Studia Podyplomowe: Java EE - produkcja oprogramowania

Wybrane Elementy Technologii Java (WET)

Bezpieczeństwo

(do użytku wewnętrznego)

Semestr letni 2020/2021

Grupa: JA20Z (JA2-A, JA2-B) - semestr drugi

Anna Derezińska
A.Derezinska[et]ii.pw.edu.pl
Instytut Informatyki
Politechnika Warszawska

Przykłady kodu zaczerpnięte z materiałów podanych w literaturze

1

Literatura

- Cay S. Horstmann, Gary Cornell: „**Java. Techniki Zaawansowane**."
  Wyd. XI, Helion 2020, ISBN 978-83-283-6066-2
  Bezpieczeństwo (Rozdział 10)
  Wyd. X, Helion 2017, ISBN 978-83-283-3479-3 (Rozdział 9)
  Wyd. IX, Helion SA, 2014, ISBN 978-83-246-7762-7 (Rozdział 9)
- Cay S. Horstmann, Gary Cornell: „**Core Java**
  Volume II — Advanced Features„ (CoreSeries),
  11th Edition © 2019 Oracle and/or its affiliates – **Security** Chapter 10
  10th Edition © 2017, ISBN-13:978-0-13-417786 (Chapter 9)
  9th Edition. © 2013, ISBN 978-0-13-7081608 (Chapter 9)
- Kody źródłowe http://horstmann.com/corejava

2

Skrót wiadomości

- cyfrowy „odcisk palca" danych bloku

Różne algorytmy szyfrujące (kryptograficzne funkcje skrótu):

- SHA – *Secure Hash Algorithm* daje rzadką (choć nie unikalną) sekwencję niezależnie od długości danych wejściowych
- Zmiana jednego bitu w danych -> zmiana skrótu
- Jest wysoce mało prawdopodobne żeby utworzyć sfałszowaną wiadomość o takim samym skrócie
- Warianty SHA-2: SHA-224, SHA-256, SHA-384, SHA-512
- Przekazanie wiadomości i skrótu różnymi drogami – możliwość weryfikacji transmisji
- Ograniczenia związane z użyciem skrótu – przechwycenie wiadomości i możliwość generacji skrótu – algorytmy są znane i NIE wymagają klucza!
- SHA-1, MD5 – złamane, NIE do podpisów cyfrowych

4

Skrót wiadomości w Javie

Biblioteki java.security.*

MessageDigest – fabryka obiektów dostarczających algorytmy,

statyczna metoda *getInstance* zwraca klasę pochodną realizującą zadany algorytm

```
MessageDigest alg = MessageDigest.
getInstance(String <algname>);
```

nazwa algorytmu np. "SHA-224"",

```
alg.update(input); // wprowadza dane
byte[] hash =alg.digest(); // przelicza
```

lub łącznie

```
byte[] hash = alg.digest(input);
```


5

Skrót wiadomości - przykład

Przykład Security.hash.Digest

1) Sprawdzić zawartość pliku *input.txt (plik poniżej JRE Sys Lib!)*

2a) Uruchomić Digest z dwoma parametrami:

input.txt "SHA-224"

lub 2b) uruchomić DigestSHA

3) Utworzyć plik *input2.txt* – np. zmienić wydziedziczonego syna

4) Utworzyć skrót dla zmienionego pliku, porównać wynik

5) Utworzyć skrót dla innych algorytmów:

"SHA-256", "SHA-384", "SHA-512"

Porównać wyniki

8

Szyfrowanie

- Ochrona poufnych informacji – np. numer karty kredytowej
- Zaszyfrowana informacja nie jest dostępna dopóki nie odszyfrujemy
- Generowanie klucza
- Szyfrowanie symetryczne – ten sam klucz do szyfrowania i odszyfrowania

