**BC/NW 2018 № 2 (33):11.1**

**РАЗРАБОТКА ПРОГРАММЫ, ОБЕСПЕЧИВАЮЩЕЙ ЗАЩИТУ JAVA ПРОГРАММ С ПОМОЩЬЮ АППАРАТНОГО КЛЮЧА SENTINEL HASP**

Темирсултанов А.Д

Одним из наиболее важных условий функционирования вычислительных систем является защита от несанкционированного использования программного обеспечения. Не существует программных продуктов, которые не поддавались бы взлому. И совсем не важно на каком языке программирования написана программа. Затраты на взлом могут быть различными и поэтому все основные действия по защите от нелегального использования должны быть направлены на затруднение возможных действий, связанных с несанкционированным копированием, отладкой, декомпиляцией и модификацией программ.

Современные средства, такие как .NET и Java[1] платформы, подвержены декомпиляции, особенно это касается Java-программ, которые распространяются свободно.

В данной работе разработан алгоритм надежной защиты программ, написанных на языке Java и поставляемых в виде байт-кода, с помощью аппаратного ключа Sentinel HASP [2].

**Анализ задания и постановка задачи**

В нынешнее время, век информационных технологий, доходы компаний-разработчиков программного обеспечения огромны. Не только операционные системы, но и профессиональные приложения (например, графическое программное обеспечение) могут быть очень дорогими. Как следствие, появилось незаконное использование программного обеспечения. С помощью всего нескольких щелчков мыши люди могут загружать программное обеспечение, применять к нему загруженный патч и использовать его без оплаты. Поставщики поняли, что защита программного обеспечения от вредоносных пользователей является серьезной проблемой. Пользователь контролирует свою машину: у него есть физический доступ к аппаратным средствам, он контролирует сетевое подключение и т. д. Тем не менее, владельцы программного обеспечения также могут противостоять этим угрозам.

Существует два основных способа защиты программного обеспечения: защита на программной основе, а также защита на основе аппаратных решений.

В программных решениях, после установки ПО на компьютере конечного пользователя, защита и лицензирование этого ПО будет связано с конкретным компьютером. ПО будет работать только после ввода пользователем ключа продукта.

В аппаратных решения вместе с программным обеспечением поставляется внешнее аппаратное устройство. Правильное функционирование программного обеспечения будет зависеть от устройства, подключенного к компьютеру конечного пользователя. Во время выполнения, ПО взаимодействует с аппаратным устройством, и функционирует правильно только в том случае, когда получает от устройства требуемый ответ.

Чем больше мер повышения безопасности применяется и чем выше уровень их сложности, тем меньше вероятность вмешательства в приложение или получения доступа к секретной информации.

Основными преимуществами программных средств защиты являются низкая стоимости и гибкость, например, его совместимость с существующими системами. Главным недостатком является их недостаточное сопротивление различным видам атак. Из-за меняющихся методов, которые обходят защиту приложений, необходимо придумать более сложные методы.

Локальное программное обеспечение и конфиденциальные данные могут быть защищены путем шифрования и аутентификации. Всякий раз, когда программное обеспечение выполняется, оно будет дешифровать свои внутренности «на лету». К сожалению, этот метод гарантирует полную безопасность, если шифрование и дешифрование выполняется в надежном оборудовании, например, на криптографическом сопроцессоре. Но когда этот компонент недоступен, как и в большинстве существующих систем, проблема становится сложнее решать. По сути, такой сопроцессор можно считать системой черного ящика, где злоумышленник может контролировать поведение ввода-вывода. Если шифрование и дешифрование не выполняются доверенным оборудованием, программа может быть перехвачена при ее отправке в память или процессор черного ящика. Кроме того, технология “*trusted computing*”[3] предлагает способ аутентификации программного обеспечения и платформы. Основываясь на небольшом защищенном аппаратном компоненте (например, *“trusted platform module*”[4]), этот метод направлен на проверку соответствия системы пользователя, то есть принуждение как аппаратного, так и программного обеспечения вести себя ожидаемым образом.

