Säker exekvering av pythonkod

C A R L B R I N G

KTH Datavetenskap
och kommunikation

Examensarbete
Stockholm, Sverige 2010

Säker exekvering av pythonkod

C A R L B R I N G

Examensarbete i datalogi om 15 högskolepoäng
vid Programmet för datateknik
Kungliga Tekniska Högskolan år 2010
Handledare på CSC var Mikael Goldmann
Examinator var Mads Dam

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2010/bring_carl_K10005.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Säker exekvering av pythonkod

Sammanfattning

Säkerhet i programvara är en allt högre prioritet för både stora multinationella företag såväl som individer. En metod för säkra exekvering av skadlig kod är genom att köra koden i en så kallad Sandlåda, vilket simulerar en begränsad exekveringsmiljö. Den här rapporten kommer att undersöka hur en Sandlåda kan implementeras för programmeringsspråket Python. Flera olika implementationer kommer att utvärderas för att reda ut vilken av de olika metoderna är bäst lämpade för användning i testsystemet Kattis[1].

Safe execution of Python code

Abstract

Safety in software is becoming an increasingly higher priority for both large multinational companies aswell as individuals. A method for safe execution of potentially malicious code is to run the code in a so-called Sandbox, which simulates a limited environment to execute in. This report will investigate how a Sandbox can be implemented for the programminglanguage Python. Different implementations of Sandboxes will be evaluated to sort out which of the various methods are best suited for use in the testing system Kattis.

Innehållsförteckning

1. Förkortningslista

- VM: Virituell Maskin
- DoS: Denial of Service
- JVM: Java Virtual Machine
- NASA: National Aeronautics and Space Administration
- CERN: ursprungligen "Conseil Européen pour la Recherche Nucléaire", men numera "Organisation Européenne pour la Recherche Nucléaire"
- JIT: Just-in-Time

2. Inledning

Python är ett programmeringsspråk som vuxit markant i popularitet under det senaste decenniet. Programmeringsspråket används till att skapa allt från enkla skript för privatpersoner till stora företagsapplikationer för t.ex. Google. En sådan utbredd användning sätter många krav på programmeringsspråkets säkerhet.

Samtidigt som användarnas exponering mot internet växer så blir säkerheten också allt mer kritisk. Till exempel så överröses användare av Flash-program[2] då de surfar på många websidor idag. Flash är en platform som används brett bland en stor variation av websidor. Flash används bland annat för att skapa reklam, animationer, interaktiva funktioner och spel. För att undvika att Flash utnyttjas av illasinnade personer som konstruerar farliga Flash-program och lägger ut på internet så använder man sig av en teknik som kallas "Sandlåda".

Genom att använda sig av ”Sandlådning” så kan man köra potentiellt skadlig kod i en säkert avskärmad miljö. Den avskärmade miljön hindrar programmet att utföra potentiellt skadliga operationer. För närvarande så finns det ingen inbyggd Sandlåda-funktion i Python och det finns inte heller något officiellt sätt att implementera det på. Däremot så finns det många olika inofficella förslag på hur man implementerar en Sandlåda för Python. Hur dessa olika metoder presterar jämtemot varandra är dock väldigt oklart.

Den här rapporten skall undersöka hur ett urval av dess Sandlåde-metoder i Python går att implementera och sedan mäta hur pass bra de presterar. På så sätt så ska rapporten utvärdera de olika metoderna. Utvärderingens syfte är framförallt till för att undersöka hur pass väl lämpat Python är att köras hos Kattis, vilket är Kungliga Tekniska Högskolans testsystem för automatisk rättning av programmeringsuppgifter.

Kattis hanterar en ständig flod av okänd kod som skall köras på systemets hårdvara. Eftersom systemet är mycket populärt och har stor användning i flera programmeringsrelaterade kurser så vore det bra att stödja så många programmeringsspråk som möjligt.

Det här ställer höga krav på Kattis som tjänst, det krävs hög säkerhet, stabilitet och prestanda. Det är kritiskt att tjänsten inte går att utnyttja till illasinnade ändamål som t.ex. fusk eller sabotage.

3. Teori & Bakgrund

För att kunna gå in mer på djupet kring säker exekvering av pythonkod så är det nödvändigt att

känna till grundläggande information kring ämnet.

3.1 Programmeringsspråket Python

Python är ett programmeringsspråk som uppfunnits av nederländaren Guido van Rossum[3] och första implementationen släpptes året 1991[4]. Guido har sedan fortsatt att utveckla Python och jobbar fortfarande med utvecklingen av Python via sitt jobb hos sökmotorn Google.

Python är implementerat under en s.k Open Source licens, vilket framförallt innebär att Python är fritt att använda, även för kommersiellt bruk. Licensen för Python administreras av företaget "Python Software Foundation"[5], vilket är en icke vinstdrivande organisation startad för att skydda de intellektuella rättigheterna för Python.

Python beskrivs som ett mycket kraftfullt och dynamiskt programmingsspråk med många olika tillämpningar. Det följer utav att Python stöder flera olika programmeringsparadigmer, däribland objektorienterad programmering, imperativ programmering och funktionell programmering. I Python så spelar indenteringen en stor roll då det är en stor del av programmeringsspråkets syntax. Standardbibloteket är stort och mycket väl underhållet, vilket är en bidragande faktor till språkets framgång.

Det finns 3 olika implementationer av programmeringsspråket Python som uppfyller produktionskvalitet[6]:

- CPython är den ursprungliga implementationen av programmeringsspråket Python[7]. CPython är skrivet i programmeringsspråket C och är tillgängligt till en rad olika operativsystem.
- Jython är en Java-implementation av programmeringsspråket Python[8]. Jython körs på Javas virituella maskin och underhålls för närvarande av utvecklare som jobbar för Sun[9].
- IronPython är en .NET implementation av programmeringsspråket Python och är skrivet helt i C#[10].

Python finns att hitta som standardkomponent i många operativsystem och har bred användning på stora företag och organisationer så som Google, NASA[11] och CERN[12].