9

Szyfrowanie w Javie

- Algorytmy w standardowej bibliotece
- *Cipher* – klasa bazowa dla algorytmów szyfrujących

```
Cipher cipher =
Cipher.getInstance(algorithm, provider);
```

lub domyślnie dla „SunJCE"

```
Cipher cipher =
Cipher.getInstance(algorithm);
```

algorytmy:

„AES" – Advanced Encryption Standard (nowszy)

„DES/CBC/PKCS5Padding" – Data Encryption Standard

10

10

Inicjacja szyfrowania

- Inicjujemy algorytm szyfrujący - tryb pracy i klucz

```
cipher.init(mode, key);
```

- Tryby pracy:

`Cipher.ENCRYPT_MODE`	zaszyfruj
`Cipher.DECRYPT_MODE`	odszyfruj
`Cipher.WRAP_MODE`	zaszyfruj klucz
`Cipher.UNWRAP_MODE`	odszyfruj klucz

11

11

Generowanie klucza szyfrowania

- Format zgodny z algorytmem
- Klucz losowy – problem prawdziwej losowości

Tworzymy klucz

```
KeyGenerator keygen =
KeyGenerator.getInstance(„AES");
```

inicjujemy generator pseudolosowy (nie klasa *Random*!)

```
SecureRandom random = new SecureRandom();
keygen.init(random);
```

inicjujemy klucz

```
SecretKey key = keygen.generateKey();
```

tworzymy klucz

lub korzystamy z losowych danych

```
byte[] keyData =...
SecretKey key =
      new SecretKeySpec(keyData,"AES");
```

12

12

Szyfrowanie bloku danych

sprawdzamy rozmiar bloku używany przez algorytm szyfrowania, lub 0 gdy nie używa bloków

```
int blockSize = cipher.blockSize();
```

alokujemy blok na dane do szyfrowania

```
byte[] inBytes = new byte[blockSize];
```

pobieramy rozmiar bloku wyjściowego

```
int outputSize =
cipher.getOutputSize(blockSize);
```

alokujemy blok na wyniki szyfrowania

```
byte[] outBytes = new byte[outputSize];
```

szyfrujemy bloki danych:

```
int outLenght = cipher.update(inBytes, 0,
outputSize, outBytes);
```

13

13

Dopełnienie ostatniego bloku

```
outBytes =
cipher.doFinal(inBytes,0,inLenght);
```

lub `cipher.doFinal();`

dopełnienie ostatniego bloku

gdy algorytmy korzystają z bloków 8-mio bajtowych

PKCS#5 – algorytm dopełniania

Przy odszyfrowywaniu – ostatni bajt to liczba bajtów dopełnienia (do usunięcia)

14

14

Szyfrowanie symetryczne - ćwiczenie

Zrobić 3 kopie programu *aes.AESTest*

W każdej kopii wybrać fragment kodu dla danego parametru, oraz wpisać odpowiednie nazwy plików lub wybrać parametry

1) Generujemy plik z kluczem dla AES

program *aes.AESTest* z parametrem i nazwą pliku na klucz:

–genkey <keyfile>

2) Szyfrujemy plik, używamy ten sam program z parametrem i nazwami 3 plików:

-encrypt <inputfile> <encryptedfile> <keyfile>

3) Odszyfrowujemy plik

-decrypt <encryptedfile> <outputfile> <keyfile>

Security *aes.AESTest*

16

16

Strumienie szyfrujące

- Automatyzacja szyfrowanie/deszyfracji – biblioteka JCE
- Nie ma potrzeby korzystania z metod *update* i *doFinal*

```
cipher.init(Cipher.ENCRYPT_MODE, key);
CipherOutStram out = new CipherOutStream(
 new FileOutputStream(outputFileName), cipher);
byte[] bytes = new Byte{BLOCKSIZE];
int inLength = getData(bytes); //pobiera dane
while (inLenght != -1)
{
      out.write(bytes, 0, inLenght);
      inLength = getData(bytes); // kolejne dane
}
out.flush(); //wyprowadza bufor, dopełnia jeśli potrzeba
```