Второй аппаратный метод защиты программного обеспечения - это защита от несанкционированного доступа (“*tamper resistance*”[5]). В этом случае программное обеспечение и данные физически защищены от атак. Средствами обеспечения данного метода защиты являются криптографические процессоры и смарт-карты. Тем не менее, даже в этом случае некоторые атаки может извлекать некоторую информацию из этих систем, применяя “атаку по сторонним каналам”[6] или “атаку по энергопетреблению”[7].

И третий метод - аппаратные ключи. Аппаратные ключи - это небольшие аппаратные токены (например, USB-накопители), которые необходимо подключить к компьютеру при работе защищенного программного обеспечения. Без ключа ПО не запускается. Одним из недостатков такого метода является то, что ключи теряются. В этом случае изготовитель должен создать или выпустить новый ключ, или должна быть установлена другая копия программного обеспечения с другим ключом.

Два основных недостатка аппаратной технологии обеспечения безопасности - высокая стоимость и возможная несовместимость с существующими открытыми компьютерными платформами. Под «стоимостью» понимаются расходы на покупку и установку, с одной стороны, и стоимость модернизации и обслуживания, с другой стороны. Следовательно, аппаратные решения обеспечивают меньшую гибкость, чем программные решения. Если механизм был поврежден, необходимо обновить аппаратное обеспечение. Эти обновления являются дорогостоящими и очень медленными для развертывания. Поэтому создаваемые компанией методы обеспечения безопасности программного обеспечения должны быть как можно более независимыми от аппаратных средств и платформ, чтобы обеспечить максимальную гибкость.

Итак, мы имеем программу, написанную на Java, аппаратный ключ Sentinel HASP и API к этому ключу. Задача состоит в том, разработать алгоритм защиты Java-программы с использования имеющихся средств. Необходимо применить алгоритм к исходной программе и проверить работоспособность полученной защищенной программы.

**Основные сведения**

***Защита SENTINEL HASP***

Sentinel HASP - это передовое решение для защиты программного обеспечения от незаконного или несанкционированного использования. Решение устраняет незаконный доступ и выполнение защищенных приложений.

Для развертывания Sentinel HASP-защищенной программы требуется доступ к определенному ключу защиты Sentinel HASP для запуска. Защищенная программа запрашивает ключ защиты Sentinel HASP для предопределенной информации. Если защитный ключ Sentinel HASP отсутствует или полученная информация неверна, программа не запускается или перестает работать.

После того, как вы выбрали метод защиты Sentinel HASP, реализация проста. Независимо от выбранной стратегии защиты, защищенные программы работают корректно, только если они могут получить доступ к информации, хранящейся в определенном ключе Sentinel HASP.

Система защиты Sentinel HASP основана на следующем:

1. Защита программ и файлов данных - Sentinel HASP обеспечивает два основных метода защиты:

* ****Sentinel HASP Envelope – это приложение, которое выполняет обертку вашего приложения и таким образом образует защитный щит. Это приложение предлагает расширенные функции защиты для повышения общего уровня безопасности вашего программного обеспечения;
* Sentinel HASP Run‐time API – это программный интерфейс, который предоставляет функции для работы с ключом защиты Sentinel HASP.

1. Идентификация ключа безопасности Sentinel HASP - основным фактором, определяющим защиту HASP, является то, может ли развернутая программа идентифицировать и получать доступ к интеллекту, содержащемуся в определенном ключе HASP, во время выполнения;
2. Шифрование AES - механизм шифрования в защитном ключе HASP основан на алгоритме AES. Шифрование HASP использует набор 128-битных ключей шифрования, которые остаются в ключе HASP;
3. Параметры конфиденциальной защиты - каждому ключу HASP присваивается уникальный код поставщика, так называемый “Vendor Code”, который должен быть конфиденциальным;
4. Использование памяти HASP - память HASP на ключах HASP можно использовать (читать и писать) в качестве компонента схемы защиты программного обеспечения.
5. Меры против отладки и обратного проектирования - HASP предоставляет функциональность для борьбы с отладчиками и обратным проектированием.

Далее будет проведен обзор функционала HASP API, так как с использованием этого API и самого ключа планируется провести исследование:

* Как программно с помощью этого API добиться максимально надежной защиты для исполнимых файлов(exe и dll в Windows);
* Как надежно защитить с использованием этих ключей приложения, поставляемые в виде исходного текста или байт-кода(Java и т.п.);
* Возможность достижения надежной защиты для регулярных файлов с помощью этого API.

