Обфускатор для программ на языке Java

**Текст программы**

2017

#### АННОТАЦИЯ

В данном документе приведен текст программы «Обфускатор для программ на языке Java», представляющей из себя программную реализацию обфускатора для программ на языке Java.

Исходным языком, используемым для данной разработки, является Java. Результатом компиляции и сборки исходных кодов является jar файл, готовый к исполнению.

**Оглавление**

[1. Текст программы 4](#_Toc484875895)

1. Текст программы

Листинг 1: Файл Obfuscator.java

|  |
| --- |
| package obfuscator;  import obfuscator.bcel.UseBCEL;  import java.io.BufferedReader;  import java.io.BufferedWriter;  import java.io.File;  import java.io.FileInputStream;  import java.io.FileNotFoundException;  import java.io.FileOutputStream;  import java.io.IOException;  import java.io.InputStreamReader;  import java.io.OutputStreamWriter;  import java.io.PrintStream;  import java.nio.file.Files;  import java.nio.file.Paths;  import java.nio.file.Path;  import java.security.SecureRandom;  import java.util.Arrays;  import java.util.Random;  import java.util.logging.Level;  import java.util.logging.Logger;  import java.util.List;  import java.util.ArrayList;  import static java.nio.file.StandardCopyOption.\*;  public class Obfuscator {  public static void main (String[] args) throws IOException {  List<String> jarPaths = new ArrayList<String>();  for (int i = 0; i < args.length; i++) {  String fileName = args[i];  //System.out.println(args[i]);  if (fileName.endsWith(".class") || fileName.endsWith(".jar")) {  jarPaths.add(args[i]);  }  else if (!fileName.endsWith(".java")) {  Files.walk(Paths.get(fileName)).forEach(filePath -> {  String name = filePath.toString();  if (name.endsWith(".class") || name.endsWith(".jar")) {  jarPaths.add(name);  }  });  }  }    if ((jarPaths.isEmpty() && args.length < 2)) {  System.out.println("You must specify input folder with .java files and output file!");  System.out.println("Or jar file, or class files separated by space!");  return;  }  if (!jarPaths.isEmpty()) {  UseBCEL bcel = new UseBCEL();  String []jarPath = new String[jarPaths.size()];  jarPaths.toArray(jarPath);  bcel.bcel(jarPath);  }  else {  String folder = args[0];  String output = args[1];  String tempFile = "temp.java";    Path fileToDeletePath = Paths.get(output);  Files.deleteIfExists(fileToDeletePath);  fileToDeletePath = Paths.get(tempFile);  Files.deleteIfExists(fileToDeletePath);    FindFunction ff = new FindFunction();  Utils ut = new Utils();  WriteAllInOneFile writefile = new WriteAllInOneFile();  writefile.writeImports(folder, output);  //writefile.writeMainFile(mainFile, tempFile);  writefile.findClass(folder, output);  try {  ff.writeChangeFunctionsName(output, tempFile);  ut.DeleteComments(tempFile, output);  ut.replaceStrings(output, tempFile);  ut.deleteLineBreaks(tempFile, output);    } catch (IOException ex) {  Logger.getLogger(Obfuscator.class.getName()).log(Level.SEVERE, null, ex);  }  }  }  } |

Листинг 2: Файл FindFunction.java

|  |
| --- |
| package obfuscator;  import java.io.BufferedReader;  import java.io.BufferedWriter;  import java.io.File;  import java.io.FileInputStream;  import java.io.FileNotFoundException;  import java.io.FileOutputStream;  import java.io.IOException;  import java.io.InputStreamReader;  import java.io.OutputStreamWriter;  import java.security.SecureRandom;  import java.util.Arrays;  import java.util.HashMap;  import java.util.Map;  import java.util.logging.Level;  import java.util.logging.Logger;  // Class to find and rename function  public class FindFunction {  private String types[] = {"String", "void", "int", "double", "float", "char", "boolean"};  private String type;  private int functionNameLength = 15;  Map<String, String> hashmap = new HashMap<String, String>();  public void find(String fileInput, String fileOutput) throws IOException {  FileInputStream fis;  try {  fis = new FileInputStream(fileInput);    FileOutputStream fos = new FileOutputStream(fileOutput, true);  BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(fos));  BufferedReader br = new BufferedReader(new InputStreamReader(fis));  String line = null;  String replace = null;  while ((line = br.readLine()) != null) {  replace = line;  if (isLineWithType(line)) {  if (isLineWithFunction(line)) {  //System.out.println("Potential function: " + line);  //replace = line.replaceAll(getFunctionName(line, this.type),  // generateRandomString(this.functionNameLength));  String functionName = getFunctionName(line, this.type);  if (!functionName.equals("toString") && !functionName.equals("main") && functionName.length() > 0) {  hashmap.put(functionName, generateRandomString(this.functionNameLength));  }    }  }  }  bw.close();  br.close();  } catch (FileNotFoundException ex) {  Logger.getLogger(FindFunction.class.getName()).log(Level.SEVERE, null, ex);  }  }  public void writeChangeFunctionsName(String fileInput,String fileOutput) throws IOException {  find(fileInput, fileOutput);    FileInputStream fis;  try {  fis = new FileInputStream(fileInput);    FileOutputStream fos = new FileOutputStream(fileOutput, true);  BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(fos));  BufferedReader br = new BufferedReader(new InputStreamReader(fis));  String line = null;      while ((line = br.readLine()) != null) {  int max = 0;  String replace = null;  for (Map.Entry entry : hashmap.entrySet()) {  if (line.contains(entry.getKey().toString())) {  if (entry.getKey().toString().length() >= max) {  max = entry.getKey().toString().length();  replace = line.replaceAll(entry.getKey().toString(), entry.getValue().toString());  }  }  //line = line.replaceAll(entry.getKey().toString(), entry.getValue().toString());  }  if (replace == null) {  replace = line;  }  bw.write(replace);  bw.newLine();  }  bw.close();  br.close();  } catch (FileNotFoundException ex) {  Logger.getLogger(FindFunction.class.getName()).log(Level.SEVERE, null, ex);  }  }    private boolean isLineWithType(String str) {  int min = str.indexOf(types[0]);  int k = -1;  //int index = str.indexOf(types[0]);  for (int i = 0; i < types.length; i++) {  Boolean found = Arrays.asList(str.split(" ")).contains(types[i]);  if (found) {  //setType(types[i]);  for (int j = 1; j< types.length; j++) {  int index = str.indexOf(types[j]);  if (index > -1 && index < min) {  min = index;  k = j;  }  }  if (k > 0) {  setType(types[k]);  }  else {  setType(types[i]);  }  return true;  }  }  return false;  }  private boolean isLineWithFunction(String line) {  if (line.contains("(") && line.contains(")") && !line.contains("=")) {  return true;  }  return false;  }  private String getFunctionName(String line, String type) {  int ind1 = line.indexOf(type) + type.length() +1;  int ind2 = line.lastIndexOf(" (");  if (ind2 == -1) {  ind2 = line.lastIndexOf("(");  }    //System.out.println("type: " + type + ", ind1: " + ind1 + ", ind2: " + ind2);    if (ind2 == -1 || ind1 > ind2) {  return "";  }  else {  //System.out.println(line.substring(ind1,ind2));  return line.substring(ind1,ind2 );  }  }  private void setType(String type) {  this.type = type;  }  public static String generateRandomString(int len) {  String AB = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";  SecureRandom rnd = new SecureRandom();  StringBuilder sb = new StringBuilder(len);  sb.append(" ");  for (int i = 0; i < len; i++) {  sb.append(AB.charAt(rnd.nextInt(AB.length())));  }  sb.append(" ");  return sb.toString();  }  } |