3.2 Sandlåda

Namnet Sandlåda kommer utav en liknelse till verkligehetens sandlåda. Vi placerar våra barn i sandlådor där de får leka fritt. På samma sätt fungerar en Sandlåda inom datavärlden. Genom att köra program (småbarn) i en avskärmad miljö (sandlådan) så kan programmen köra fritt (leka) utan att ha någon påverkan utanför exekveringsmiljön.

Begreppet Sandlåda är väldigt mångtydigt och kan förknippas med många olika saker bara inom datorvärlden. Inom mjukvaruutveckling så förknippas begreppet Sandlåda ofta med en utvecklingsmiljö som ännu inte är satt i produktion, men som simuleras i en produktionsliknande miljö[13]. Inom spel så beskriver begreppet Sandlåda en öppen virituell värld i någon form, som till exempel en rymdsimulator, havssimulator eller stadssimulator[14].

Även för datorsäkerhet har begreppet Sandlåda en egen betydelse. Inom datorsäkehet syftar Sandlåda på en kontrollerad exekveringsmiljö för program, denna miljö kan ha begränsade resurser inom till exempel nätverk, diskutrymme, arbetsminne eller operativsystem[15]. Det är den här typen av Sandlåda som är aktuell i rapporten. Genom att begränsa resurserna i exekveringsmiljön så kan

man försäkra sig om att program som körs i sandlådan inte kan göra någon skada.

Det finns flera olika exempel på generella sandlådor inom datorsäkerheten.

- Chroot jail är namnet på en metod specifik för Unix relaterade operativsystem. Metoden går ut på att man med operationen Chroot flyttar root-katalogen för en process och på så sätt begränsas processens externa åtkomst, därav namnet "Chroot Jail"[16]. Den här metoden begränsar framförallt åtkomsten av filsystemet.
- Virtuell maskin är en mjukvaruimplementation av en maskin, t.ex. en dator. Det finns i huvudsak två olika kategorier av virtuella maskiner.
  - Den första typen är systemvirtualiserande maskiner vilka ämnar att tillhandahålla en hårdvarunära miljö tillräcklig för att till exempel köra operativsystem på[17]. Det finns en rad olika operativssystemspecifika virituella maskiner med varierande egenskaper[18].
  - Den andra typen är processvirtualiserande maskiner vilket ämnar tillhandahålla en högre abstraktionsnivå lämpad för att skapa till exempel plattformsoberoende. Ett typiskt exempel på en processvirtualiserande maskin är programmeringsspråket Javas virtuella maskin[19].

3.3 Illasinnad kod

Illasinnad kod är ett samlingsnamn för en rad olika skadliga typer av programvara, t.ex. virus, maskar, trojanska hästar och spionmjukvara[20]. Det är ett mycket brett och aktuellt ämne i dagens samhälle.

Illasinnad kod har givetvis funnits väldigt länge bland mjukvara, men det är först under 90-talet som användandet exploderat. Ökningen av illasinnad kod beror till stor del på en enorm expansion av datoranvändande i samhället, där internet bidragit och blivit ett effektivt spridningsmedium.

Tidiga exempel på illasinnad kod som hamnat på alla världens tidningars framsidor är viruset melissa[21]. Melissa var ett virus som spreds via mail och den dåvarande versionen av Microsoft Word. Genom att användare öppnade ett bifogat dokument så infekterade viruset datorn och skickade genast nya mail till användarens kontakter genom en sårbarhet i Microsoft Word.

När man kategoriserar illasinnad kod så försöker man att kategorisera den efter syftet med koden och dess spridningsväg. Detta har också gett upphov till namn som går att dra liknelser till i verkligeheten, vilket är populärt inom datorvärlden.

- Virus är en speciell typ av illasinnad kod som vid infektion självreplikerar och sprider sig till datorn genom att gömma sig i annan programvara[22]. Syftet med virus kan vara att bara irritera användaren eller förstöra för användaren.
- Maskar är likt virus självreplikerande och sprider sig vidare via nätverkshål[23]. Ofta är maskar konstruerade för att inte märkas av användaren, men samtidigt utnyttja värddatorn med fristående program. Maskar är t.ex. populära bärare av olika spam-attacker.
- Trojanska hästar är kamouflerade som någonting oskyldigt för användaren, men bär i själva verket på någon form av illasinnad kod innuti[24]. Trojanska hästar är populära bärare av maskar och spionprogram.
- Spionmjukvara försöker spionera på användaren och kartlägger information som t.ex. lösenord, kreditkort, beteende och privatliv[25]. Självklart så försöker spionmjukvara även

förbli oupptäckt för användaren.

- Botnet är en term för nätverk av kontrollerade slavdatorer[26]. Botnets styrs vanligtvis av skaparen och expanderar med hjälp av kombinationer utav maskar och trojanska hästar. Genom att utnyttja infekterade slavdatorers tyrka så kan Botnets vara mycket kraftfulla och sända ut enorma mängder data i form av spam eller attacker.

Senaste tidens utveckling av illasinnad programvara har blivit mer inriktad mot att gynna skaparen i någon mån. Ofta är det i ekonomiska eller kriminella syften som man nu skapar illasinnad programvara. Utvecklingen har lett till att det numera är stora pengar inblandat i illasinnad mjukvara, antingen för att stjäla känslig information likt senaste intrånget i Googles mailtjänst[27] eller för att hyra enorma Botnets för att skicka ut spam och terrorisera konkurrenter med DoS-attacker.

4. Material

Material som används vid en utvärdering är mycket relevant information för att läsare ska kunna dra nytta av dess slutsats. I det här fallet kretsar utvärderingen framförallt kring mjukvara. Mjukvara, speciellt open source, har mycket hög förändringshastighet, det är därför viktigt att veta vilka versioner av mjukvaran som utvärderingen utförs på. Även hårdvara är relevant eftersom det avgör vilken typ av mjukvara som kan utvärderas.