***SENTINEL HASP API***

Пропусти предварительные требования необходимые для начала работы с ключом и сразу приступим к рассмотрения функционала, который предоставляет API.

*Функция Login*

Для успешного поиска и связи с ключом HASP, необходимо создать успешный сеанс входа в систему посредством функции login. Для входа в систему необходимо предоставить идентификатор *Feature ID* и код поставщика *Vendor Code*. На рис.1. показана работа функции login.

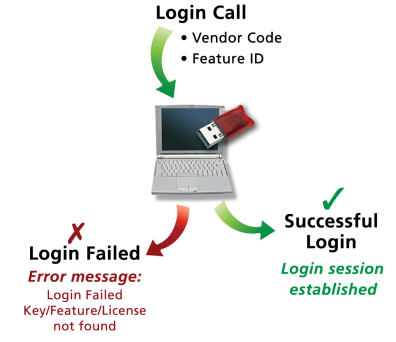


Рис.1. Функция login

*Функциональность HASP API*

Степень защиты, предоставляемая API, зависит от способа ее реализации. Обращения к ключу HASP, которые вставляются в исходный код, управляют доступом к приложению во время выполнения.

Функции Sentinel HASP Run-time API подразделяются на пять групп, основанных на общих функциях и связях:

* Session функции – сессия открывается путем успешного входа в систему путем успешного обращения к лицензии, находящейся в определенном ключе HASP. Закрытие сессии выполняется вызовом функции logout;
* Encryption функции – функции для шифрования или дешифрования буферов данных, используя механизм шифрования на основе AES в защитном ключе HASP. Механизм шифрования использует симметричное шифрование. Это означает, что один и тот же ключ шифрования используется позже для дешифрования буфера данных;
* Memory функции – выполняют доступ к памяти HASP, в которой можно хранить данные, которые будут использоваться приложением во время выполнения, а также информацию, которая может использоваться позже для проверки и идентификации конечного пользователя;
* Time функции – при использовании HASP HL Time key или HASP HL NetTime key, предоставляется возможность доступа к часам реального времени на ключе;
* Management функции - API включает функции, которые позволяют вам получать информацию о системных компонентах, текущем сеансе входа в систему, состоянии развернутого ключа защиты Sentinel HASP и обновлениях лицензий.

Весь перечень функций представлен в таблице 1.

Таблица 1

Функции HASP API

|  |  |
| --- | --- |
| Функция | Описание |
| hasp\_datetime\_to\_hasptime() | Преобразование значения времени и даты в hasptime |
| hasp\_decrypt() | Дешифрование буфера с использованием алгоритма AES |
| hasp\_detach() | Отделяет продукт и его лицензию от HASP SL ключа, в соответствии с настраиваемыми параметрами |
| hasp\_encrypt() | Шифрование буфера с использованием алгоритма AES |
| hasp\_free() | Освобождает выделенные ресурсы памяти |
| hasp\_get\_rtc() | Считывает текущее время с HASP HL ​​Time или с HASP HL ​​NetTime |
| hasp\_get\_sessioninfo() | Извлекает информацию о контексте сеанса |
| hasp\_get\_info() | Извлекает информацию в соответствии с настраиваемыми параметрами поиска и представляет их в соответствии с  настраиваемым форматом |
| hasp\_get\_size() | Возвращает размер памяти ключа в байтах |
| hasp\_hasptime\_to\_datetime() | Преобразование hasptime в значения времени и даты |
| hasp\_login() | Создание нового сеанса работы |
| hasp\_login\_scope() | Получает регистрационную информацию в соответствии с настраиваемыми параметрами поиска |
| hasp\_logout() | Выход из контекста или сеанса |
| hasp\_read() | Считывание памяти ключа |
| hasp\_update() | Записывает обновление лицензии HASP |
| hasp\_write() | Запись в память ключа |

**Алгоритм решения задачи**

Для несанкционированного использования злоумышленник, как правило, должен выполнить набор манипуляций по отладке или декомпиляции Java кода, а так же его модификации. Защита от отладки и декомпиляции является моментом, на котором можно максимально затруднить задачу взломщика.

