# Mechanizmy bezpieczeństwa udostępniane przez Javę

Temat III

# Model atakującego

- duża baza kodowa może prowadzić do pojawiania się
  - niechcąco lub
  - złośliwie

  konstrukcji narażających bezpieczeństwo,
- atakujący może wstawić małą ilość kodu na etapie kodowania lub dostarczania systemu,
- atakujący może używać dowolnych zasobów do wykorzystania wprowadzonej powyżej luki.

# Zabezpieczenia w języku – dostęp

- zmienne i klasy prywatne – bezpośredni dostęp tylko z poziomu właściciela deklaracji (uwaga na aliasy i inne obiekty tej samej klasy),
- zmienne i klasy zabezpieczone (protected) – bezpośredni dostęp tylko z poziomu właściciela deklaracji, jego podklas oraz klas tego samego pakietu,
- zmienne i klasy publiczne – dostęp mają wszyscy,
- zmienne i klady bez określenia – dostęp z poziomu właściciela deklaracji oraz klas tego samego pakietu, co właściciel.

# Zabezpieczenia w języku – ochrona pamięci

- brak możliwości bezpośredniego zapisu poza wyznaczony zakres danych
- obiekty i metody określone jako final nie mogą zostać zmienione ani nadpisane (SecurityManager polega na tej zasadzie)
- zakres tablic jest zawsze sprawdzany
- ograniczone rzutowanie obiektów
- zmiennych nie można używać przed inicjalizacją
- śmieciarka automatycznie zwalnia niepotrzebne obiekty (ale ich nie zeruje)

# Zabezpieczenia w języku – ochrona pamięci

- metody native mogą robić, co im się podoba
- pola final w bajtkodzie można zmieniać
- możliwość zacieniania deklaracji zmiennych
- inicjalizacja dzieje się w maszynie wirtualnej

# Dwa modele ochrony

1. Model z piaskownicą
2. Model z podpisywaniem

# Trzy wykorzystania

1. JDK 1.0 – kod bezpieczny (CLASSPATH) i kod niebezpieczny (URI)
   (kod bezpieczny nie przechodzi nawet przez weryfikator)
2. JDK 1.1 – kod bezpieczny (CLASSPATH+kod podpisany) i kod niebezpieczny (URI)
3. JDK 2.0 – drobnoziarniste zasady bezpieczeństwa

# Aktywne elementy modeli bezpieczeństwa

- weryfikator bajtkodu (verifier)
- ładowarka klas (class loader)
- zarządca bezpieczeństwa (security manager)

# Model z piaskownicą – główne zagrożenia

- ataki modyfikujące system
- ataki na prywatność użytkownika
- ataki przez wyczerpanie zasobów maszyny
- ataki denerwujące użytkownika

# Model z piaskownicą – czego nie można

Jeśli domyślne zasady bezpieczeństwa są niezmienione, to załadowany aplet, któremu nie ufamy nie może:

- czytać plików z systemu plików klienta,
- zapisywać do plików z systemu plików klienta,
- usuwać plików z systemu plików klienta (ani przez użycie File.delete(), ani przez użycie poleceń mv lub rm)
- przemianowywać plików (ani przez File.renameTo(), ani przez użycie poleceń mv lub rename)

# Model z piaskownicą – czego nie można c.d.

- tworzyć katalogów po stronie klienta (ani przez File.mkdir(), ani za pomocą polecenia mkdir)
- wypisywać zawartość katalogu
- sprawdzać, czy dany plik istnieje
- uzyskiwać informacje o pliku, w tym o jego rozmiarze, typie, i czasie modyfikacji

# Model z piaskownicą – czego nie można c.d.

- ustanawiać połączenia sieciowe z komputerami innymi niż ten, z którego pochodzi applet
- nasłuchiwać i przyjmować połączenia na jakimkolwiek porcie systemu klienta
- tworzyć okno głównego poziomu bez powiadomienia o tym, że okno to nie jest zufane
- uzyskiwać informacje o identyfikatorze użytkownika i jego katalogu domowym (w szczególności przez odczyt zmiennych systemowych user.name, user.home, user.dir, java.home, java.class.path)

# Model z piaskownicą – czego nie można c.d.

- definiować jakiekolwiek zmienne systemowe,
- uruchamiać jakikolwiek program po stronie klienta przy pomocy metod Runtime.exec()
- sprawiać, że interpreter Javy zakończy pracę w wyniku użycia System.exit() lub Runtime.exit()
- ładować dynamicze biblioteki po stronie klienta przy pomocy metod load() lub loadLibrary() w klasach Runtime i System