4.1 Hårdvaruspecifikation

Testerna är utförda på följande hårdvara.

Modell	Dell Studio 1537
Processor	Intel Core2Duo T5800
Internminne	3GB DDR2 800MHz
Grafikkort	ATI Mobility Radeon HD 3400
Hårddisk	Hitachi 500GB
Nätverkskort	Broadcom NetLink

4.2 Mjukvaruspecifikation

Namn	**Version**	**Kommentar**
Operativsystem		
Windows 7 Professional 32bit	Build 7600	Värddator [http://www.microsoft.com/windows/windows-7/]
Ubuntu Karmic Koala	9.10	Viritualiserad [http://www.ubuntu.com/]

Programvara på Windows

VirtualBox	3.1.4	Virituell Maskin, kör Ubuntu [www.virtualbox.org/]

Programvara på Ubuntu/Linux

Linux Kernel	2.6.31-20-generic	[http://www.linux.org/]
Python	2.6.4	[www.python.org/]
Java Runtime Environment	1.6.0_15	[http://java.sun.com/]
Jython	2.5.1	[http://www.jython.org/]
PyPy	1.2	[http://pypy.org/]
RestrictedPython	3.5.1	[http://pypi.python.org/pypi/RestrictedPython/3.5.1]

5. Metod

Den här delen av rapporten ämnar beskriva hur utvärderingen fungerar och vilka aspekter den beaktar. Metod-delen kommer ge läsaren grundläggande information för att kunna utläsa relevant information ur Resultat-delen av rapporten.

5.1 Implementation

För att utvärdera olika metoder av säker exekvering av Python så bör man för det första implementera metoderna. Själva metoden i sig kan nämligen vara mer eller mindre problematiskt att impelementera, vilket bör beaktas i utvärderingssynpunkt.

Utvärderingen av impelementation kommer ske i en s.k. virituell maskin med hjälp av programvaran ViritualBox. Den virituella maskinen kommer att köra operativsystemet Ubuntu, vilket är en Linuxbaserad distrubution. Under detta operativsystem kommer de olika metoderna till säker exekvering av Python att implementeras.

Vid utvärderingen av implementations-steget av en metod så bör man beakta hur pass enkel *installation* och *konfigurering* är. Detta kan mätas genom att mäta hur lång tid det tar implementera, samt hur pass komplicerat det är att implementera.

5.2 Praktiskt test

För att utvärdera implementationer av säker exekvering av Python i praktiken så är det nödvändigt att göra riktiga tester av mjukvaran. För att effektivt utvärdera implementationen så körs automatiska tester med potentiellt farlig kod.

Ett Python-skript som kan konfigureras efter implementationens krav startar en serie tester som vardera är designade för att testa någon specifik sårbarhet. Skriptet skriver sedan ut en kortfattad

rapport för testerna med eventuella felutskrifter.

På så sätt så kan dessa praktiska tester snabbt kartlägga grundläggande egenskaper för implementationen. Det gör det enkelt att matcha egenskaper som inte får förekomma på Kattis system.

5.2.1 Test: Skriva till fil

Det är intressant att veta hur pass begränsad skrivning av filer är. Det här testet kommer att pröva att skapa olika filer, vilket potentiellt kan utnyttjas genom att t.ex. skapa massor av filer på olika platser som inte har säkra skrivrättigheter. Den här typen av kod skall inte gå att köra på Kattis.

5.2.2 Test: Systemanrop

Genom att använda systemanrop så kan man få tillgång till olika operativsystemspecifika funktioner[28]. Det finns många olika sätt att illasinnat använda systemanrop, t.ex. genom att upprepat anropa fork för att skapa fler processer. Den här typen av kod skall inte gå att köra på Kattis.

5.2.3 Test: Terminalkommandon

Terminalen är ett mycket kraftfullt verktyg i många operativsystem vilket ger tillgång till en stor variation av systemanrop och program med potential att utnyttjas illasinnat. Terminalkommandon som t.ex. *rm -rf *,* kan användas illasinnat för att förstöra data. Det här testet kommer att pröva att anropa ett ofarligt terminalkommando *echo*. Den här typen av kod skall inte gå att köra på Kattis.

5.2.4 Test: Nätverk

För att testa nätverket och åtkomst till internet så använder sig testet av ett Python-biblotek kallat httplib[29] som i sin tur utnyttjar sockets som via TCP/IP prövar att ansluta till google.se och lyssna efter ett lyckat HTTP-svar. Nätverksåtkomst kan användas illasinnat för DoS-liknande attacker och skall inte gå att köra på Kattis.

5.2.5 Test: HelloWorld

Ett simpelt test som skriver ut "Hello World!" i terminalen, vilket är någonting som ska fungera på Kattis.

6. Resultat

Här presenteras de resultat som undersökningen har kommit fram till. Resultaten ämnar att ge en klar bild av hur de olika implementationerna presterar med hjälp av att presentera testresultat och observationer.

6.1 CPython

CPython är ursprungsimplementationen av programmeringsspråket Python. Det är nödvändigt att utföra testerna på CPython för att att kunna jämföra resultaten från andra implementationer av Python.

6.1.1 Installation

CPython kommer förinstallerat i många moderna Linux-distrubitioner och behöver därför ingen installation. Det finns även installationspaket att tillgå på Pythons hemsida för väldigt många platformar.

6.1.2 Konfiguration

CPython behöver ingen konfiguration för att köra.

6.1.3 Praktiskt test

CPython ger följande resultat.

Test	Resultat	Output

Systemanrop.py	Sårbart	`Systemanrop: FORK`
		`Parent pid: 12172`
		`Child`

Filskrivning.py	Sårbart	`Writing the following to foobar.66: test foo bar test`
Terminalkommando.py	Sårbart	`echo`
Network.py	Sårbart	`Host:  www.google.se` `Response:  200 OK`
Helloworld.py	Säkert	`Hello World!`

Resultattabell över testkörning på CPython

Alla tester gick igenom felfritt under körning av CPython.