В языке программирования Java, компилятор не создает конечный машинный код, а всего лишь его платформенно независимое представление. Полученный байт-код содержит очень много осмысленной информации, которая может помочь разобраться взломщику программы в принципе её работы. Наиболее популярным средством декомпиляции является программа «Java Decompiler» [8].

В связи с данной проблемой было решено разработать методику защиты Java-приложений несанкционированного использования. В ее основе лежат принцип шифрования «на лету» и динамическая загрузка классов. Для защиты байт-кода используется его шифрование. Аппаратный ключ HASP, включающий в себя криптопроцессор, реализующий алгоритм шифрования AES со 128-битным ключом, отлично подойдет для этого.

         Однако одно только шифрование не достаточно для построения стабильной защиты Java-приложении. Одной из основных особенностей платформы Java является модель динамической загрузки классов, которая позволяет загружать исполняемый код в JRE (Java Runtime Environment [1]) не перезагружая основное приложение. Во время работы приложения, не только система, но и пользователь (расширяя функциональность класса java.lang.ClassLoader [1]) имеет возможность создавать загрузчики классов. Связь между различными загрузчиками регламентируется моделью делегирования загрузки (Рис.2).

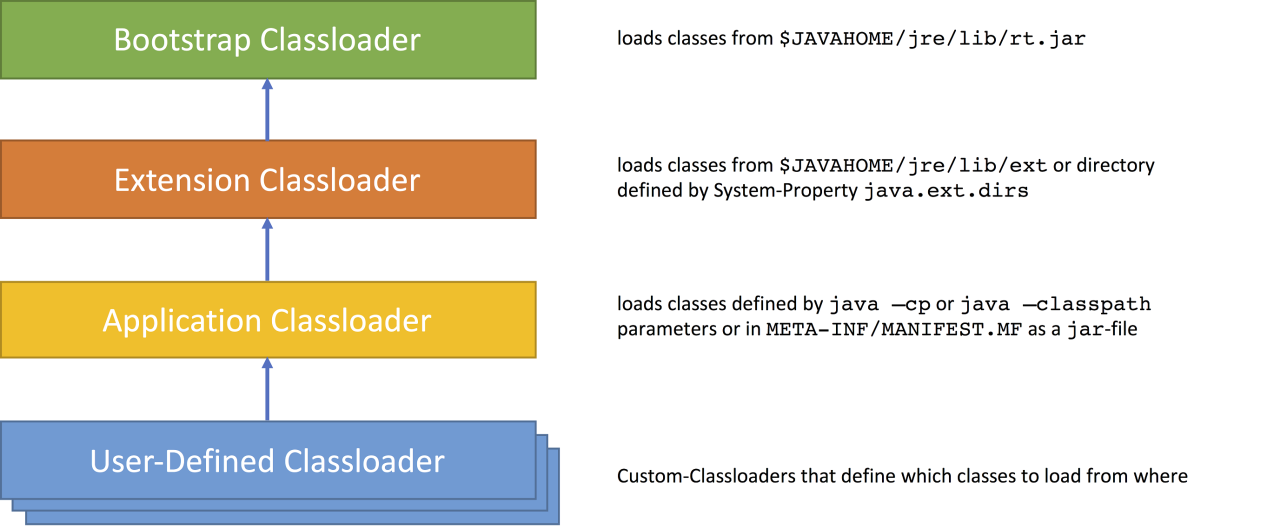


Рис.2. Модель делегирования загрузки

Для динамической загрузки Java-классов было принято решение разработать пользовательский загрузчик. Выбранные программистом, необходимые для корректной работы программы классы, шифруются ключом HASP и хранятся в зашифрованном виде. При выполнении программы, необходимые классы расшифровываются на лету нашим ключом и передаются на выполнение в JRE. Без ключа выполнение программы невозможно.