# Model z piaskownicą – czego nie można c.d.

- tworzyć i wykonywać operacje na wątku, który znajduje się poza tą samą grupą wątków (ThreadGroup), co applet
- tworzyć ładowarki klas (ClassLoader)
- tworzyć zarządcę bezpieczeństwa (SecurityManager)
- określać funkcje sterujące operacjami sieciowymi (np. za pomocą ContentHandlerFactory, SocketImplFactory lub URLStreamHandlerFactory)
- definiować klasy, które są częścią pakietów pochodzących z systemu klienta

## Weryfikator bajtkodu

Sprawdza, że:

- plik klasowy ma prawidłowy format
- nie nastąpi przekroczenie zakresu stosu operacji
- instrukcje bajtkodu mają poprawne typy
- nie pojawiają się niedopuszczalne konwersje danych
- prawa dostępu private, public, protected są przestrzegane (uwaga na final)
- wszystkie dostępy i zapisy do rejestrów są poprawne

# Weryfikator bajtkodu – przebiegi

1. format pliku klasowego
2. sprawdzenie bez analizy instrukcji
3. analiza przepływu danych
4. sprawdzanie w czasie wykonywania

## Weryfikator bajtkodu – format pliku

m.in.

- sprawdzenie magicznej liczby
- sprawdzenie, że wszystkie atrybuty mają prawidłową długość
- kod w bajtkodzie nie jest ani za długi, ani za krótki
- pula stałych (constant pool) może być prawidłowo sparsowana

## Weryfikator bajtkodu – analiza bez instrukcji

m.in.

- nie tworzone są podklasy klas final,
- metody final nie są nadpisywane
- każda klasa ma nadklasę (oprócz java.lang.Object)
- format puli stałych
- prawidłowość nazw oraz sygnatur typowych dla metod i pól w puli stałych

## Weryfikator bajtkodu – analiza przepływu danych

w każdym punkcie programu zachodzi m.in.

- stos operacji ma ten sam rozmiar i znajdują się na nim dane tych samych typów
- dostęp do rejestrów (zm. lokalnych) odbywa się zgodnie z typem danych
- metody są wywoływane z odpowiednią liczbą argumentów oraz ich typy są prawidłowe
- pola są zmieniane za pomocą wartości odpowiedniego typu
- argumenty wszystkich instrukcji bajtkodu tak na stosie, jak i w rejestrach są odpowiednich typów, jest ich odpowiednia liczba
- zmienne są prawidłowo zainicjalizowane

# Weryfikator bajtkodu – analiza przepływu c.d.

- instrukcje sterujące przepływem skaczą do poprawnych instrukcji,
- dostęp do zmiennych lokalnych odbywa się w ramach wyznaczonych metod,
- pozycje w puli stałych są wykorzystywane zgodnie z ich typami,
- obsługa wyjątków zaczyna się i kończy na poprawnych instrukcjach oraz początek obsługi znajduje się przed końcem

# Weryfikator bajtkodu – w czasie wykonania

pozostałe rzeczy, których

- nie da się sprawdzić statycznie
- sprawdzenie łatwiej jest zapewniać w trakcie wykonania

(uwaga: sprawdzanie w czasie wykonania jest drogie)

# Problemy z weryfikacją

- definicja bezpieczeństwa
- luki w projekcie systemu Javy
  http://www.cigital.com/hostile-applets/HostileArticle.html
- sprawdzanie problemów przez testowanie
- sprawdzanie problemów przez weryfikację oprogramowania (powoli)

# Ładowarki klas – funkcje

- pobieranie definicji klasy z odległego źródła,
- zarządzanie przestrzeniami nazw
  - łatwość wiązania polityki bezpieczeństwa ze źródłem,
  - łatwość separowania kodu (np. separacja przestrzeni nazw dla różnych apletów)
- krytyka wiązania pobierania z przestrzeniami nazw

## Ładowarki klas – rodzaje

- pierwotna ładowarka (zwykle napisana w C),
  - ładowanie standardowej biblioteki
  - ładowanie klas ze ścieżki
  - klasy tak ładowane nie są automatycznie weryfikowane
- obiekty klasy ClassLoader
  - AppletClassLoader (przeglądarka)
  - SecureClassLoader
  - RMIClassLoader
- własne wynalazki (zmieniaj tylko loadClass!)

