nepreporučena usporedba
        // klasa objekta a se zove Klasa
}else{
        // klasa objekta a ima neki drugi naziv
}
```

Bolji način uspoređivanja je sljedeći:

```
 if(a.getClass() == this.getClassLoader().loadClass("Klasa")){
        // klasa objekta a se zove jednako kao i klasa koju ova klasa
        // naziva "Klasa"
}else{
        // klasa objekta a se ne zove jednako kao i klasa koju ova klasa
        // naziva "Klasa"
}
```

10. Izbjegavati pohranjivanje tajnih podataka u programskom kodu

Programeri često padaju u iskušenje da određene osjetljive informacije poput kriptografskih ključeva, uključe u aplikacije i biblioteke. Tako pohranjene informacije su u potpunosti dostupne svakome tko izvršava taj programski kod. Ne postoji efikasna metoda kojom bi se zlonamjerni programer ili određena aplikacija spriječila od kopanja po programskom kodu i otkrivanju osjetljivih informacija.

5. Zaključak

Aplikacije pisane u Java programskom jeziku imaju određene nedostatke koji su karakteristični za većinu programskih jezika. U prvom redu to su nedostaci uzrokovani nepažnjom ili neznanjem programera. Također, povremeno se u određenim programskim bibliotekama otkriju određeni propusti u standardnim funkcijama. Ipak, Java posjeduje i prednosti koje je čine veoma pogodnom za korištenje, U prvom redu to je objektna orijentiranost sa svim svojim karakteristikama (polimorfizam, klase, metode, nasljeđivanje, enkapsulacija, apstrakcija, itd.). Također, Java je neovisna o operacijskom sustavu što omogućava izvođenje istog koda na više različitih platformi. Zahvaljujući korištenju *Java Virtual Machine* mehanizma koji izvodi i kontrolira programski kod, aplikacija ne posjeduje neograničeni pristup memoriji pa je teoretski nemoguće izvesti prepisivanje spremnika. Time su narušene performanse, ali je razina sigurnosti podignuta na višu razinu.

Uz sve opisane prednosti Java jezika, Java posjeduje i mogućnost korištenja različitih ekstenzija koje omogućavaju autentikaciju, autorizaciju, kriptiranje, korištenje SSL-a i sl. Sigurnosti Java programskog jezika znatno doprinosi i metodologija prema kojoj se otkriveni propusti u Java programskom jeziku ne razotkrivaju javnosti.

Unatoč svim standardnim i nestandardnim mehanizmima koji podižu razinu sigurnosti Java programskog jezika, najveći broj sigurnosnih propusta uzrokovan je neznanjem programera. Stoga je najbolja prevencija protiv sigurnosnih propusta dobro poznavanje programskog jezika i programskih biblioteka koje se koriste.

6. Reference

[1] Cross-Site Java breaks Sandbox Isolation for Unsigned Applets, http://www.securityfocus.com/archive/1/341815, srpanj 2006.

[2] Non final public static variables in 2D source, http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4126751, srpanj 2006.

[3] Security Code Guidelines, http://java.sun.com/security/seccodeguide.html, srpanj 2006.

[4] Java Security, http://java.sun.com/javase/technologies/security.jsp, srpanj 2006.

[5] Java and JVM security vulnerabilities, LSD, http://www.blackhat.com/presentations/bh-asia-02/LSD/bh-asia-02-lsd.pdf#search=%22lsd%20java%20security%22, srpanj 2006.

[6] William Rushmore, GIAC Security Essentials: Securing Server Side Java, 2003.