17

17

Szyfrowanie z kluczem publicznym

- Szyfrowanie symetryczne – problem bezpiecznego przekazywania klucza
- Szyfrowanie z parą kluczy – prywatnym i publicznym
- Algorytmy szyfrujące z kluczem publicznym **wolniejsze** niż z kluczem symetrycznym
- RSA – algorytm Rivesta, Shamira, Adelmana

18

18

Schemat przesyłania M od A do B

1. B tworzy parę kluczy prywatny MyB i publiczny KeyB, przesyła klucz publiczny do A
2. A tworzy klucz Key1 dla algorytmu symetrycznego i szyfruje wiadomość, M->DM
3. A szyfruje klucz symetryczny Key1 korzystając z klucza publicznego KeyB od B , Key1->DKey1
4. A przesyła do B zaszyfrowaną wiadomość DM oraz zaszyfrowany klucz DKey1
5. B odszyfrowuje klucz symetryczny DKey1 za pomocą swojego klucza prywatnego MyB, DKey1->Key1
6. B odszyfrowuje wiadomość od A korzystając z odszyfrowanego klucza symetrycznego, DM->M

19

19

Tworzenie pary kluczy w Javie

```
keyPairGenerator pairgen =
  KeyPAirGenerator.getInstance(„RSA”);
SecureRandom  random = new
     SecureRandom();
pairgen.initialise(KEYSIZE, random);
```

generacja pary kluczy

```
KeyPAir keyPair =
pairgen.generateKeyPAir();
Key publicKey = keyPair.getPublic();
Key privateKey = keyPair.getPrivate();
```

20

20

Szyfrowanie asymetryczne - ćwiczenie

Zrobić 3 kopie programu *rsa.RSATest*

W każdej kopii wybrać fragment kodu dla danego parametru, oaz wpisać odpowiednie nazwy plików lub wybrać parametry

1) Generujemy 2 pliki z kluczami dla RSA

Program *rsa.RSATest* z parametrem i nazwami plików na klucz publiczny i prywatny:

–genkey <publickeyfile> <privatekeyfile>

2) Generujemy klucz dla AES i szyfrujemy kluczem publicznym . Tworzy plik zawierający: długość zaszyfrowanego klucza, zaszyfrowany klucz, plik zaszyfrowany przez AES

-encrypt <inputfile> <encryptedfile> <publickeyfile>

3) Odszyfrowujemy plik

-decrypt <encryptedfile> <outputfile> <privatekeyfile>

Security *rse.RSATest*

23

23

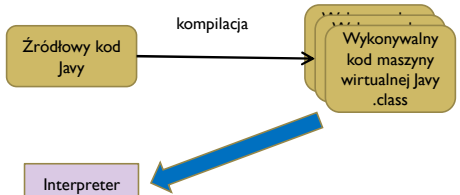
Uruchamianie bezpiecznego kodu

1. Ładowanie klas
2. Weryfikacja kodu maszyny wirtualnej
3. Menedżer bezpieczeństwa

24

24

Ładowanie klas – niezbędne klasy



- ładowanie wskazanej klasy
- ładowanie klas wg relacji zależności (klasy bazowe, typy obiektów w klasie, ..)
- ładowanie klas na żądanie podczas wykonania programu

25

25

Kolejność ładowania

1. Klasy systemowe (*rt.jar*, część maszyny wirtualnej)
2. Standardowe rozszerzenia maszyny wirtualnej (*jre/lib/ext*)
3. Klasy aplikacji

Procedury ładowania klas 2) i 3) są instancjami klasy *URLClassLoader*

Zwykle automatycznie, ale można zmienić na własną procedurę ładowania

26

26

Ładowanie wątków

Każdy wątek ma referencję procedury ładującej (kontekstu)
Wątek główny – systemowa procedura ładująca
(szuka plików wg zmiennej CLASSPATH lub opcji *–classpath)*
inne wątki – procedura wątku nadrzędnego, lub własna