Для динамической загрузки классов в Java используется рефлексия (Java Reflection API [1]). Этот механизм позволяет исследовать данные о программе во время ее выполнения. Для нас важно, что этот механизм позволяет нам, после расшифрования и создания класса загрузчиком, создавать объекты этого класса. Проблема заключается в том, что для полноценного использования нашего нового объекта, необходимо привести его тип к типу нашего класса. Эта проблема решается созданием абстрактного класса [1]. Для каждого шифруемого класса создаем соответствующий абстрактный класс, где прописываем все поля и сигнатуры методов. А шифруемые классы наследуем от соответствующих абстрактных классов. Таким образом, приведя тип объекта к типу абстрактного класса, мы получаем полноценный доступ к нашему объекту. Также использование абстрактных классов позволяет уменьшит размеры шифруемых классов, путем переноса не особо важных на усмотрение программиста методов из шифруемого класса в его абстрактный родительский класс.

И, наконец, добавим в программу периодическую проверку наличия ключа, для этого напишем поток, который будет работать в фоновом режиме и осуществлять проверку.

Также напишем программу для шифрования наших классов. Программная реализация представлена в следующем разделе.

Итак, распишем алгоритм по пунктам:

1. Определиться с тем, какие классы программы необходимо зашифровать;
2. Пишем потоковый класс, который будет в фоновом режиме проверять наличие ключа и оповещать остальные классы о его отсутствии путем изменения статической переменной;
3. Для каждого шифруемого класса пишем соответствующий абстрактный класс и наследуем класс от него, также в каждом шифруемом классе пишем проверку переменной, оповещающей об отсутствии ключа ;
4. Шифруем классы программой шифрования, либо утилитой шифрования *Sentinel ToolBox*, полученные файлы сохраняем с теми же названиями, расширения меняем с *.class* на *.eclass*;
5. Пишем пользовательский загрузчик классов, который загружаем поток байтов на входе, расшифровывает их с помощью HASP, и создает класс;
6. Модифицируем остальные классы, в которых использовались зашифрованные классы. Вначале проверяем наличие ключа. В главном классе запускаем поток проверки. Далее создаем объекта нашего пользовательского загрузчика. Загружаем с помощью загрузчика необходимые раннее зашифрованные классы, создаем объекты этих классов, приводя их типы к соответствующим абстрактным классам.

**Программная реализация алгоритма**

Для реализации нашего алгоритма выберем исходную программу, состоящую из трех классов. Программа вычисляет погрешность при выполнении прямого и обратного дискретного преобразования Фурье. Все три класса находятся в одном пакете. Исходный код незащищенной программы представлен в *Приложении Б.*

Первый класс Function1содержит функцию, выполняющую прямое дискретное преобразование Фурье:

**package** hasp;**public class** Function1 {  
 *//прямое дискретное преобразование Фурье* **public void** dft(**float**[] x, **float**[] yReal, **float**[] yImag) {  
 **int** n = x.**length**;  
 **for** (**int** k = 0; k < n; k++) {   
 **float** sumreal = 0;  
 **float** sumimag = 0;  
 **for** (**int** j = 0; j < n; j++) {   
 **float** angle = (**float**) (2 \* Math.***PI*** \* j \* k / n);  
 sumreal += x[j] \* Math.*cos*(angle);  
 sumimag += -x[j] \* Math.*sin*(angle);  
 }  
 yReal[k] = (**float**) (1.0 / n \* sumreal);  
 yImag[k] = (**float**) (1.0 / n \* sumimag);  
 }  
 }  
}

Второй класс Function2содержит функцию, выполняющую обратное дискретное преобразование Фурье:

**package** hasp;**public class** Function2 {  
 *//обратное дискретное преобразование Фурье* **public void** dftInverse(**float**[] yReal, **float**[] yImag, **float**[] x) {  
 **int** n = x.**length**;  
 **for** (**int** k = 0; k < n; k++) { *// For each output element* **float** sumreal = 0;  
 **float** sumimag = 0;  
 **for** (**int** j = 0; j < n; j++) { *// For each input element* **float** angle = (**float**) (2 \* Math.***PI*** \* j \* k / n);  
 sumreal += yReal[j] \* Math.*cos*(angle) - yImag[j] \* Math.*sin*(angle);  
 sumimag += yReal[j] \* Math.*sin*(angle) + yImag[j] \* Math.*cos*(angle);  
 }  
 x[k] = (**float**) Math.*sqrt*(sumreal \* sumreal + sumimag \* sumimag);  
 }  
 }  
}

Третья функция Main высчитывает погрешность массива чисел после прямого и обратного преобразования Фурье. Операция выполняется с различным набором чисел в массиве.