Totalt tog testerna **1** sekund att genomföra.

6.2 Jython

Jython är ett open source projekt likt Python som ämnar att tillhandahålla en implementation av programmeringsspråket Python. Skillnaden är att Python-koden kompileras om till Java-bytekode och körs sedan på Java Virtual Machine likt vanliga class-filer. Jython möjliggör dessutom att man kan kombinera Python- och Java-kod tillsamans på ett smidigt sätt.

6.2.1 Installation

För att installera Jython så behövs först Java installeras. Det görs smidigt via färdiga installationspaket som finns på Suns hemsida[30]. På Ubuntu så går det även att enkelt installera via den inbyggda pakethanteraren.

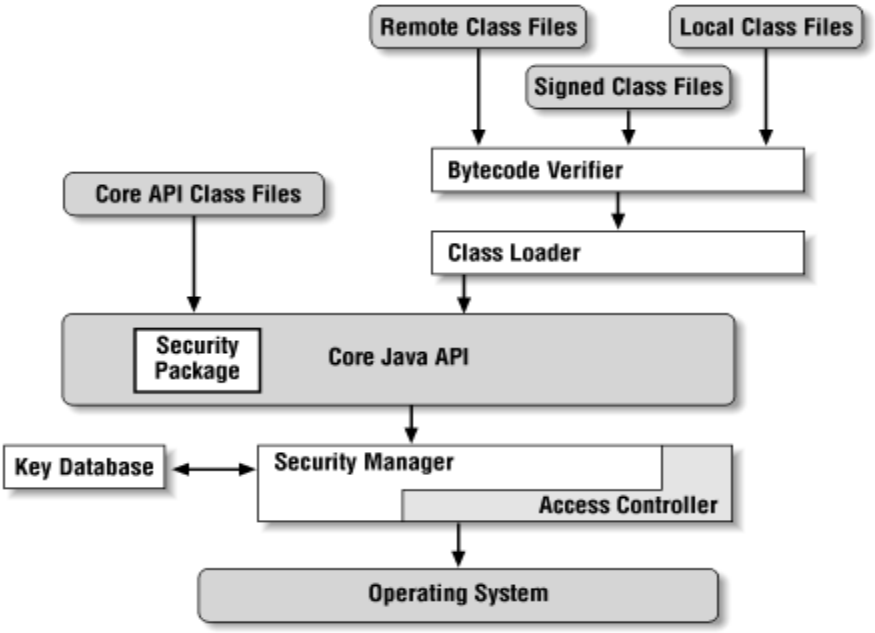
Installationen av Jython[31] görs via ett Jar-paket som körs med hjälp av Java. Installationen är automatisk och självkonfigurerande.

Summering: Installationen är mycket smidig.

6.2.2 Konfiguration

Jython i sig självt kräver ingen som helst konfiguration för att köra Python-filer på JVM, utan fungerar "out of the box".

För att utnyttja de inbyggda säkerhetsfunktionerna i JVM så vill man använda sig av dess Security Manager och Access Controller. Dessa två funktioner agerar som ett mellanlager inuti JVM:en som kontrollerar koden.



Beskrivning av de olika delarna i Java Virtual Machine

Security Manager och Access Controller behöver aktiveras genom att ge en flagga till JVM.

```
Java -Djava.security.manager [javaklass]
```

Det innebär att default inställningar läses in från Javas säkerhetskonfigurationsfiler, vilket för Ubuntu ligger under: */usr/lib/jvm/java-6-sun-1.6.0.15/jre/lib/security/java.policy*.

Det går utmärkt att konfigurera egna säkerhetskonfigurationsfiler och skicka med till JVM. Det görs enklast genom att specificera ett ytterligare argument för JVM som kompletterar default-konfigurationsfilen med den specificerade konfigurationsfilen.

```
Java -Djava.security.manager -Djava.security.policy=[säkerhetspolicy-fil] [klass]
```

Default-säkerhetskonfigurationsfilen innehåller minimala privilegier för att köra program i JVM, men för att Jython ska kunna köras behövs ytterliga privilegier.

```
grant{
  // Konfiguration för att kunna köra Jython med Security Manager
  permission java.util.PropertyPermission "*", "read,write";
  permission java.lang.RuntimePermission "createClassLoader";
  permission java.lang.RuntimePermission "getClassLoader";
  permission java.lang.RuntimePermission "getProtectionDomain";
  permission java.lang.RuntimePermission "accessDeclaredMembers";
  permission java.lang.RuntimePermission "modifyThread";
  permission java.lang.RuntimePermission "setIO";
  permission java.io.FilePermission "/usr/java/-", "read, execute";
  permission java.io.FilePermission "/usr/lib/jvm/java-6-sun-1.6.0.15/-",
"read, execute";
  permission java.io.FilePermission "/home/kalasclaes/jython2.5.1/-", "read,
execute";

  // Konfiguration för att köra tester, Jython behöver läs & skriv rättigheter
  permission java.io.FilePermission "/home/kalasclaes/Desktop/dkand10/-",
"read, execute";
};
```

Exempel på minimal säkerhetskonfigurationsfil för att köra tester i Jython

Syntaxen för säkerhetskonfigurationsfilen fungerar på följande vis:

```
grant [codeBase "sökväg"] {
      permission java.io.FilePermission "path" "skriv/läsflaggor";
}
```

Grant applicerar tillstånd antingen globalt eller på specifik java-kod (med hjälp av ange codeBase "sökväg"). Tillstånd beskrivs med `permission` efterföljt av en tillståndsklass och dess parametrar. I det här exemplet så anges `java.io.FilePermission`-tillståndsklassen[32] vilket har två parametrar, "sökväg" och "skrivflaggor".

För att slutligen köra Jython med en fungerade konfiguration av Security Manager så använder man kommandot:

```
jython -J-Djava.security.manager -J-Djava.security.policy=[policy-fil] [python-fil]
```

Summering: Konfigurationen av Jython är ganska komplicerad och det finns mycket lite information om hur säkerhetskonfigurationsfiler bör se ut för att fungera med Jython.