Листинг 3: Файл GenerateFalseCode.java

|  |
| --- |
| package obfuscator;  import java.security.SecureRandom;  import java.util.Random;  import java.util.logging.Level;  import java.util.logging.Logger;  import javax.script.ScriptEngine;  import javax.script.ScriptEngineManager;  import javax.script.ScriptException;  public class GenerateFalseCode {  public String potentialTypes[] = {"int", "float", "double"};  public String arith\_symbols[] = {"+", "-", "/", "\*", "%"};  public String createWrongCode() {  FindFunction gencode\_ffunc = new FindFunction();  Random rand = new Random();  String str;  str = " {\n";  for (int i = 0; i < 5; i++) {  String randVar = gencode\_ffunc.generateRandomString(5).toString();  str += this.potentialTypes[rand.nextInt(2)].toString() + " " + randVar;  str += "; \n";  str += randVar + " = " + rand.nextInt() + this.arith\_symbols[rand.nextInt(this.arith\_symbols.length - 1)];  str += " " + rand.nextInt() + ";\n";  }  str += "}";  return str;  }    public String number(){  Random r = new Random();  return ""+(r.nextInt(40-1)+1);  }  public String operator(){  String signs[] = {"+","-","%","\*"};  return ""+(signs[new Random().nextInt(signs.length)]);  }  public String level(int n){  if(n==1) return number() +  operator() +  number();  else return level(n-1) +  operator() +  number();  }    public String createIfStatement() {  String str = "";  Random rand = new Random();  double num = 0;  String exp = level(rand.nextInt(5)+1);  str = "int x = "+rand.nextInt(100)+";\n";  str += "if( "+ implictPredicate(rand.nextInt(100))+" )";  /\*try {  num = countMathExpression(exp)-rand.nextInt(100)-1;  } catch (ScriptException ex) {  System.out.println(ex.getMessage());  }  str+=" > "+num+" )";\*/  str+=createWrongCode();  //System.out.println(str);  return str;  }  public double countMathExpression(String exp) throws ScriptException {  ScriptEngineManager mgr = new ScriptEngineManager();  ScriptEngine engine = mgr.getEngineByName("JavaScript");    return Double.parseDouble(engine.eval(exp).toString());  }    public String implictPredicate(int num) {  String predicates[] = {"~x != x",  "(x + x & 1) == 0",  "(x + -x & 1) == 0",  "(-x & 1) == (x & 1)",  "((-x ^ x) & 1) == 0",  "(x \* 0x80 & 0x56) == 0",  "(x << 1 ^ 0x1765) != 0",  "~(-x \* 0x40) != x << 6",  "(~(x \* 0x80) & 0x3d) == 0x3d",  "x - 0x9d227fa9 != x - 0x699c945e",  "(x ^ x - 0x35f5f4d2) != 0x42a26409",  "(x \* 0x20000000 & 0x19a27923) == 0",  "(x \* 4 & 8) == (x + x \* 3 - 0x1fef9d8f & 8)",  "(x | 0xffffdbe8) - 0x1baa != x || (x & 0x10) == 0x10",  "(x ^ 0x1145f) != 0 || (x | 0xfffeffff) == 0xffffffff"  };  return predicates[num%predicates.length];  }  } |

Листинг 4: Файл Utils.java

|  |
| --- |
| package obfuscator;  import java.io.BufferedReader;  import java.io.BufferedWriter;  import java.io.File;  import java.io.FileInputStream;  import java.io.FileNotFoundException;  import java.io.FileOutputStream;  import java.io.FileWriter;  import java.io.IOException;  import java.io.InputStreamReader;  import java.io.OutputStreamWriter;  import java.nio.ByteBuffer;  import java.nio.file.Files;  import java.util.Map;  import java.util.Scanner;  import java.util.logging.Level;  import java.util.logging.Logger;  import java.math.BigInteger;  import java.nio.file.Path;  import java.nio.file.Paths;  public class Utils {  public void DeleteComments(String input, String output) throws IOException {  BufferedReader reader = new BufferedReader(new InputStreamReader(new FileInputStream(input)));  BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(new FileOutputStream(output)));  boolean inBlockComment = false;  boolean inSlashSlashComment = false;  int char1 = reader.read();  if (char1 != -1) {  int char2;  while (char1 != -1) {  if ((char2 = reader.read()) == -1) {  writer.write(char1);  break;  }  if (char1 == '/' && char2 == '\*') {  inBlockComment = true;  char1 = reader.read();  continue;  } else if (char1 == '\*' && char2 == '/') {  inBlockComment = false;  char1 = reader.read();  continue;  } else if (char1 == '/' && char2 == '/' && !inBlockComment) {  inSlashSlashComment = true;  char1 = reader.read();  continue;  }  if (inBlockComment) {  char1 = char2;  continue;  }  if (inSlashSlashComment) {  if (char2 == '\n') {  inSlashSlashComment = false;  writer.write(char2);  char1 = reader.read();  continue;  } else if (char1 == '\n') {  inSlashSlashComment = false;  writer.write(char1);  char1 = char2;  continue;  } else {  char1 = reader.read();  continue;  }  }  writer.write(char1);  char1 = char2;  }  writer.flush();  writer.close();  }  }  public void deleteLineBreaks(String input, String output) throws IOException {  BufferedReader reader = new BufferedReader(new InputStreamReader(new FileInputStream(input)));  BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(new FileOutputStream(output)));  String content = new Scanner(new File(input)).useDelimiter("\\Z").next();  System.out.println(content);  content = content.replaceAll("\n", "").replaceAll("\r", "");  File file = new File(output);  FileWriter fileWriter = new FileWriter(file);  fileWriter.write(content);  fileWriter.flush();  fileWriter.close();  }  // Delete spaces at begin and end of string and replace string in quotes with their hex representation  public void replaceStrings(String input, String output) {  try {  FileInputStream fis = new FileInputStream(input);  FileOutputStream fos = new FileOutputStream(output);  BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(fos));  BufferedReader br = new BufferedReader(new InputStreamReader(fis));  String line = null;  try {  while ((line = br.readLine()) != null) {  int start = 0;  int end\_ = 0;  while (true) {  start = line.indexOf("\"", start);  if (start == -1) break;  end\_ = line.indexOf("\"", start + 1);  if (end\_ == -1) {  //System.out.println("No end of string, exiting");  break;  }  String replace = "G12365Ghgd4367.fshd1213(\"" + String.format("%x", new BigInteger(1, line.substring(start + 1, end\_).getBytes())) + "\")";  StringBuilder buf = new StringBuilder(line);  buf.replace(start, end\_ + 1, replace);  line = buf.toString();  start = start + 2 + replace.length();  }  //System.out.println(line);  bw.write(line.trim());  bw.newLine();  }  } catch (Exception ex) {  Logger.getLogger(Utils.class.getName()).log(Level.SEVERE, "Problem with replacing strings!", ex);  }  bw.write("class G12365Ghgd4367 {public static String fshd1213(String a){StringBuilder b = new StringBuilder();StringBuilder t = new StringBuilder();for( int i=0; i<a.length()-1; i+=2 ){String o = a.substring(i, (i + 2));int d = Integer.parseInt(o, 16);b.append((char)d);t.append(d);}return b.toString();}}");  bw.newLine();  bw.close();  br.close();  fis.close();  fos.close();  } catch (FileNotFoundException ex) {  Logger.getLogger(Utils.class.getName()).log(Level.SEVERE, "File not found!", ex);  } catch (IOException ex) {  Logger.getLogger(Utils.class.getName()).log(Level.SEVERE, "Error!", ex);  }  }  public String convertStringToHex(String str){  char[] chars = str.toCharArray();  StringBuffer hex = new StringBuffer();  for(int i = 0; i < chars.length; i++){  hex.append(Integer.toHexString((int)chars[i]));  }  return hex.toString();  }  public String convertHexToString(String hex){  StringBuilder sb = new StringBuilder();  StringBuilder temp = new StringBuilder();  //49204c6f7665204a617661 split into two characters 49, 20, 4c...  for( int i=0; i<hex.length()-1; i+=2 ){  //grab the hex in pairs  String output = hex.substring(i, (i + 2));  //convert hex to decimal  int decimal = Integer.parseInt(output, 16);  //convert the decimal to character  sb.append((char)decimal);  temp.append(decimal);  }  //System.out.println("Decimal : " + temp.toString());  return sb.toString();  }  byte[] toByteArray(int value) {  return ByteBuffer.allocate(4).putInt(value).array();  }  int fromByteArray(byte[] bytes) {  return ByteBuffer.wrap(bytes).getInt();  }  } |