Zmiana procedury ładującej:

```
Thread t = Thread.currentThread();
t.setContextClassLoader(loader);
```

gdzie *loader* – nasza nowa procedura ładująca
i ładowanie klasy

```
Class cl = loader.loadClass(className);
```

Identyfikacja klasy w maszynie wirtualnej
– pełna nazwa pakietowa i procedura ładująca

27

27

Własne procedury ładujące

Możliwość działań przed załadowaniem klasy, np. związanych z bezpieczeństwem
- kontrola pochodzenia klasy
- sprawdzanie uprawnień

Rozszerzamy klasę *ClassLoader* i zastępujemy metodę *findClass(String className)*
- robi co chce, np. odszyfrowuje kod
- ładuje kod klasy (skompilowany)
- przekazuje kod do MV – metoda *defineClass*

28

28

Weryfikacja kodu maszyny wirtualnej

- inicjalizacja zmiennych
- wywołania metod zgodne z typami referencji obiektów
- dostęp do prywatnych składowych i metod
- stos dla zmiennych lokalnych
- brak przepełnienia stosu

Kod maszyny wirtualnej może nie pochodzić z kompilatora Javy, był modyfikowany przez inne programy. Możliwe działanie wirusów, i ...

29

29

Weryfikacja kodu I - ćwiczenie

Na poziomie kodu źródłowego Javy
VerifierTest, w funkcji fun() zmienić n=2 na m=2, czyli

```
23 public static int fun()
24  {
25     int m;
26     int n;
27     m = 1;
28     n = 2;                m = 2;
29     int r = m + n;
```

Błąd kompilacji, bo użycie zmiennej `n` bez inicjalizacji
Przywrócić poprawny kod.

CoreJava10SecurityCh9. verifier.VerifierTest

31

31

Analiza kodu maszyny wirtualnej

Otworzyć zakładkę *Navigator* np. *Window->Show View->*General->
Odtworzyć plik z kodem np. *Navigator->bin->VerifierTest.class*
Korzystamy z *Class File Viewer*

```
23 public static int fun()
24  {
25     int m;
26     int n;
27     m = 1;
28     n = 2;
29     int r = m + n;
....
```

```
public static int fun() {
    /* L27 */
    0 iconst_1;              04
    1 istore_0;    /* m */   3B
    /* L28 */
    2 iconst_2;              05
    3 istore_1;    /* n */   3C
    /* L29 */
    4 iload_0;     /* m */   1A
    5 iload_1;     /* n */   1B
    6 iadd;                  60
    7 istore_2;    /* r */   3D
```


32

Modyfikacja kodu maszyny wirtualnej

- Edytory Hexadecymalne (szestnastkowe)
- Dodatki do Eclipse (Winows, Linux), np.:
  - Eclipse Hex Editor Plugin (EHEP)
  - Java Hex Editor
  Navigator-> „name".class-> Open with np. HexEditor
- Linux ma wbudowany edytor szestnastkowy
/usr/local/wxHexEditor/wxHexEditor

33

Weryfikacja kodu2 - ćwiczenie

Na poziomie ByteCodu Javy
VerifierTest, w funkcji fun zmienić n=2 na m=2, czyli

```
1 istore_0;     /* m */    3B
2 iconst_2;                05
3 istore_1;     /* n */    3C      3 istore_1;     /* m */    3B
```

Otworzyć plik VerifierTest.class (ByteCode klasy) za pomocą edytora szesnastkowego
np. wxHexEditor (Linux)
Znaleźć sekwencję:
043B053**C**1A1B603D1CAC
Zmienić kod, 3C na 3B

34

Weryfikacja kodu 3 - ćwiczenie

- Uruchomić *VerifierTest.class* – może być błąd

Location: verifier/copy/VerifierTest.fun()I @5: iload_1
Reason: Type top (current frame, locals[1]) is not assignable to integer
Bytecode: 0x0000000: 043b 053b 1a1b 603d 1cac