6.2.3 Praktiskt test

Jython <u>utan</u> Security Manager aktiverad ger följande resultat.

Test	Resultat	Output
Systemanrop.py	Säkert	`Unexpected error: <type 'exceptions.AttributeError'>`
Filskrivning.py	Sårbart	`Writing the following to foobar.95: test foo bar test`
Terminalkommando.py	Sårbart	`echo`
Network.py	Sårbart	`Host:  www.google.se` `Response:  200 OK`
Helloworld.py	Säkert	`Hello World!`

Resultattabell över testkörning på Jython utan Security Manager

Alla tester förutom systemanropet gick igenom felfritt vid körning av Jython utan Security Manager.

Systemanrop så som *fork* ser ut att inte vara implementerade och ger istället AttributeError för Jython, vilket i det här fallet även är önskat.

Totalt tog testerna **15** sekunder att genomföra.

Jython med Security Manager aktiverad ger följande resultat.

Test	Resultat	Output
Systemanrop.py	Säkert	`Unexpected error: <type 'exceptions.AttributeError'>`
Filskrivning.py	Säkert	`Unexpected error: <type 'java.security.AccessControlException'>`
Terminalkommando.py	Säkert	`Unexpected error: <type 'exceptions.TypeError'>`
Network.py	Säkert	`Unexpected error: <class 'socket.error'>`
Helloworld.py	Säkert	`Hello World!`

Resultattabell över testkörning på Jython med Security Manager

Enbart Helloworld.py-testet gick igenom med Security Manager aktiverad.

Totalt tog testerna **16** sekunder att genomföra.

Jython med Security Manager och 100 gångers repetition av alla tester aktiverad ger följande resultat.

Test	Resultat	Output
Systemanrop.py	Säkert	`Unexpected error: <type 'exceptions.AttributeError'>`
Filskrivning.py	Säkert	`Unexpected error: <type 'java.security.AccessControlException'>`

Terminalkommando.py	Säkert	`Unexpected error: <type 'exceptions.TypeError'>`
Network.py	Säkert	`Unexpected error: <class 'socket.error'>`
Helloworld.py	Säkert	`Hello World!`

Resultattabell över testkörning av 100 repetitioner på Jython med Security Manager

Enbart Helloworld.py-testet gick igenom med Security Manager aktiverad.

Totalt tog testerna **16.5** sekunder att genomföra trots att varje test körs 100 gånger var.
Det uppstår en fördröjning i uppstart på ungefär **2.4** sekunder för Jython[33].

6.3 Chroot Jail

Chroot är ett systemanrop som finns hos Unix-baserade operativsystem. Anropet ändrar positionen av filträdsroten till en specificerad katalog för en process och dess barnprocesser. På så vis så kan man begränsa diskåtkomst genom att isolera program under roten.

6.3.1 Installation

Det kan vara mycket komplicerat att köra program som Python i "Chroot jail" eftersom Python laddar många olika moduler utifrån och stöter då på problem när roten är flyttad. Men i Pythons standardbiblotek så finns en funktion i modulen *os*[34] för att skapa en "Chroot jail", vilket gör det hela betydligt smidigare.

Summering: Ingen installation behövs utöver Python.

6.3.2 Konfiguration

Genom att utnyttja standardbiblotekets implementation av chroot så blir konfigurationen mycket simpel. Det enda som krävs är att Python-programmet startas med root-privilegier och gör följande biboleksanrop.

```
os.chroot(path)
```

Summering: Konfigurationen är mycket enkel.

6.3.3 Praktiskt test

Chroot Jail ger följande resultat.

Test	Resultat	Output

Systemanrop.py	Sårbart	`Child`
		`Systemanrop: FORK`
		`Parent pid: 13790`

Filskrivning.py	Sårbart	`Writing the follow to foobar.146: test foo bar test`
Terminalkommando.py	Säkert	`[ingen output]`
Network.py	Säkert	`Unexpected error: <type`

		`'exceptions.LookupError'>`
Helloworld.py	Säkert	`Hello World!`

Resultattabell över testkörning på Chroot Jail

Nätverkstestet misslyckades och terminalkommandot gav varken felmeddelande eller output.

Totalt tog testerna **0.6** sekunder att genomföra.

6.4 PyPy

PyPy är en implementation av programmeringsspråket Python. Senaste versionen av PyPy implementerar egenskaperna för version 2.5 av Python. PyPy kommer med en JIT-kompilator för att generera bättre prestanda än ursprungsimplementationen CPython. PyPy kommer även i en stackless- och sandlåde-version. Stackless PyPy är tänkt att kunna hantera en stor mängd trådar effektivt. Sandboxed PyPy fokuserar på att köra Python-applikationer i en säker sandlåde-miljö. Sandboxed PyPy kommer att användas i testerna nedan.

6.4.1 Installation

Tyvärr hänger sig installationen av PyPy med följande felmeddelande:

```
[sandlib:call] ll_os.ll_os_getenv('PYPY_GENERATIONGC_NURSERY')
```

Enligt en gammal chatlogg[35] från PyPys irc-kanal så beskrivs det som en bug relaterad till operativsystem som körs i VM av Windows. Tyvärr så finns ingen lösning att finna för det här problemet i denna stund.

6.4.2 Konfiguration

Inte genomförbart på grund av fel i *6.4.1 Installation.*

6.4.3 Praktiskt test

Inte genomförbart på grund av fel i *6.4.1 Installation.*

6.5 RestrictedPython

RestrictedPython möjliggör pythonkod att exekvera i en begränsad miljö genom att i kompileringssteget ta hänsyn till en rad användardefinerade restriktioner.