Листинг 5: Файл WriteAllInOneFile.java

|  |
| --- |
| package obfuscator;  import java.io.BufferedReader;  import java.io.BufferedWriter;  import java.io.FileInputStream;  import java.io.FileNotFoundException;  import java.io.FileOutputStream;  import java.io.IOException;  import java.io.InputStreamReader;  import java.io.OutputStreamWriter;  import java.nio.file.Files;  import java.nio.file.Paths;  import java.util.Random;  import java.util.logging.Level;  import java.util.logging.Logger;  import java.io.File;  import java.util.concurrent.atomic.AtomicInteger;  public class WriteAllInOneFile {  public String imports[];  public void findClass(String path, String fileOutput) throws IOException {  Files.walk(Paths.get(path)).forEach(filePath -> {  System.out.println(filePath.toString());  if (!filePath.toString().contains("Obfuscator.java")) {  FileInputStream fis;  try {  fis = new FileInputStream(filePath.toString());  boolean isWrite = false;  FileOutputStream fos = new FileOutputStream(fileOutput, true);  BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(fos));  BufferedReader br = new BufferedReader(new InputStreamReader(fis));  String line = null;  int openBrackets = 0, closeBrackets = 0;  boolean alreadyPasteFalseCode = false;  int inFunc = 0;  while ((line = br.readLine()) != null) {  if (line.contains("class") && line.contains("{")  && (line.contains("public") || line.contains("private")|| line.contains("protected"))  &&!line.contains("(")) {  line = line.substring(line.indexOf("class") );  isWrite = true;  inFunc = 0;  }  if (inFunc == 0 && (line.contains("public") || line.contains("private")|| line.contains("protected"))  && ((line.contains("void") || line.contains("int")|| line.contains("String")|| line.contains("Double")|| line.contains("Float")))) {  alreadyPasteFalseCode = false;  inFunc = 1;  openBrackets = 0;  closeBrackets = 0;  }  if (inFunc == 1) {  openBrackets += line.chars().filter(num -> num == '{').count();  closeBrackets += line.chars().filter(num -> num == '}').count();  }  if (openBrackets == closeBrackets) {  inFunc = 0;  }  if (isWrite) {  bw.write(line);  bw.newLine();  Random rnd = new Random();  if (inFunc ==1 && !alreadyPasteFalseCode && !line.contains(" return ") && rnd.nextInt(5)==3) {  //System.out.println("Previous line: " + line);  GenerateFalseCode fc = new GenerateFalseCode();  bw.write(fc.createIfStatement());  bw.newLine();  alreadyPasteFalseCode = true;  }  }  }  bw.close();  br.close();  fis.close();  fos.close();  } catch (Exception ex) {  Logger.getLogger(WriteAllInOneFile.class.getName()).log(Level.SEVERE, null, ex);  }  }  });  }  public void writeImports(String path, String fileOutput) throws IOException {  String[] pkg = new String[1];  AtomicInteger isUsedBufferedReader = new AtomicInteger();  isUsedBufferedReader.set(0);  FileOutputStream fos = new FileOutputStream(fileOutput, true);  BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(fos));    Files.walk(Paths.get(path)).forEach(filePath -> {  FileInputStream fis;  try {  fis = new FileInputStream(filePath.toString());  boolean isWrite = false;  BufferedReader br = new BufferedReader(new InputStreamReader(fis));  String line = null;  while ((line = br.readLine()) != null) {  if (line.contains("import ") && !line.contains("(")) {  bw.write(line);  bw.newLine();  }  if (line.contains("package")) {  pkg[0] = line + "\n";  }  }    br.close();  fis.close();  } catch (Exception e) {  }  });    /\*  bw.write("import java.nio.ByteBuffer;");  bw.newLine();  \*/    String line = null;  bw.close();  fos.close();    // Write package at first line  if (!pkg[0].isEmpty())  {  File mFile = new File(fileOutput);  FileInputStream fis = new FileInputStream(mFile);  BufferedReader br = new BufferedReader(new InputStreamReader(fis));  String result = "";  line = "";  while( (line = br.readLine()) != null){  result = result + line + "\n";  }  result = pkg[0] + result;  mFile.delete();  fos = new FileOutputStream(mFile);  fos.write(result.getBytes());  fos.flush();  }  }  public void writeMainFile(String fileInput, String fileOutput) throws IOException {  FileInputStream fis;  fis = new FileInputStream(fileInput);  boolean isWrite = false;  FileOutputStream fos = new FileOutputStream(fileOutput, true);  BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(fos));  BufferedReader br = new BufferedReader(new InputStreamReader(fis));  String line = null;  while ((line = br.readLine()) != null) {  if (!line.contains("package"))  {  bw.write(line);  bw.newLine();  }  }  bw.close();  br.close();  fis.close();  fos.close();  }  } |

Листинг 6: Файл NameGenerator.java

|  |
| --- |
| package obfuscator.bcel;  /\*\*  \* Generates names for classes, fields and methods.  \*/  public class NameGenerator{  private static int currentIndex;  private static final String[] NAME\_TABLE = new String[] { "a", "b", "c",  "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p",  "q", "r", "s", "t", "u", "v", "w", "x", "y", "z", "A", "B", "C",  "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "O", "P",  "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z" };  public NameGenerator() {  //reset();  }  protected static String getName(int index) {  final int length = NAME\_TABLE.length;  int letters = index / length;  if (letters == 0) {  return NAME\_TABLE[index];  } else {  // we shall assume there are no more than 52^2 + 52 names that need  // generating  StringBuilder sb = new StringBuilder();  for (int i = 0; i < length; i++) {  if ((index - length - length \* i) >= 0  && (index - length - length \* i) < length) {  sb.append(NAME\_TABLE[i]);  int secondIndex = (index - length)  - (length \* i);  sb.append(NAME\_TABLE[secondIndex]);  }  }  return sb.toString();  }  }  public void reset() {  currentIndex = 0;  }  public String next() {  return getName(currentIndex++);  }  public String current() {  return currentIndex == 0 ? null : getName(currentIndex - 1);  }  } |