**package** hasp;**public class** Main {  
 *//длина массива* **private static final int *N*** = 10;  
 **public static void** main(String[] args) {  
 **float**[] x = **new float**[***N***];  
 **float**[] yReal = **new float**[***N***];  
 **float**[] yImag = **new float**[***N***];  
 **float**[] xChange = **new float**[***N***];  
 **float** delta = (**float**) 0.0;  
 Function1 f1 = **new** Function1();  
 Function2 f2 = **new** Function2();  
 **for** (**float** alpha = (**float**)0.0000000001;alpha<= 1000000.0; alpha \*= 10.0) {  
 *//задаем элементы массива x* **for** (**int** i = 1; i <= ***N***; i++) {  
 **if**(i % 2 == 0) x[i-1] = (**float**) (i / 10.0 + alpha);  
 **else** x[i-1] = (**float**) (i / 10.0);  
 }  
 *//прмямое преобразование Фурье над массивом X* f1.dft(x, yReal, yImag);  
 *//обратное преобразование Фурье  
 //получаем массив xChange* f2.dftInverse(yReal, yImag, xChange);  
 *//вычисляем максимальную относительную погрешность преобразований* **for** (**int** i = 0; i < ***N***; i++) {  
 **float** tmp = Math.*abs*(x[i] - xChange[i])/x[i] \* 100;  
 **if**(delta < tmp) delta = tmp;  
 }  
 System.***out***.printf(**"%30.15f --- %30.20f\n"**, alpha, delta);  
 }  
 }  
}

Выполним по пунктам алгоритм защиты над данной исходной программой:

1. Для шифрования выберем два класса Function1 и Function2, так как они выполняют основные вычисления в программе и без их загрузки программа бесполезна.
2. Пишем потоковый класс KeyBackgroundCheck, который в фоновом режиме проверяет наличие ключа и изменять переменную haspPresentKey при его отсутствии, заканчивает выполнение при прерывании. Добавим переменную haspPresentKey в класс Main:

**public class** Main {  
  
 **private static final int *N*** = 10;  
 **public static boolean** *haspKeyPresent* = **false**;**public static void** main(String[] args) { ... }

Так как проверка наличия ключа требует подключения к нему напишем класс HaspCodes и добавляем featureId для этого класса:

**package** hasp;**import** Aladdin.Hasp;  
**import** java.util.HashMap;  
**import** java.util.Map;  
**public class** HaspCodes {  
 *//vendorCode ключа* **public static final** String ***vendorCode*** = **"AzIceaq.../ttys7VKf4J6+b/a/Y"**;  
 *//featureId используемые в программе* **public static** Map<String, Long> *faetureId* = **new** HashMap<String, Long>();  
 *//добавление featureId* **static** {  
 *faetureId*.put(**"Main"**, Hasp.***HASP\_DEFAULT\_FID***);  
  
 }  
}

Теперь можем написать класс проверки:

**package** hasp;**import** Aladdin.Hasp;  
**import** Aladdin.HaspStatus;  
**public class** KeyBackgroundCheck **extends** Thread {  
 **private final** String **scope** =  
 **"<?xml version=\"1.0\" encoding=\"UTF-8\" ?>"** +  
 **"<haspscope/>"**;  
 **private final** String **format** =  
 **"<?xml version=\"1.0\" encoding=\"UTF-8\" ?>"** +  
 **"<haspformat root=\"hasp\_info\">"** +  
 **" <feature>"** +  
 **" <attribute name=\"id\" />"** +  
 **" <attribute name=\"locked\" />"** +  
 **" <attribute name=\"expired\" />"** +  
 **" <attribute name=\"disabled\" />"** +  
 **" <attribute name=\"usable\" />"** +  
 **" </feature>"** +  
 **"</haspformat>"**;  
 **private** String **info**;  
 @Override  
 **public void** run() {  
 *//Объект hasp* Hasp hasp = **new** Hasp(HaspCodes.*faetureId*.get(**"Main"**));  
 *//Подключение к ключу* hasp.login(HaspCodes.***vendorCode***);  
 **try** {  
 **while** (**true**) {  
 *//Если поток не прерван* **if** (!Thread.*interrupted*()) {  
 System.***out***.println(Thread.*currentThread*().getName());  
 *//запрашиваем статус* **info** = hasp.getInfo(**scope**,**format**,HaspCodes.***vendorCode***);  
 **int** status = hasp.getLastError();  
 *//проверка статуса* **if** (HaspStatus.***HASP\_STATUS\_OK*** != status) {  
 *//если ключ не найден, меняем переменную* Main.*haspKeyPresent* = **false**;  
 System.***out***.printf(**"Error #%d"**, status);  
 } **else** {  
 *//если все хорошо, ждем пол секунды* Main.*haspKeyPresent* = **true**;  
 Thread.*sleep*(500);  
 }  
 } **else** {  
 **throw new** InterruptedException();  
 }  
 }  
 } **catch** (InterruptedException ex) {  
 hasp.logout();  
 **return**;  
 }  
  
 }  
}

1. Для каждого шифруемого класса пишем соответствующий абстрактный класс. Для класса Function1 - AbstractFunction1:

**package** hasp;**public abstract class** AbstractFunction1 {  
 *//абстрактный метод прямого преобразования* **public abstract void** dft(**float**[] x, **float**[] yReal, **float**[] yImag);  
}

Для класса Function2 – AbstractFunction2:

**package** hasp;**public abstract class** AbstractFunction2 {  
 *//абстрактный метод обратного преобразования* **public abstract void** dftInverse(**float**[] yReal, **float**[] yImag, **float**[] x);  
  
}

Наследуем каждый класс от соответствующего абстрактного класса и добавляем проверку переменной наличия ключа. Класс Function1:

**public class** Function1 **extends** AbstractFunction1{  
 **public void** dft(**float**[] x, **float**[] yReal, **float**[] yImag) {  
 **if**(Main.*haspKeyPresent*) { ... }

}

}

Класс Function2:

**public class** Function2 **extends** AbstractFunction2{  
 **public void** dftInverse(**float**[] yReal, **float**[] yImag, **float**[] x) {  
 **if**(Main.*haspKeyPresent*) { ... }

}

}

1. Шифруем классы Function1 и Function2 программой шифрования EncryptFiles, представленной в *Приложении Г*, Для шифрования каждого класса выберем различные featureId и добавим их в наш набор в HaspCodes:

*//добавление featureId***static** {  
 *faetureId*.put(**"Main"**, Hasp.***HASP\_DEFAULT\_FID***);  
 *faetureId*.put(**"Function1"**, **new** Long(33));  
 *faetureId*.put(**"Function2"**, **new** Long(101));  
}

Либо утилитой шифрования *Sentinel ToolBox*. Полученные файлы сохраняем с теми же названиями, расширения меняем с *.class* на *.eclass*;

1. Пишем пользовательский загрузчик классов EClassLoader, который загружаем поток байтов на входе, расшифровывает их с помощью HASP, и создает класс:

**package** hasp;**import** Aladdin.Hasp;  
**import** Aladdin.HaspStatus;  
**import** java.io.\*;

**public class** EClassLoader **extends** ClassLoader {  
 *//путь к папке с зашифрованными файлами* **private** String **pathToEClass**;  
  
 **public** EClassLoader(String pathToEClass, ClassLoader parent) {  
 **super**(parent);  
 **this**.**pathToEClass** = pathToEClass;  
 }  
 *//поиск классов* @Override  
 **protected** Class<?> findClass(String name) **throws** ClassNotFoundException{  
 **try**{  
 *//имя пакета* String packageName = Main.**class**.getPackage().getName() + **"."**;  
 *//загружаем и расшифровываем необходимый класс*

*//с соответствующий featureId* **byte** b[] = fetchClassFromFS(**pathToEClass** + name + **".eclass"**,