## Ładowarki klas – algorytm

1. sprawdź, czy klasa już została załadowana, jeśli tak, podaj załadowany kod,
2. sprawdź, czy pierwotna ładowarka może załadować klasę ze ścieżki CLASSPATH (ochrona przed podszywaniem się),
3. sprawdź, czy ładowarce wolno stworzyć ładowaną klasę (decyzję podejmuje zarządca bezpieczeństwa), jeśli nie rzuć wyjątek bezpieczeństwa
4. wczytaj plik klasy do tablicy bajtów (z pliku, bazy danych, z sieci itp.)
5. utwórz obiekt Class na podstawie tablicy bajtów
6. zanim użyjesz klasy rozwiąż klasy do których nowa klasa się bezpośrednio odwołuje (m.in. klasy ze statycznych inicjalizatorów, rozszerzane klasy)
7. sprawdź klasę weryfikatorem

# Zarządca bezpieczeństwa

- opracowywany przez twórcę aplikacji (np. przeglądarki)
- opakowanie operacji czułych na bezpieczeństwo

## Zarządca bezpieczeństwa – przeglądarki

- aplety nie mogą czytać i zapisywać do lokalnych plików
- aplety nie mogą otwierać połączenia klienckiego z maszyną inną niż źródło pochodzenia apletu
- aplety napisane pod JDK 1.1 lub w późniejszej wersji mogą otwierać gniazdo serwera o porcie większym niż zakres uprzywilejowany (zwykle 1024),
- aplety mogą odczytywać tylko 9 własności systemowych (twórca JVM, numer wersji VM, znak oddzielający katalogi, znak końca wiersza itp.)
- aplety ładowane z plików (o ile są poza CLASSPATH) są ładowane przez AppletClassLoader

# Model bezpieczeństwa w Java 2

- możliwość przydzielania przywilejów, gdy są potrzebne
- możliwość udostępniania minimalnych koniecznych przywilejów
- możliwość szczegółowego zarządzania konfiguracją bezpieczeństwa systemu

# Podpisywanie kodu – własności

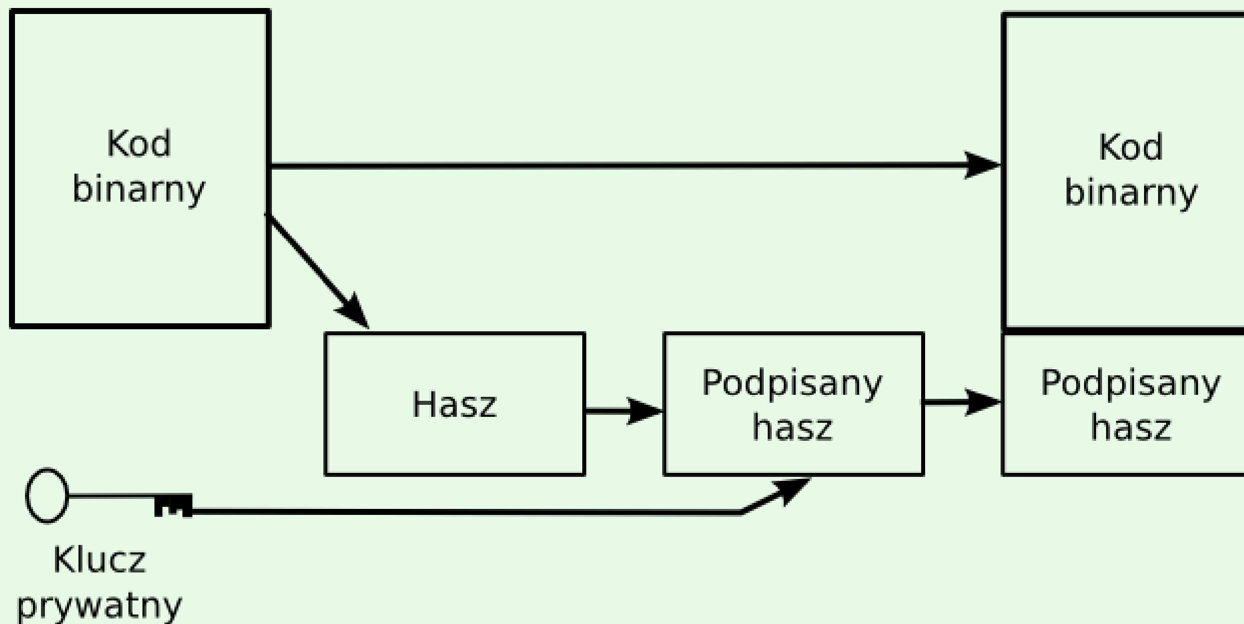
Podpisy muszą być:

- weryfikowalne
- niepodrabialne
- jednorazowego użytku
- niezmienialne
- nie do wyparcia się

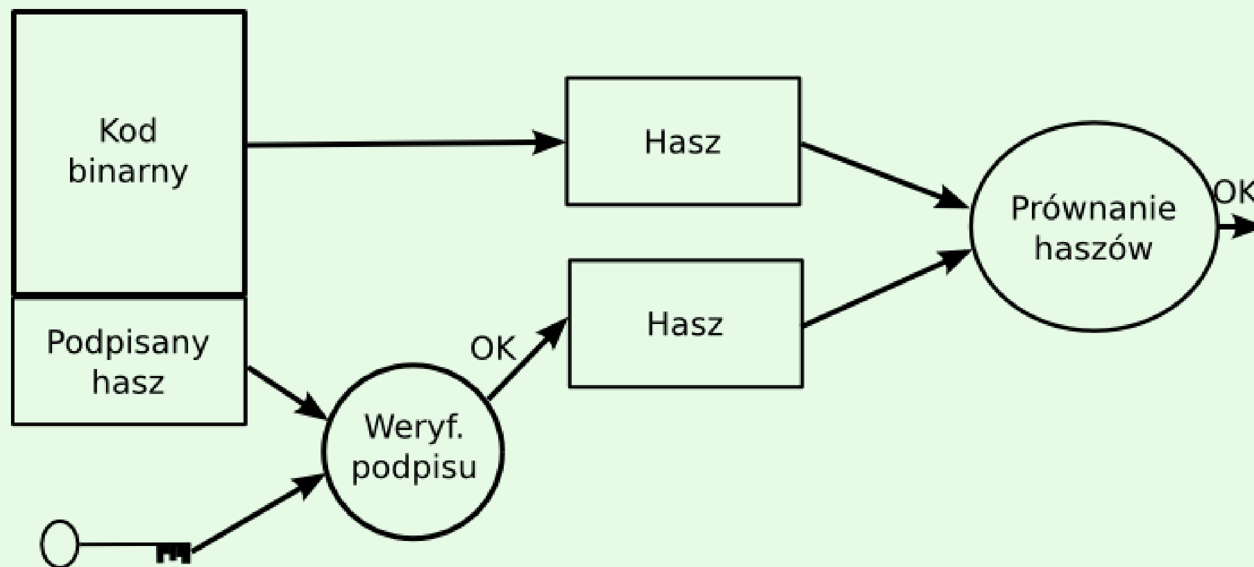
# Podpisywanie kodu – praktyka

- certyfikaty X.509
- urzędy certyfikacyjne (np. VeriSign)
- konieczność sprawdzania autentyczności podpisu
- problemy z unieważnianiem

# Podpisywanie kodu – proces



# Podpisywanie kodu – weryfikacja



# Bezpieczeństwo w Java 2

Własności:

- Drobnoziarniste sterowanie dostępem
- Konfigurowalne zasady bezpieczeństwa
- Rozszerzalna strutura sterowania dostępem
- Sprawdzanie bezpieczeństwa we wszystkich programach

# Bezpieczeństwo w Java 2

- Przydzielanie uprawnień kodu w oparciu o:
  - URL pochodzenia
  - klucze użyte do podpisania kodu
- Domeny zabezpieczenia
  - tworzone na żądanie
  - przypisane na podstawie wartości CodeBase i SignedBy
- Addytywne przyznawanie uprawnień

# Dozorowanie stosu wywołań

- Każda ramka jest oznaczona etykietą mocodawcy
- Cztery operacje związane z dozorowaniem:
  - enablePrivilege()
  - disablePrivilege()
  - checkPrivilege()
  - revertPrivilege()

# Dozorowanie stosu wywołań c.d.

- Algorytm:
  - przeszukujemy ramki od najnowszej do najstarszej,
  - jeśli napotykamy ramkę
    * z zabronionym dostępem do obiektu docelowego w ramach lokalnych zasad bezpieczeństwa, lub
    * z jawnie odebranymi prawami dostępu, to poszukiwanie się kończy z zabronionym dostępem,
  - jeśli docieramy do końca stosu bez przyznania przywilejów, to
    * ich nie przyznajemy (Netscape) lub
    * przyznajemy (Sun/Microsoft).