Листинг 7: Файл UseBCEL.java

|  |
| --- |
| package obfuscator.bcel;  import java.io.File;  import java.io.FileNotFoundException;  import java.io.FileOutputStream;  import java.io.IOException;  import java.io.InputStream;  import java.io.PrintWriter;  import java.nio.file.Files;  import java.nio.file.Path;  import java.nio.file.Paths;  import java.text.SimpleDateFormat;  import java.util.ArrayList;  import java.util.Map;  import java.util.HashMap;  import java.util.Date;  import java.util.Enumeration;  import java.util.List;  import java.util.Locale;  import java.util.jar.JarEntry;  import java.util.jar.JarFile;  import java.util.jar.JarOutputStream;  import java.util.logging.Level;  import java.util.logging.Logger;  import obfuscator.bcel.transform.ClassRenamerTransform;  import obfuscator.bcel.transform.FieldRenamerTransform;  import obfuscator.bcel.transform.MethodRenamerTransform;  import obfuscator.bcel.transform.ObTransform;  import obfuscator.bcel.transform.UnconditionalBranchTransform;  import obfuscator.bcel.transform.StringEncryptorTransform;  import org.apache.bcel.Repository;  import org.apache.bcel.classfile.ClassParser;  import org.apache.bcel.classfile.Code;  import org.apache.bcel.classfile.ConstantMethodref;  import org.apache.bcel.classfile.ConstantNameAndType;  import org.apache.bcel.classfile.ConstantUtf8;  import org.apache.bcel.classfile.JavaClass;  import org.apache.bcel.classfile.Method;  import org.apache.bcel.generic.ClassGen;  public class UseBCEL {  public static Map<String, ClassGen> classes;  public static List<JarEntry> entries;  private static final Class<?>[] TRANSFORMS = new Class<?>[]{UnconditionalBranchTransform.class, FieldRenamerTransform.class, MethodRenamerTransform.class, ClassRenamerTransform.class, StringEncryptorTransform.class};  private static boolean currentlyJar;  private static PrintWriter logOutput;  public static boolean isCurrentlyJar() {  return currentlyJar;  }  public static void println(final Object o) {  System.out.println(o);  }  public static void bcel(String[] args) {  classes = new HashMap<String, ClassGen>();  if (args.length < 1) {  System.out.println("Usage: java com.speed.ob.Obfuscate [files...]");  } else {  /\*try {  logOutput = new PrintWriter(new File("logs", SimpleDateFormat  .getDateTimeInstance(SimpleDateFormat.SHORT, SimpleDateFormat.LONG, Locale.UK)  .format(new Date()).replaceAll("[\\/: ]", " ")  + ".log"));  } catch (FileNotFoundException e1) {  e1.printStackTrace();  }\*/  UseBCEL.println("Obfuscation is starting!");  for (int i = 0; i < args.length; i++) {  String fileName = args[i];  if (fileName.endsWith(".class") && !fileName.endsWith("\_bak.class")) {  File file = new File(fileName);  if (!file.exists()) {  UseBCEL.println(fileName + " doesn't exist!");  } else {  try {  ClassParser cp = new ClassParser(fileName);  JavaClass jc = cp.parse();  Repository.addClass(jc);  ClassGen cg = new ClassGen(jc);  classes.put(fileName, cg);  } catch (IOException e) {  e.printStackTrace();  }  }  }  }    if (classes.size() > 0) {  UseBCEL.println("\n\nBefore changing!");  printCode();  }    for (int i = 0; i < args.length; i++) {  String fileName = args[i];  File file = new File(fileName);  if (!file.exists()) {  UseBCEL.println(fileName + " doesn't exist!");  } else {  UseBCEL.println("\r\nProcessing: " + fileName);  if (fileName.endsWith(".class") && !fileName.endsWith("\_bak.class")) {  currentlyJar = false;  ClassGen cg = classes.get(fileName);  transformClass(cg);  } else if (fileName.endsWith(".jar")) {  try {  currentlyJar = true;  entries = new ArrayList<JarEntry>();  transformJar(new JarFile(file));  } catch (IOException e) {  e.printStackTrace();  }  } else {  UseBCEL.println(fileName + " not supported.");  }  }  }    // Write modified classes  if (currentlyJar == false) {  for (Map.Entry<String, ClassGen> entry: UseBCEL.classes.entrySet()) {  String fileName = entry.getKey();  ClassGen clazz = entry.getValue();  new File(fileName).renameTo(new File(fileName.replace(".class", "\_bak.class")));  try {  Path fileToDeletePath = Paths.get(fileName);  Files.deleteIfExists(fileToDeletePath);    String className = clazz.getClassName();  StringBuilder buf = new StringBuilder(className);  buf.replace(0, className.lastIndexOf(".") + 1, "");  className = buf.toString();    buf = new StringBuilder(fileName);  buf.replace(fileName.lastIndexOf(File.separator) + 1, fileName.length(), className + ".class");  fileName = buf.toString();  clazz.getJavaClass().dump(fileName);  }catch (IOException e) {  e.printStackTrace();  }  }  }    if (classes.size() > 0) {  UseBCEL.println("\n\nAfter changing!");  printCode();  }  //logOutput.close();  }  }  private static void transformJar(JarFile jarFile) {  Enumeration<JarEntry> en = jarFile.entries();  JarOutputStream out = null;  try {  out = new JarOutputStream(new FileOutputStream(jarFile.getName().replace(".jar", "-ob.jar")));  } catch (IOException e) {  e.printStackTrace();  }  while (en.hasMoreElements()) {  JarEntry entry = en.nextElement();  try {  JarEntry newEntry = new JarEntry(entry.getName());  if (entry.getName().endsWith(".class")) {  ClassParser cp = new ClassParser(jarFile.getInputStream(entry), entry.getName());  JavaClass jc = cp.parse();  Repository.addClass(jc);  ClassGen cg = new ClassGen(jc);  entries.add(newEntry);  classes.put(entry.getName(), cg);  } else {  out.putNextEntry(newEntry);  byte[] buffer = new byte[1024];  InputStream in = jarFile.getInputStream(entry);  int read;  while ((read = in.read(buffer)) != -1) {  out.write(buffer, 0, read);  }  out.flush();  out.closeEntry();  }  } catch (IOException e) {  e.printStackTrace();  }  }  for (Class<?> cl : TRANSFORMS) {  for (Map.Entry<String, ClassGen> entry: UseBCEL.classes.entrySet()) {  ClassGen cg = entry.getValue();  try {  ObTransform transform = (ObTransform) cl.getConstructor(ClassGen.class).newInstance(cg);  transform.execute();  } catch (Exception e) {  e.printStackTrace();  UseBCEL.println(cl.getCanonicalName() + " failed to transform class: " + cg.getClassName());  }  }  }  for (Map.Entry<String, ClassGen> entry: UseBCEL.classes.entrySet()) {  ClassGen cg = entry.getValue();  // JarEntry newEntry = entries.get(i);  String name = cg.getClassName().replace('.', '/').concat(".class");  JarEntry entry2 = new JarEntry(name);  try {  out.putNextEntry(entry2);  out.write(cg.getJavaClass().getBytes());  out.flush();  out.closeEntry();  } catch (IOException e) {  e.printStackTrace();  }  }  try {  out.close();  } catch (IOException e) {  e.printStackTrace();  }  }  private static void printCode() {  for (Map.Entry<String, ClassGen> entry: UseBCEL.classes.entrySet()) {  ClassGen cg = entry.getValue();  UseBCEL.println("Class " + cg.getClassName());  for (Method m : cg.getMethods()) {  Code code = m.getCode();  UseBCEL.println(m.getName() + ": " + code);  }  }  }    private static void transformClass(ClassGen cg) {  try {  for (Class<?> clazz : TRANSFORMS) {  ObTransform transform = (ObTransform) clazz.getConstructor(ClassGen.class).newInstance(cg);  transform.execute();  }  } catch (Exception e) {  e.printStackTrace();  }  }  } |

Листинг 8: Файл ObTransform.java

|  |
| --- |
| package obfuscator.bcel.transform;  import org.apache.bcel.classfile.Method;  import org.apache.bcel.classfile.Code;  import org.apache.bcel.generic.ClassGen;  public abstract class ObTransform {  protected final ClassGen cg;  public ObTransform(ClassGen cg) {  this.cg = cg;  }  public abstract void execute();  } |