6.5.1 Installation

RestrictedPython-modulen fanns inte med i grundbibloteket och behöver därför installeras. Modulen använder ez_setup vid installation som också saknas i grundbibloteket. Installation av ez_setup går till på följande vis:

```
wget http://peak.telecommunity.com/dist/ez_setup.py
sudo python ez_setup.py
```

RestrictedPython-modulen installeras sedan enkelt genom att extrahera paketet och skriva på följande vis:

```
sudo python setup.py install
```

Summering: Installationen är smidig.

6.5.2 Konfiguration

```
# importera biblotek och restriktioner
import sys
from RestrictedPython import compile_restricted
from RestrictedPython.PrintCollector import PrintCollector
from RestrictedPython.Guards import safe_builtins
from RestrictedPython.Eval import default_guarded_getitem
# definera restriktioner
_print_ = PrintCollector
_getattr_ = getattr
_getitem_ = default_guarded_getitem
restricted_globals = dict(__builtins__ = safe_builtins)
# kompilera testet med restriktioner och kör sedan testet
def restrict(argv=None):
        argv = sys.argv
        src = open(argv[1]).read()
        code = compile_restricted(src, '<string>', 'exec')
        exec(code)
        print main()
```

Exempel på minimal konfigurationsfil för att köra tester med RestrictedPython

Det finns mycket ont om information hur RestrictedPython skall konfigureras. Den konfigurationsfilen som anges ovan är baserad på grundläggande exempel från RestrictedPythons hemsida.

Konfigurationsfilens syfte är framförallt att överlagra ett antal operationer och därmed tillämpa restriktionerna. Tyvärr krävs ett fåtal ändringar i de tester som körs för att skrivning till stdout (print) skall fungera.

Summering: Konfigurationen är mycket dåligt dokumenterad och krånglig.

6.5.3 Praktiskt test

RestrictedPython ger följande resultat.

Test	Resultat	Output
Systemanrop.py	Säkert	`UnboundLocalError: local variable 'pid' referenced before assignment`
Filskrivning.py	Säkert	`Unexpected error: <type 'exceptions.NameError'>`

Terminalkommando.py	Säkert	`Unexpected error: <type 'exceptions.NameError'>`
Network.py	Säkert	`Unexpected error: <type 'exceptions.NameError'>`
Helloworld.py	Säkert	`Hello World!`

Resultattabell över testkörning på RestrictedPython

Enbart Helloworld.py-testet gick igenom med körning i RestrictedPython.

Totalt tog testerna **1.1** sekunder att genomföra.

7. Diskussion

Den här delen av rapporten ämnar att diskutera resultaten i föregående rubrik. Diskussionen kommer att ta upp utmärkande resultat i de olika testförsöken och reflektera över vad som är bra, dåligt, kan bli bättre och så vidare. De åsikter som presenteras här kommer att lägga grunden för den slutsats som presenteras i nästa rubrik.

7.1 CPython

Genom att först köra alla tester i CPython så får vi en översikt hur standardimplementationen av Python presterar. Vi vet dessutom redan flera styrkor hos CPython som:

- stöd för hela Pythonbibloteket
- bra prestanda
- bra underhållet
- hög produktionskvalitet

Det största problemet som belyses i den här rapporten är säkerheten hos CPython. Det räcker med att titta på *6.1.3 Praktiskt Test* så ser vi att alla potentiellt farliga tester fungerar utmärkt i CPython. Detta är förstås ett väntat resultat vars syfte endast är att belysa de styrkor som följande Sandlåde-implementationer har.

7.2 Jython

Jython imponerar med en mycket smidig installation i *6.2.1 Installation*, vilket Jython delvis har Sun att tacka för. Om vi helt hoppar över konfigurationen och tittar på testerna som körs utan någon Security Manager i *6.2.3 Praktiskt Test* så presterar Jython däremot inte speciellt bra.

Med körning tillsamans med konfigurationsfil för Security Manager i *6.2.3 Praktiskt Test* så misslyckas alla skadliga tester. I testet används en mycket simpel policy-fil tillsamans med grundpolicy-filen för Java, men detta kan lätt utökas med mycket robusta och väldokumenterade funktioner ur standardbibloteket för Security Manager.

Vid körning av 100-gångers repetition av testerna tillsamans med konfigurationsfil för Security Manager i *6.2.3 Praktiskt Test* så får man ändå säga att Jython presterar mycket bra. Den stora svagheten hos Jython är dess overhead, det går åt mycket arbete i uppstarten av Jython för att kompilera om pythonkod till Java-kod och sedan starta JVM, vilket fördröjer exekveringen av

pythonkod. I *6.2.3 Praktiskt Test* ser vi dock att hastigheten vid längre körning är i linje med de andra implementationerna.

Det som imponerar i Jython är hur kraftfull och väldokumenterad Security Manager är, vilket förstås är någonting som kommer gratis med Java-implementationen. Intrycket av Jython självt är positivt med bra underhåll och dokumentation. Tyvärr så är uppstartstiden för program ett problem för Kattis där hastighet är mycket viktigt.

7.3 Chroot

Chroot är ett systemanrop som numera finns implementerat i de flesta Unix-besläktade operativsystem. Därav så blir installationen i *6.3.1 Installation* trivial, vilket är positivt. Att konfigurera en bra miljö för att köra Chroot till är däremot mycket komplicerat. Problemet är att programmeringsspråket Python inte är konstruerat för att fungera väl i en isolerad miljö som Chroot skapar. De flesta applikationer kommer att stöta på problem med importering av moduler och liknande. Detta resulterar i att Chroot kanske skapar mer problem än det löser. En annan aspekt att ta upp med Chroot är att det kräver root-access för att fungera, detta är någonting man vill undvika inom säkerhet. Det är därför lämpligt att avsäga sig root-rättigheterna efter chroot har anropats, vilket Python även har en funktion för kallad `os.setuid(uid)`.

I *6.3.2 Konfiguration* så används istället Pythons färdiga funktion för Chroot, vilket anropas efter att alla moduler förhoppningsvis har importerats och på så sätt undviks mycket krångel. I *6.3.3 Praktiskt Test* så falerar två tester framförallt för att tillgång saknas i filsystemet.