# Nowe mechanizmy bezpieczeństwa

1. Tożsamość
2. Prawa dostępu
3. Implikowanie praw
4. Definiowanie zasad bezpieczeństwa
5. Przypisywanie zasad bezpieczeństwa

# Nowe mechanizmy bezpieczeństwa c.d.

6. Domeny ochorony
7. Kontrola dostępu
8. Przywileje
9. Zarządca bezpieczeństwa, ładowarka klas, piaskownica
10. Dodawanie praw dostępu

## Tożsamość (ang. identity)

- Pochodzenie i podpis
- Reprezentowane w java.security.CodeSource
- Możliwe wzorce: "gdziekolwiek" (ang. anywhere) i "niepodpisane" ("unsigned")
- Rozszerzenie schematu ładowania klas z JDK 1.1

# Prawa dostępu (ang. permissions)

- Abstrakcyjna klasa java.security.Permission
- Użycie podklas
- Opisują obiekt docelowy i akcję

## Prawa dostępu – wybrane

- w systemie:
  - java.io.FilePermission – dostęp do plików
  - java.net.SocketPermission – dostęp do sieci
  - java.lang.PropertyPermission – własności Javy
  - java.lang.RuntimePermission – dostęp do zasobów systemu wykonania
  - java.net.NetPermission – dostęp do różnych operacji sieciowych
  - java.awt.AWTPermission – dostęp do zasobów graficznych, np. okien

# Implikowanie praw

- Podklasy Permission mają metodę implies(Permission op)
- Daje wartość true, gdy posiadanie obecnego prawa dostępu oznacza prawo dostępu do op
- Uwaga na dziwną konstrukcję

# Definiowanie zasad bezpieczeństwa

- Przeważnie na starcie VM z pliku konfiguracyjnego
- W Sun-owskiej Javie
  - $JAVA_HOME/lib/security/java.policy
  - $HOME/.java.policy
- Także -Djava.policy=<nazwa pliku>
- Przykład:

```
grant CodeBase "https://www.rstcorp.com/users/gem", SignedBy "*" {
   permission java.io.FilePermission "read,write", "/applets/tmp/*";
   permission java.net.SocketPermission "connect", "*.rstcorp.com";
};
```

# Przypisywanie zasad bezpieczeństwa

- Musi się zgadzać
  - prefiks URL-a
  - podpis
- Możliwe wielokrotne podpisy
- Prawa dodawane
  - uwaga: trudne w rozumieniu

# Domeny ochorony

- Związane z ładowarkami klas
- Prawa dostępu przyznawane domenom
- Zasady bezpieczeństwa określają, jakie domeny powinny być utworzone i jakie prawa dostępu im przyznane
- Domena systemowa

# kontrola dostępu

- Klasa java.security.AccessController implementuje algorytm sprawdzania stosu
- Dowolny kod może używać tej klasy
- Sprawdzenie wykonuje metoda checkPermission()
- Brak uprawnień oznacza AccessControlException
- Przykład:

```
FilePermission p = new FilePermission("/tmp/junk", "read");
AccessController.checkPermission(p);
```

## Przyznawanie przywilejów

- Początkowo beginPrivileged i endPrivileged
- Problemy z łapaniem wyjątków i optymalizacją (JIT)
- AccessController.doPrivileged(PrivilegedAction pa)
- Anonimowe klasy PrivilegedAction
- Metoda run dająca w wyniku Object

## Zarządca bezpieczeństwa, ładowarka klas, piaskowni

- Zarządca bezpieczeństwa – kompatybilny z wcześniejszym
- Wprowadzony java.security.SecureClassLoader – ładowanie wszystkiego, poza pierwotną ładowarką
- Można w piaskownicy umieszczać lokalne aplikacje

# Dodawanie praw dostępu

- Tworzymy podklasę java.security.Permission
- Najlepiej w pakiecie, gdzie są stosowane
- Dodanie opisu do pliku z zasadami bezpieczeństwa
- Można dodać w swoim kodzie checkPermission() z klasy AccessController (nie trzeba myśleć o ładowarkach i zarządcach bezpieczeństwa)

## Zagrożenia dla modelu

- Słaba infrastruktura PKI
  - sprawdzanie aktualności kluczy
  - brak powszechności (model finansowania)
- Błędy w implementacji Javy i JVM
- Komplikacja zasad bezpieczeństwa (rozumienie i zarządzanie)