Листинг 9: Файл GenericSignatureParser.java

|  |
| --- |
| /\*  \* Bytecode Analysis Framework  \* Copyright (C) 2004, University of Maryland  \*  \* This library is free software; you can redistribute it and/or  \* modify it under the terms of the GNU Lesser General Public  \* License as published by the Free Software Foundation; either  \* version 2.1 of the License, or (at your option) any later version.  \*  \* This library is distributed in the hope that it will be useful,  \* but WITHOUT ANY WARRANTY; without even the implied warranty of  \* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU  \* Lesser General Public License for more details.  \*  \* You should have received a copy of the GNU Lesser General Public  \* License along with this library; if not, write to the Free Software  \* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA  \*/  package obfuscator.bcel.resources;  import java.util.Iterator;  import java.util.NoSuchElementException;  import org.apache.bcel.classfile.Attribute;  import org.apache.bcel.classfile.Method;  import org.apache.bcel.classfile.Signature;  import org.apache.bcel.generic.ConstantPoolGen;  import org.apache.bcel.generic.InvokeInstruction;  /\*\*  \* A simple class to parse method signatures that include generic information.  \* <p>  \*  \* Modified from edu.umd.cs.findbugs.ba.SignatureParser  \*  \* @author Nat Ayewah  \*/  public class GenericSignatureParser {  private class ParameterSignatureIterator implements Iterator<String> {  private int index = 1;  public boolean hasNext() {  return index < signature.length() && signature.charAt(index) != ')' && signature.charAt(index) != '^';  }  public String next() {  if (!hasNext())  throw new NoSuchElementException();  StringBuilder result = new StringBuilder();  boolean done;  do {  done = true;  char ch = signature.charAt(index);  switch (ch) {  case 'B':  case 'C':  case 'D':  case 'F':  case 'I':  case 'J':  case 'S':  case 'Z':  case '\*': // wildcard  result.append(signature.charAt(index));  ++index;  break;  case 'L':  case 'T':  String tmp = "";  int startsemi = index;  int leftCount = 0;  int i = startsemi + 1;  loop: while (true) {  char c = signature.charAt(i);  switch (c) {  case ';':  if (leftCount == 0)  break loop;  break;  case '<':  leftCount++;  break;  case '>':  leftCount--;  break;  }  i++;  }  String foo = signature.substring(startsemi, i + 1);  result.append(foo);  index = i + 1;  break;  case '[':  case '+':  case '-':  result.append(signature.charAt(index));  ++index;  done = false;  break;  case ')':  case '^':  throw new NoSuchElementException("Should have already thrown NoSuchElementException");  case 'V':  default:  throw new IllegalStateException("Invalid method signature: '" + signature + "' : "  + signature.substring(index) + " " + result);  }  } while (!done);  return result.toString();  }  public void remove() {  throw new UnsupportedOperationException();  }  }  private final String signature;  /\*\*  \* Parses a generic method signature of the form:  \* <code>(argument\_signature)return\_type\_signature</code>  \*  \* @param signature  \* the method signature to be parsed  \*/  public GenericSignatureParser(String signature) {  // XXX not currently handling Type parameters for class, interface or  // method definitions  int s = signature.indexOf('(');  String sig = signature;  if (s > 0)  sig = sig.substring(s);  else if (s < 0 || sig.indexOf(':') >= 0 || sig.startsWith("(V)"))  throw new IllegalArgumentException("Bad method signature: " + signature);  this.signature = sig;  }  /\*\*  \* Get an Iterator over signatures of the method parameters.  \*  \* @return Iterator which returns the parameter type signatures in order  \*/  public Iterator<String> parameterSignatureIterator() {  return new ParameterSignatureIterator();  }  /\*\*  \* Get the method return type signature.  \*  \* @return the method return type signature  \*/  public String getReturnTypeSignature() {  int endOfParams = signature.lastIndexOf(')');  if (endOfParams < 0)  throw new IllegalArgumentException("Bad method signature: " + signature);  return signature.substring(endOfParams + 1);  }  /\*\*  \* Get the number of parameters in the signature.  \*  \* @return the number of parameters  \*/  public int getNumParameters() {  int count = 0;  for (Iterator<String> i = parameterSignatureIterator(); i.hasNext();) {  i.next();  ++count;  }  return count;  }  /\*\*  \* Get the number of parameters passed to method invocation.  \*  \* @param inv  \* @param cpg  \* @return int number of parameters  \*/  public static int getNumParametersForInvocation(InvokeInstruction inv, ConstantPoolGen cpg) {  GenericSignatureParser sigParser = new GenericSignatureParser(inv.getSignature(cpg));  return sigParser.getNumParameters();  }  /\*\*  \* @param target  \* the method whose signature is to be parsed  \* @return an iterator over the parameters of the generic signature of  \* method. Returns null if the generic signature cannot be parsed  \*/  public static Iterator<String> getGenericSignatureIterator(Method target) {  try {  GenericSignatureParser parser = null;  String genericSignature = null;  for (Attribute a : target.getAttributes()) {  if (a instanceof Signature) {  Signature sig = (Signature) a;  if (genericSignature != null) {  if (!genericSignature.equals(sig.getSignature())) {  if (false) {  System.out.println("Inconsistent signatures: ");  System.out.println(genericSignature);  System.out.println(sig.getSignature());  }  return null; // we've seen two inconsistent  // signatures  }  continue;  }  genericSignature = sig.getSignature();  if (compareSignatures(target.getSignature(), genericSignature))  parser = new GenericSignatureParser(genericSignature);  }  }  Iterator<String> iter = parser == null ? null : parser.parameterSignatureIterator();  return iter;  } catch (RuntimeException e) {  } // degrade gracefully  return null;  }  /\*\*  \* Compare a plain method signature to the a generic method Signature and  \* return true if they match  \*/  public static boolean compareSignatures(String plainSignature, String genericSignature) {  GenericSignatureParser plainParser = new GenericSignatureParser(plainSignature);  GenericSignatureParser genericParser = new GenericSignatureParser(genericSignature);  if (plainParser.getNumParameters() != genericParser.getNumParameters())  return false;  return true;  }  } |