Chroot är ett bra alternativ som inte offrar någon prestanda och fungerar antagligen mycket bra i kombination med en kompleterande systemanropsövervakare som blockerar systemanrop hos CPython. Systemanropsövervakare är precis vad Gunnar Kreitz[36] föreslår i en mailkonversation angående potentiella lösningar till säker exekvering av Python. Ett problem med systemanropsövervakare är däremot att implementationer kan vara mycket oportabla och bökiga att konfigurera.

7.4 PyPy

I *6.4.1 Installation* gick PyPy med stor besvikelse inte att installera på testmiljön som fanns till hands för utvärderingen. Det finns ingen information att hitta på PyPy:s hemsida och även utanför hemsidan så finns mycket begränsad information. Tyvärr så tvingas därför PyPy att uteslutas ur utvärderingen helt.

7.5 RestrictedPython

RestrictedPython har enligt *6.5.1 Installation* ett välfungerande system för automatisk installation av modulen. Men i *6.5.2 Konfiguration* så saknas bra dokumentation kring hur RestrictedPython fungerar och resultatet blir mycket trassel och förlorad tid. Det blir därför svårt att avgöra om funktionaliteten i RestrictedPython är robust eller ej.

I *6.5.3 Praktiskt Test* så tvingades testerna att modifieras för att fungera med RestrictedPython. Det beror på att RestrictedPython kompilerar om koden till att använda sig av en wrapper för print-funktionen i Python som ändrar specifikationen av hur programmeringsspråket är definerat att se ut.

Trots att det praktiskta resultatet är mycket bra så saknar RestrictedPython dokumentationen för att

bevisa att det är robust nog att fungera i en produktionsmiljö som Kattis.

8. Slutsats

Kattis är ett system med fokus på säkerhet och prestanda, vilket är någonting som kan vara utmanande att kombinera. Detta syns också i resultaten för utvärderingen där ingen implementation lyckas ge ett perfekt resultat.

- PyPy gick inte att installera på testmiljön och blev därför helt utesluten ur utvärderingen.
- RestrictedPython presterar relativt bra i testerna men var problematiskt att integrera med tidigare Python-program. Kompatibilitetsproblemen i RestrictedPython beror delvis på konfigurationen som är för dåligt dokumenterad och saknar robusthet.
- Att använda Chroot var mycket smidigt i Python, men tyvärr så är det inte robust nog att hindra alla sårbara tester. Att kombinera Chroot med en operativsystemspecifik systemanropsövervakare är ett bra sätt eliminera de svagheter Chroot har vid ensam körning.
- Jython har mycket goda resultat och ett robust och väldokumenterad säkerhetssytem. Den enda riktiga nackdelen med Jython är den initiala tid det tar för program att starta upp p.g.a. kompileringssteget samt uppstart av JVM. Men eftersom det redan finns en Java-implementation hos Kattis så är uppstartstiden av JVM rimligtvis godtagbar.

De undersökta implementationerna presterar inte tillräckligt bra för att direkt tas i bruk i ett viktigt system som Kattis. Samtidigt är det viktigt att poängtera att de kandidater med bäst potential att användas i Kattis är Jython och Chroot. Om uppstartstiden för Jython faller under det acceptabla för Kattis så är Jython en utmärkt kandidat att köra säker exekvering av pythonkod.

9. Bilagor

Testscript.py

```
#!/usr/bin/python

import os
import sys
import time
import subprocess

#exec_command = "python restrict.py"
exec_command = "python"
#exec_command = "jython"
#exec_command = "jython -J-Djava.security.manager -J-
Djava.security.policy=/home/kalasclaes/Desktop/dkand10/jython.policy"
relpath = "tests/"
TIME = 90

def run_test(filename, testindex):

	print "TEST #" + str(testindex+1) + ": " + filename
	print "=================== OUTPUT ===================="

	test = subprocess.Popen(exec_command + " " + relpath + filename, shell = True)

	n = 0
	while test.poll() == None and n < TIME * 10:
		time.sleep(0.1)
		n += 1
	if test.poll() == None:
		test.kill()
		test.terminate()
		print "Process exceeded timelimit of " + str(TIME) + " seconds."
	test.wait()
	retcode = test.returncode

	print "================================================"

	if retcode:
		print filename + ": terminated UNSUCCESSFULLY\n"
	else:
		print filename + ": terminated SUCCESSFULLY\n"

def main(argv=None):
	start = time.time()

	tests = os.listdir("tests")
	for t, i in zip(tests, range(len(tests))):
		run_test(t, i)

	elapsed = (time.time() - start)
	print "Total time elapsed: " + str(elapsed)
	return 0
```

```
if __name__ == "__main__":
	sys.exit(main())
```

Filskrivning.py

```
import os
import sys
import random

def main(argv=None):
	if argv is None:
		argv = sys.argv
	try:
		filename = "foobar." + str(random.randint(0, 255))
		FILE = open(filename, "w")
		print("Writing the follow to " + filename + ": test foo bar test")
		FILE.write("test foo bar test")
		FILE.close()
	except:
		print "Unexpected error:", sys.exc_info()[0]
		return 1
	return 0

if __name__ == "__main__":
	sys.exit(main())
```

Helloworld.py

```
import os
import sys

def main(argv=None):
	if argv is None:
		argv = sys.argv
	try:
		print "Hello World!"
	except:
		print "Unexpected error:", sys.exc_info()[0]
		return 1
	return 0

if __name__ == "__main__":
	sys.exit(main())
```

Network.py

```
import os
import sys
import httplib

def main(argv=None):
	HOST = "www.google.se"    # The remote host
	try:
		conn = httplib.HTTPConnection(HOST)
		conn.request("GET", "/index.html")
```

```
            res = conn.getresponse()
            print "Host: ", HOST
            print "Response: ", res.status, res.reason
      except:
            print "Unexpected error:", sys.exc_info()[0]
            return 1
      return 0


if __name__ == "__main__":
      sys.exit(main())
```