- Uruchomić bez weryfikacji

np. java –noverify verifier.VerifierTest
argument dla Maszyny Virtualnej
(*Run as- > Run Configurations -> Arguments -> VM arguments ->* -noverify)
Program się wykona, ale wyświetli zmieniony wynik

lub -Xverify:none

35

Menedżer bezpieczeństwa

Weryfikuje kod załadowany do MV (restrykcje na potencjalnie niebezpieczne operacje) np.:
- nowa procedura ładująca
- zatrzymanie MV
- dostęp do składowej innej klasy przez refleksję
- dostęp do pliku
- dostęp do schowka systemowego
- otwarcie połączenia przez gniazdko sieciowe,
...

36

Menedżer bezpieczeństwa - działanie

- Standardowo – brak MB
- Można
  - skonfigurować domyślny MB
  - zaimplementować własny (kłopotliwe)

  Przykład użycia -– *appletviewer* do uruchamiania apletów instaluje własny MB

  Zalecane – odwoływać się do MB w usługach systemowych standardowej biblioteki

37

Pozwolenia (ang. *Permissions*)

- Pozwolenie – właściwość kontrolowana przez MB
- Typy pozwoleń - hierarchia klas hermetyzujących pozwolenia

FilePermission p =
new FilePermission("/tmp/*", "read,write");
Pozwolenie na czytanie/pisanie dowolnego pliku z katalogu /tmp
Można tworzyć też własne klasy pozwoleń

Sprawdzanie pozwolenia wykonania operacji –
sprawdzenie wszystkich metod ze stosu wykonań
Przy braku uprawnień *SecurityException*

38

38

Polityka bezpieczeństwa

Pliki polityki – przyporządkowanie pozwoleń do źródeł kodu
permission java.io.FilePermission "/tmp/*", "read,write";
permission java.io.FilePermission "${user.home}", "read,write"; pliki z katalogu użytkownika

Składnia:
permission <className> <targetName>, <actionList>;
className – pełna nazwa klasy pozwolenia
targetName – nazwa podmiotu pozwolenia, np. pliku, hosta, numer portu
actionList – lista akcji

39

39

Położenie plików polityki bezpieczeństwa

Zbiór globalnych reguł
java.policy
zwykle w podkatalogu jre/lib/security
Np. na Linux /usr/java/jdk…/jre/lib/security

.java.policy w katalogu domowym użytkownika

Położenie tych plików i inne konfiguracje w pliku jre/lib/security/java.security

40

40

Plik polityki bezpieczeństwa dla aplikacji

plik MyAppl.policy
Użycie pliku polityki (oraz globalnych polityk):
1) Ustawić w metodzie *main()*
*System.setProperty("java.security.policy","*MyAppl.policy");
lub
2) Uruchomić maszynę wirtualną
java -Djava.security.policy=MyAppl.policy MyAppl

Dla apletu
appletviewer –J-Djava.security.policy =
MyApplet.policy MyApplet.html

41

41

Instalacja menedżera bezpieczeństwa

MB potrzebny żeby skorzystać z polityki bezpieczeństwa
1) W metodzie *main()*
*System.setSecurityManager(**new** SecurityManager());*
lub
2) Uruchomić maszynę wirtualną
java -Djava.security.manager -Djava.security.policy =
MyAppl.policy MyAppl

42

42

Praca końcowa - aplikacja

- Min 2 wątki przekazujące zaszyfrowaną informację
- Operacje szyfrowania i odszyfrowania
- Ochrona współdzielonych danych

(locks, synchronized, niepodzielne metody, atomic,..)

- Synchronizacja informacji o gotowości zaszyfrowanych danych (pliki, obiekty)

Można korzystać z ustawiania flag w sekcji krytycznej, sprawdzania warunków i czekania, przekazywania informacji przez współbieżne struktury – Kolejka Blokująca, Współbieżna Hashmap, ...

46

46