Листинг 10: Файл ClassRenamerTransform.java

|  |
| --- |
| package obfuscator.bcel.transform;  import obfuscator.bcel.UseBCEL;  import java.util.Map;  import obfuscator.bcel.NameGenerator;  import org.apache.bcel.classfile.Constant;  import org.apache.bcel.classfile.ConstantClass;  import org.apache.bcel.classfile.ConstantUtf8;  import org.apache.bcel.generic.ArrayType;  import org.apache.bcel.generic.ClassGen;  import org.apache.bcel.generic.ConstantPoolGen;  import org.apache.bcel.generic.Type;  /\*\*  \* Renames classes.  \*/  public class ClassRenamerTransform extends ObTransform {  // this is static so each instance doesn't just name it's class A  private static NameGenerator nameGen = new NameGenerator();  public ClassRenamerTransform(ClassGen cg) {  super(cg);  UseBCEL.println("Starting class renamer on class " + cg.getClassName());  }  public void execute() {  if (cg.containsMethod("main", Type.getMethodSignature(Type.VOID, new Type[] { new ArrayType(Type.STRING, 1) })) != null) {  // we usually want to prevent classes with main methods from being  // renamed, breaks too many things  UseBCEL.println(cg.getClassName() + " not renamed as contains as main method");  return;  }  // create the new name for the class, leaves the packages intact.  String className = cg.getClassName();  int ind = className.lastIndexOf('.');  String newName;  if (ind > -1) {  newName = className.substring(0, className.lastIndexOf('.')) + '.' + nameGen.next();  } else {  newName = nameGen.next();  }  cg.setClassName(newName);  // sets the class name in the actual class file  String fileName = cg.getFileName();  UseBCEL.println("fileName: " + fileName);  int ut = cg.getConstantPool().lookupUtf8(fileName);  if (ut > -1) {  // changes the source file attribute  ConstantUtf8 c = (ConstantUtf8) cg.getConstantPool().getConstant(ut);  UseBCEL.println("\trenamed source file: " + fileName + " to " + nameGen.current() + ".java");  cg.getConstantPool().setConstant(ut, new ConstantUtf8(nameGen.current() + ".java"));  }  // fix references to the class  //printConstantPool(cg, className, newName);  fixConstantPool(cg, className, newName);  //printConstantPool(cg, className, newName);  UseBCEL.println("\t" + className + " renamed to " + newName);  if (UseBCEL.classes.size() > 0) {  for (Map.Entry<String, ClassGen> entry: UseBCEL.classes.entrySet()) {  ClassGen c = entry.getValue();  ConstantPoolGen cpg = c.getConstantPool();  fixConstantPool(c, className, newName);  int index = cpg.lookupClass(className);  if (index > -1) {  ConstantClass con = (ConstantClass) cpg.getConstant(index);  int utf = con.getNameIndex();  ConstantUtf8 utf8 = (ConstantUtf8) cpg.getConstant(utf);  UseBCEL.println("class:" + cpg.getConstant(utf).getClass().getCanonicalName() + ", index: " + index + ", utf: " + utf + ", newName: " + newName.replace(".", "/"));  cg.getConstantPool().setConstant(utf, new ConstantUtf8(newName.replace(".", "/")));  //cg.getConstantPool().setConstant(utf, new ConstantClass(utf));  UseBCEL.println("\t" + className + " renamed to " + newName + " in class " + c.getClassName());  }  }  }  }  private void fixConstantPool(ClassGen cg, String className, String newName) {  ConstantPoolGen cpg = cg.getConstantPool();  newName = newName.replace('.', '/');  int counter = 0;  for (Constant c : cpg.getConstantPool().getConstantPool()) {  if (c instanceof ConstantUtf8) {  ConstantUtf8 con = (ConstantUtf8) c;  String className1 = className.replace('.', '/');  if (con.getBytes().contains("L" + className1 + ";")) {  String bytes = con.getBytes().replace('L' + className1 + ';', 'L' + newName + ';');  UseBCEL.println("\treplacing " + con.getBytes() + " to " + bytes);  cpg.setConstant(counter, new ConstantUtf8(bytes));  }  else if (con.getBytes().equals(className1)) {  UseBCEL.println("\treplacing " + con.getBytes() + " to " + newName);  cpg.setConstant(counter, new ConstantUtf8(newName));  }  }  counter++;  }  }    private void printConstantPool(ClassGen cg, String className, String newName) {  ConstantPoolGen cpg = cg.getConstantPool();  newName = newName.replace('.', '/');  for (Constant c : cpg.getConstantPool().getConstantPool()) {  if (c instanceof ConstantUtf8) {  ConstantUtf8 con = (ConstantUtf8) c;  String className1 = className.replace('.', '/');  if (con.getBytes().contains("L" + className1 + ";")) {  UseBCEL.println("\tConstant in pool: " + con.getBytes());  }  }  }  }  } |

Листинг 11: Файл FieldRenamerTransform.java

|  |
| --- |
| package obfuscator.bcel.transform;  import obfuscator.bcel.UseBCEL;  import java.util.Map;  import obfuscator.bcel.NameGenerator;  import org.apache.bcel.classfile.ConstantFieldref;  import org.apache.bcel.classfile.ConstantNameAndType;  import org.apache.bcel.classfile.ConstantUtf8;  import org.apache.bcel.classfile.Field;  import org.apache.bcel.generic.ClassGen;  import org.apache.bcel.generic.ConstantPoolGen;  /\*\*  \* Renames fields in a class, and any classes in the same JAR that reference it.  \*/  public class FieldRenamerTransform extends ObTransform {  public FieldRenamerTransform(ClassGen cg) {  super(cg);  UseBCEL.println("Starting field renamer on class "  + cg.getClassName());  }  public void execute() {  NameGenerator nameGen = new NameGenerator();  for (Field field : cg.getFields()) {  ConstantPoolGen cpg = cg.getConstantPool();  String originalName = field.getName();  int index = field.getNameIndex();  String newName = nameGen.next();  int newIndex = cpg.addUtf8(newName);  cpg.setConstant(index, cpg.getConstant(newIndex));  UseBCEL.println("\tRenaming field " + cg.getClassName() + "."  + originalName + " to " + newName);  if (UseBCEL.isCurrentlyJar()) {  int count = 0;  for (Map.Entry<String, ClassGen> entry: UseBCEL.classes.entrySet()) {  ClassGen clazz = entry.getValue();  int fieldRef = clazz.getConstantPool().lookupFieldref(  cg.getClassName(), originalName,  field.getSignature());  if (fieldRef > -1) {  ConstantFieldref ref = (ConstantFieldref) clazz  .getConstantPool().getConstant(fieldRef);  ConstantNameAndType type = (ConstantNameAndType) clazz  .getConstantPool().getConstant(  ref.getNameAndTypeIndex());  int nameIndex = type.getNameIndex();  if (nameIndex > -1) {  ConstantUtf8 utf8 = (ConstantUtf8) cpg  .getConstant(nameIndex);  if (utf8 != null) {  count++;  cpg.setConstant(nameIndex, new ConstantUtf8(newName));  }  }  }  }  UseBCEL.println("\t" + count  + " constant pool references to " + originalName  + " changed to " + newName);  }  }  }  } |