Systemanrop.py

```
import os
import sys

def main(argv=None):
      if argv is None:
            argv = sys.argv
      try:
            pid = os.fork()
      except:
            print "Unexpected error:", sys.exc_info()[0]
            return 1
      if pid:
            print "Systemanrop: FORK"
            print "Parent pid: " + str(pid)
            os.waitpid(pid, 0)
      else:
            print "Child"
            sys.exit(0)
      return 0

if __name__ == "__main__":
      sys.exit(main())
```

Terminalkommando.py

```
import os
import sys

def main(argv=None):
      if argv is None:
            argv = sys.argv
      try:
            os.system("echo 'echo'")
      except:
            print "Unexpected error:", sys.exc_info()[0]
            return 1
      return 0

if __name__ == "__main__":
      sys.exit(main())
```

Referenser

Ross Anderson. 2001. *Security Engineering.* http://www.cl.cam.ac.uk/~rja14/book.html

En bok om datorsäkerhet som släppts gratis på internet av författaren.

Fred L. och Drake Jr. 2010. *Python documentation.* http://docs.python.org/

Programmeringsspråket Pythons huvudsakliga dokumentation och informationskälla.

Armin Rigo, Maciej Fijalkowski och Amaury F. d'Arc. 2010. *PyPy Sandbox.* http://www.pypy.org/

Hemsida för Python-implementationen PyPy.

Maciej Fijalkowski och Holger Krekel. 2009. *Python in a sandbox.* http://codespeak.net/pypy/extradoc/talk/pycon2009/pypy-sandbox/sandbox.pdf

En presentation av Sandboxing i PyPy.

Zope Corporation. 2009. *RestrictedPython.* http://pypi.python.org/pypi/RestrictedPython/3.5.1

Hemsida för Python-implementationen RestrictedPython.

Finn Bock. 2009. *Jython.* http://www.jython.org/

Hemsida för Python-implementationen Jython.

Sun Corporation. 2006. *Java Security Documentation.* http://java.sun.com/javase/7/docs/technotes/guides/security/

Dokumentationen av säkerheten i Java Virtual Machine.

Hänvisningar

1 Kattis, Testsystem [https://kattis.csc.kth.se/]
2 Adobe Flash [*www.**adobe**.com/products/**flash**/*]
3 Guido van Rossum [http://www.python.org/~guido/]
4 Python, Wikipedia [http://en.wikipedia.org/wiki/Python_(programming_language)]
5 Python Software Foundation, företag som håller i intellektuella rättigheterna till Python. [http://www.python.org/psf/]
6 Python, Produktionskvalietet [http://en.wikipedia.org/wiki/CPython]
7 Python, CPython [http://wiki.python.org/moin/CPython]
8 Python, Jython [http://wiki.python.org/moin/Jython]
9 Jython, Utvecklas av Sun [http://en.wikipedia.org/wiki/Jython#History]
10 Python, IronPython [http://wiki.python.org/moin/IronPython]
11 NASA, Amerikanska rymdmyndigheten [*www.**nasa**.gov/]*
12 CERN, Europeiska organisationen för kärnkraftsforskning [www.cern.ch/]
13 Sandlåda, begrepp inom mjukvaruutveckling [http://en.wikipedia.org/wiki/Sandbox_(software_development)]
14 Sandlåda, begrepp inom spel [http://en.wikipedia.org/wiki/Sandbox_game]
15 Sandlåda, begrepp inom datorsäkerhet [http://en.wikipedia.org/wiki/Sandbox_(computer_security)]
16 Sandlåda, Chroot jail [http://en.wikipedia.org/wiki/Chroot]
17 Virtuell maskin, systemvirtualiserande maskiner [http://en.wikipedia.org/wiki/Virtual_machine#System_virtual_machines]
18 Virtuell maskin, olika systemvirtualiserande maskiner [http://en.wikipedia.org/wiki/Operating_system-level_virtualization#Implementations], [http://en.wikipedia.org/wiki/Virtual_machines#List_of_virtual_machine_software]
19 Virtuell maskin, processvirtualiserande maskiner [http://en.wikipedia.org/wiki/Virtual_machine#Process_virtual_machines]
20 Illasinnad kod [http://en.wikipedia.org/wiki/Malware]
21 Illasinnad kod, Melissa viruset [http://en.wikipedia.org/wiki/Melissa_virus]
22 Illasinnad kod, Virus [http://en.wikipedia.org/wiki/Computer_virus]
23 Illasinnad kod, Maskar [http://en.wikipedia.org/wiki/Computer_worm]
24 Illasinnad kod, Trojanska hästar [http://en.wikipedia.org/wiki/Trojan_horse_(computing)]
25 Illasinnad kod, Spionmjukvara [http://en.wikipedia.org/wiki/Spyware]
26 Illasinnad kod, Botnets [http://en.wikipedia.org/wiki/Botnet]
27 Illasinnad kod, Attack mot Google [http://googleblog.blogspot.com/2010/01/new-approach-to-china.html]
28 Systemanrop, Linux [http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html]
29 Httplib, Python-biblotek [http://docs.python.org/library/httplib.html]
30 Java, Installationspaket [http://java.sun.com/javase/downloads/index.jsp]
31 Jython, Installationspaket [http://www.jython.org/downloads.html]
32 Jython, Tillståndsklass [http://java.sun.com/j2se/1.4.2/docs/guide/security/permissions.html]
33 Jython, Uppstart [http://www.jython.org/archive/22/userfaq.html#id6]
34 Chroot, Pythons standardbiblotek [http://docs.python.org/library/os.html#os.chroot]
35 Chatlogg, PyPy [http://tismerysoft.de/pypy/raw-irc-logs/pypy/%23pypy.log.20091209]
36 Gunnar Kreitz, En av skaparna till Kattis [http://www.csc.kth.se/~gkreitz/]