Листинг 12: Файл MethodRenamerTransform.java

|  |
| --- |
| package obfuscator.bcel.transform;  import obfuscator.bcel.UseBCEL;  import org.apache.bcel.Repository;  import org.apache.bcel.classfile.ConstantMethodref;  import org.apache.bcel.classfile.ConstantNameAndType;  import org.apache.bcel.classfile.ConstantUtf8;  import org.apache.bcel.classfile.JavaClass;  import org.apache.bcel.classfile.Method;  import org.apache.bcel.classfile.LocalVariable;  import org.apache.bcel.generic.ClassGen;  import obfuscator.bcel.resources.GenericSignatureParser;  import java.util.Map;  import java.lang.NullPointerException;  import obfuscator.bcel.NameGenerator;  /\*\*  \* Renames methods in a class and local variables in methods  \*/  public class MethodRenamerTransform extends ObTransform {  private NameGenerator names;  public MethodRenamerTransform(ClassGen cg) {  super(cg);  }  public void execute() {  names = new NameGenerator();  UseBCEL.println("Starting method renamer on " + cg.getClassName());  // lets rename abstract class methods and interface methods first  for (Method m : cg.getMethods()) {  try {  UseBCEL.println(m.getName());  for (LocalVariable l: m.getLocalVariableTable().getLocalVariableTable()) {  if (!(l.getName().equals("args") && m.getName().equals("main") || l.getName().equals("this"))) {  String newName = names.next();  UseBCEL.println("\t\tLocal variable " + l.getName() + " was renamed to " + newName);  int index = l.getNameIndex();  cg.getConstantPool().setConstant(index, new ConstantUtf8(newName));  }  }  }  catch (NullPointerException ie) {}    if (m.isNative() || m.getName().equals("<clinit>") || m.getName().equals("<init>")  || m.getName().equals("main")) {  // we dont want to mess with native methods, constructors or  // static initialisers  continue;  }  // look through hierarchy and do not rename any overriden methods  if (lookThroughHierarchy(cg, m, "")) {  UseBCEL.println("\tNot renaming: " + m.getName());  continue;  }  // generate a new name  String oldName = m.getName();  String newName = names.next();  UseBCEL.println("\tRenaming " + oldName + " to " + newName);  // fix the references in this class first  int utf8 = m.getNameIndex();  if (utf8 > -1) {  ConstantUtf8 utf = (ConstantUtf8) cg.getConstantPool().getConstant(utf8);    // set the new name  if (utf.getBytes().equals(oldName))  cg.getConstantPool().setConstant(utf8, new ConstantUtf8(newName));  }  fixClass(cg, cg.getClassName(), oldName, newName, m);  // fix references to this method in the other classes  for (Map.Entry<String, ClassGen> entry: UseBCEL.classes.entrySet()) {  ClassGen clazz = entry.getValue();  fixClass(clazz, cg.getClassName(), oldName, newName, m);  }  }  }  private void fixClass(final ClassGen cg, String clazzName, final String oldName, final String newName,  final Method m) {  int index = cg.getConstantPool().lookupMethodref(clazzName, oldName, m.getSignature());  // look for the method reference  if (index > -1) {  ConstantMethodref ref = (ConstantMethodref) cg.getConstantPool().getConstant(index);  // find the name reference  int typeInd = ref.getNameAndTypeIndex();  if (typeInd > -1) {  UseBCEL.println("\tUpdating method reference in " + cg.getClassName());  ConstantNameAndType nameType = (ConstantNameAndType) cg.getConstantPool().getConstant(typeInd);  int utf8 = nameType.getNameIndex();  // find the utf8  if (utf8 > -1) {  ConstantUtf8 utf = (ConstantUtf8) cg.getConstantPool().getConstant(utf8);  // set the new name  if (utf.getBytes().equals(oldName)) {  cg.getConstantPool().setConstant(utf8, new ConstantUtf8(newName));  cg.getConstantPool().setConstant(index, ref.copy());  cg.getConstantPool().addMethodref(clazzName, newName, m.getSignature());  }  UseBCEL.println("Old name: " + utf.getBytes());  index = cg.getConstantPool().lookupMethodref(clazzName, newName, m.getSignature());  if (index > -1) {  ref = (ConstantMethodref) cg.getConstantPool().getConstant(index);  typeInd = ref.getNameAndTypeIndex();  if (typeInd > -1) {  utf8 = nameType.getNameIndex();  // find the utf8  if (utf8 > -1) {  utf8 = nameType.getNameIndex();  utf = (ConstantUtf8) cg.getConstantPool().getConstant(utf8);  UseBCEL.println("New name: " + utf.getBytes());  }  }  }  }  }  }  }  private boolean lookThroughHierarchy(ClassGen cg, Method m, String previous) {  if (cg.isAnnotation())  return true;  String superclass = cg.getSuperclassName();  ClassGen sup = null;  try {  JavaClass clazz = Repository.lookupClass(superclass);  sup = new ClassGen(clazz);  } catch (ClassNotFoundException e) {  sup = null;  }  if (sup == null) {  return false;  }  // now look through interfaces  for (String s : cg.getInterfaceNames()) {  try {  JavaClass jc = Repository.lookupClass(s);  for (Method meth : jc.getMethods()) {  boolean b = false;  if (meth.getGenericSignature() != null) {  String s1 = meth.getGenericSignature();  b = GenericSignatureParser.compareSignatures(m.getSignature(), s1);  }  if (meth.getName().equals(m.getName()) && (meth.getSignature().equals(m.getSignature()) || b)) {  return true;  }  }  } catch (ClassNotFoundException e) {  UseBCEL.println("\tCannot find interface: " + s);  }  }  for (Method meth : sup.getMethods()) {  // method is overriden, keep the name  if (meth.getName().equals(m.getName()) && meth.getSignature().equals(m.getSignature())) {  return true;  }  }  if (sup.getSuperclassName() == null || previous.equals(sup.getSuperclassName()))  return false;  return lookThroughHierarchy(sup, m, superclass);  }  } |

Листинг 13: Файл StringEncryptorTransform.java

|  |
| --- |
| package obfuscator.bcel.transform;  import java.util.ArrayList;  import java.util.List;  import obfuscator.bcel.UseBCEL;  import org.apache.bcel.Constants;  import org.apache.bcel.classfile.ConstantString;  import org.apache.bcel.classfile.ConstantUtf8;  import org.apache.bcel.classfile.Method;  import org.apache.bcel.generic.ALOAD;  import org.apache.bcel.generic.ASTORE;  import org.apache.bcel.generic.ClassGen;  import org.apache.bcel.generic.ConstantPoolGen;  import org.apache.bcel.generic.DUP;  import org.apache.bcel.generic.GOTO;  import org.apache.bcel.generic.I2B;  import org.apache.bcel.generic.I2C;  import org.apache.bcel.generic.IAND;  import org.apache.bcel.generic.ICONST;  import org.apache.bcel.generic.IINC;  import org.apache.bcel.generic.ILOAD;  import org.apache.bcel.generic.ISTORE;  import org.apache.bcel.generic.IXOR;  import org.apache.bcel.generic.InstructionFactory;  import org.apache.bcel.generic.InstructionHandle;  import org.apache.bcel.generic.InstructionList;  import org.apache.bcel.generic.InvokeInstruction;  import org.apache.bcel.generic.LDC;  import org.apache.bcel.generic.MethodGen;  import org.apache.bcel.generic.ObjectType;  import org.apache.bcel.generic.POP;  import org.apache.bcel.generic.SIPUSH;  import org.apache.bcel.generic.Type;  /\*\*  \* Encrypts strings.  \*/  public class StringEncryptorTransform extends ObTransform {  private String methodName;  private int callsChanged;  private List<String> encryptedStrings;  public StringEncryptorTransform(final ClassGen cg) {  super(cg);  methodName = "aA";  while (cg.containsMethod(methodName, Type.getMethodSignature(  Type.STRING, new Type[] { Type.STRING })) != null) {  methodName = methodName + "A";  }  encryptedStrings = new ArrayList<String>();  System.out.printf(  "Loading encryptor for %s, decrypt method name: %s\n",  cg.getClassName(), methodName);  }  public void execute() {  UseBCEL.println("Starting encryption on class " + cg.getClassName());  changeCalls();  UseBCEL.println("\tChanged " + callsChanged + " LDCs");  if (callsChanged > 0) {  insertMethod();  UseBCEL.println("\tInserted method " + methodName);  }  }  private void insertMethod() {  // this method generates the decryption method, this way I could  // technically encrypt strings with information from the classes  // themselves, making it harder to decrypt  InstructionList list = new InstructionList();  MethodGen newMethod = new MethodGen(  (Constants.ACC\_PRIVATE | Constants.ACC\_STATIC), Type.STRING,  new Type[] { Type.STRING }, new String[] { "encrypted" },  methodName, cg.getClassName(), list, cg.getConstantPool());  ConstantPoolGen cpg = cg.getConstantPool();  InstructionFactory factory = new InstructionFactory(cpg);  list.append(factory.createNew((ObjectType) Type  .getType(StringBuilder.class)));  list.append(new DUP());  list.append(factory.createInvoke("java.lang.StringBuilder", "<init>",  Type.VOID, new Type[] {}, Constants.INVOKESPECIAL));  list.append(new ASTORE(1));  list.append(new ALOAD(0));  list.append(factory.createInvoke("java.lang.String", "length",  Type.INT, new Type[] {}, Constants.INVOKEVIRTUAL));  list.append(new SIPUSH((short) 255));  list.append(new IAND());  list.append(new I2B());  list.append(new ISTORE(2));  list.append(new ICONST(0));  InstructionHandle go1Prev = list.append(new ISTORE(3));  InstructionHandle go1After = list.append(new ALOAD(1));  list.append(new ALOAD(0));  list.append(new ILOAD(3));  list.append(factory.createInvoke("java.lang.String", "charAt",  Type.CHAR, new Type[] { Type.INT }, Constants.INVOKEVIRTUAL));  list.append(new ILOAD(2));  list.append(new IXOR());  list.append(new I2C());  list.append(factory.createInvoke("java.lang.StringBuilder", "append",  Type.getType("Ljava/lang/StringBuilder;"),  new Type[] { Type.CHAR }, Constants.INVOKEVIRTUAL));  list.append(new POP());  list.append(new ILOAD(2));  list.append(new ILOAD(3));  list.append(new IXOR());  list.append(new I2B());  list.append(new ISTORE(2));  list.append(new IINC(3, 1));  list.append(go1Prev, new GOTO(list.append(new ILOAD(3))));  list.append(new ALOAD(0));  list.append(factory.createInvoke("java.lang.String", "length",  Type.INT, new Type[] {}, Constants.INVOKEVIRTUAL));  list.append(InstructionFactory.createBranchInstruction(  Constants.IF\_ICMPLT, go1After));  list.append(new ALOAD(1));  list.append(factory.createInvoke("java.lang.StringBuilder", "toString",  Type.STRING, new Type[] {}, Constants.INVOKEVIRTUAL));  list.append(factory.createInvoke("java.lang.String", "intern",  Type.STRING, new Type[] {}, Constants.INVOKEVIRTUAL));  list.append(InstructionFactory.createReturn(Type.STRING));  newMethod.setMaxLocals();  newMethod.setMaxStack();  cg.addMethod(newMethod.getMethod());  }  private void changeCalls() {  InvokeInstruction invoke = new InstructionFactory(cg).createInvoke(  cg.getClassName(), methodName, Type.STRING,  new Type[] { Type.STRING }, Constants.INVOKESTATIC);  boolean isChanged;  for (Method m : cg.getMethods()) {  if (m.isAbstract() || m.isNative()) {  continue;  }  ConstantPoolGen cpg = cg.getConstantPool();  MethodGen mg = new MethodGen(m, cg.getClassName(), cpg);  InstructionList il = mg.getInstructionList();  isChanged = false;  for (InstructionHandle handle : il.getInstructionHandles()) {  if (handle.getInstruction() instanceof LDC) {  LDC ldc = (LDC) handle.getInstruction();  if (ldc.getType(cpg).equals(Type.STRING)) {  int cpIndex = ldc.getIndex();  String original = ldc.getValue(cpg).toString();  if (!encryptedStrings.contains(original)) {  isChanged = true;  String encrypted = A.decrypt(original);  ConstantString string = (ConstantString) cpg  .getConstant(cpIndex);  int utf8 = string.getStringIndex();  ConstantUtf8 con = (ConstantUtf8) cpg  .getConstant(utf8);  cpg.setConstant(utf8, new ConstantUtf8(encrypted));  encryptedStrings.add(encrypted);  UseBCEL.println("The original \"" + original + "\" was encrypted to \"" + encrypted + "\"");  }  il.insert(handle.getNext(), invoke);  callsChanged++;  }  }  }  if (isChanged == true) {  il.update();  mg.setInstructionList(il);  mg.setMaxLocals();  mg.setMaxStack();  mg.removeLocalVariables();  mg.update();  cg.replaceMethod(m, mg.getMethod());  }  }  }  }  final class A {  public final static String decrypt(String encrypted) {  StringBuilder builder = new StringBuilder();  byte xor = (byte) (encrypted.length() & 0xFF);  for (int pos = 0; pos < encrypted.length(); pos++) {  builder.append((char) (encrypted.charAt(pos) ^ xor));  xor ^= pos;  }  return builder.toString().intern();  }  } |

Листинг 14: Файл UnconditionalBranchTransform.java

|  |
| --- |
| package obfuscator.bcel.transform;  import org.apache.bcel.generic.\*;  import java.util.Random;  import java.security.SecureRandom;  import org.apache.bcel.classfile.Method;  import org.apache.bcel.classfile.Attribute;  import org.apache.bcel.classfile.StackMap;  import org.apache.bcel.classfile.ConstantUtf8;  import org.apache.bcel.Constants;  /\*\*  \* Finds unconditional branches and adds a condition to them. This results in  \* loop obfuscation as many loops will have unconditional branches. Obstructs  \* the control flow.  \*  \* @author Shivam Mistry  \*/  public class UnconditionalBranchTransform extends ObTransform {  private String fieldName;  public UnconditionalBranchTransform(ClassGen cg) {  super(cg);  fieldName = "controlField";  }  public void execute() {  if (cg.isInterface()) {  return;  }  insertControlField();  findUnconditionalBranches();  }  private void insertControlField() {  while (cg.containsField(fieldName) != null) {  // create a control field with a name, this should be name  // obfuscated anyway  fieldName += "A";  }  FieldGen fg = new FieldGen(  Constants.ACC\_PRIVATE | Constants.ACC\_STATIC, Type.INT,  fieldName, cg.getConstantPool());  // add the field to the class  cg.addField(fg.getField());  }  private InstructionHandle insertInvalidCode(InstructionList list, InstructionFactory factory, MethodGen mg) {  InstructionHandle end = list.getEnd();  //TODO: add some sort of variation  InstructionList dead = new InstructionList();  Random random = new SecureRandom();  dead.append(new ICONST(random.nextInt(6)));  dead.append(new BIPUSH((byte) random.nextInt(128)));  dead.append(new DUP\_X1());  dead.append(new SWAP());  dead.append(new POP());  dead.append(new POP());  dead.append(new POP());  if(mg.getType() == Type.INT || mg.getType() == Type.BOOLEAN || mg.getType() == Type.CHAR || mg.getType() == Type.SHORT  || mg.getType() == Type.LONG || mg.getType() == Type.BYTE) {  dead.append(new ICONST(0));  dead.append(new POP());  } else if(!mg.getReturnType().equals(Type.VOID)){  dead.append(new ACONST\_NULL());  dead.append(new POP());  }  dead.append(new GOTO(dead.getStart()));  //dead.append(new RETURN());  dead.update();  return list.append(end, dead);  }    private void findUnconditionalBranches() {  // Random random = new Random();  for (Method m : cg.getMethods()) {  if (m.isAbstract() || m.isNative() || m.getName().equals("<init>") || m.getName().equals("<clinit>"))  continue;  MethodGen mg = new MethodGen(m, cg.getClassName(),  cg.getConstantPool());  InstructionHandle deadCodeStart = null;  for (InstructionHandle ih : mg.getInstructionList()  .getInstructionHandles()) {  // find unconditional branches, add conditions to them  if (ih.getInstruction() instanceof GOTO) {  InstructionList list = mg.getInstructionList();  InstructionFactory factory = new InstructionFactory(cg);  if (deadCodeStart == null) {  deadCodeStart = insertInvalidCode(list, factory, mg);  }  // push zero on to the stack  InstructionHandle zero = list.append(ih.getPrev(),  new ICONST(0));  // get the value of the 'control' field  list.append(zero, factory.createFieldAccess(  cg.getClassName(), fieldName, Type.INT,  Constants.GETSTATIC));  // compare integers, check if control is zero, complete the  // jump if it is  InstructionHandle target = ((GOTO) ih.getInstruction())  .getTarget();  // random between not equal to and equal to, doesn't matter  // as the goto will jump to the target anyway  ih.setInstruction(InstructionFactory  .createBranchInstruction(Constants.IF\_ICMPEQ,  target));  // go to the invalid code added at the end of the method  list.append(ih, new GOTO(deadCodeStart));  list.update();  mg.setInstructionList(list);  }  }  mg.setMaxLocals();  mg.setMaxStack();  mg.removeLocalVariables();  mg.update();  cg.replaceMethod(m, mg.getMethod());  }  }  } |