HaspCodes.*faetureId*.get(name));  
 *//из массива байт создаем объекта Class нашего класса* **return** defineClass(packageName + name, b, 0, b.**length**);  
 } **catch** (IOException ex) {  
 *//в случае неудачи вышестоящий загрузчик загружает класс* **return super**.findClass(name);  
 }  
 }  
 *//функция для чтения и расшифрования классов* **private byte**[] fetchClassFromFS(String path,иLong featureId)и**throws** IOException{  
 *//Создание сессии с ключом* Hasp hasp = **new** Hasp(featureId);  
 hasp.login(HaspCodes.***vendorCode***);  
 **int** status = hasp.getLastError();  
 **if** (HaspStatus.***HASP\_STATUS\_OK*** != status) {  
 System.***out***.printf(**"Error #%d"**, status);  
 **throw new** IOException(**"Hasp error"**);  
 }  
 *//открывам поток чтения файла* InputStream is = **new** FileInputStream(**new** File(path));  
 *//проверка длины файла* **long** length = **new** File(path).length();  
 **if**(length > Integer.***MAX\_VALUE***) {  
 **throw new** IOException(**"Too long file"** + path);  
 }  
 **byte**[] bytes = **new byte**[(**int**)length];  
 *//чтение файла* **int** offset = 0;  
 **int** numRead = 0;  
 **while** ((offset < bytes.**length**) && (((numRead = is.read(bytes,

offset, bytes.**length** - offset))) >= 0))   
 offset += numRead;  
 **if**(offset < bytes.**length**) {  
 **throw new** IOException(**"could not completely read file"** + path);  
 }  
 *//расшифровываем массив байт* hasp.decrypt(bytes);  
 hasp.logout();  
 is.close();  
 **return** bytes;  
 }  
}

1. Изменяем главный класс Main. Добавляем в него проверку ключа, запуск потока проверки ключа KeyBackgroundCheck, добавляем загрузчик классов EClassLoader и изменяем процесс создания классов Function1 и Function2. Класс Main после внесения изменений:

**package** hasp;  
**import** Aladdin.Hasp;  
**import** Aladdin.HaspStatus;

**public class** Main {  
 **private static final int *N*** = 10;  
 **public static boolean** *haspKeyPresent* = **false**;**public static void** main(String[] args) {  
 *//создание сессии и проверка статуса* Hasp hasp = **new** Hasp(HaspCodes.*faetureId*.get(**"Main"**));  
 hasp.login(HaspCodes.***vendorCode***);  
 **int** status = hasp.getLastError();  
 **if** (HaspStatus.***HASP\_STATUS\_OK*** != status)  
 System.***out***.printf(**"Error #%d"**, status);  
 *//запускаем поток фоновой проверки ключа* KeyBackgroundCheck keyBackgroundCheck = **new** KeyBackgroundCheck();  
 keyBackgroundCheck.start();

**float**[] x = **new float**[***N***];  
 **float**[] yReal = **new float**[***N***];  
 **float**[] yImag = **new float**[***N***];  
 **float**[] xChange = **new float**[***N***];  
 **float** delta = (**float**) 0.0;

*//создаем загрузчик* EClassLoader eClassLoader = **new**

EClassLoader(**"out/production/HASP\_After/hasp/EClasses/"**,

ClassLoader.*getSystemClassLoader*());  
 **try** {  
 *//Загрузка зашифрованного класса Function1  
 //с помощью нашего пользовательского загрузчика* Class c1 = eClassLoader.loadClass(**"Function1"**);  
 *//Создание объекта f1 класса Function1  
 //Приведение типа объекта к типу абстарктного класса AbstractFunction1* AbstractFunction1 f1 = (AbstractFunction1) c1.newInstance();  
 *//Загрузка класса Function2* Class c2 = eClassLoader.loadClass(**"Function2"**);  
 *//Создание объекта f2 класса Function2  
 //Приведение типа объекта к типу абстарктного класса AbstractFunction2* AbstractFunction2 f2 = (AbstractFunction2) c2.newInstance();  
 **for** (**float** alpha = (**float**)0.00000001;alpha<= 1000000.0; alpha \*= 10.0) {  
 *//задаем элементы массива x* **for** (**int** i = 1; i <= ***N***; i++) {  
 **if**(i % 2 == 0) x[i-1] = (**float**) (i / 10.0 + alpha);  
 **else** x[i - 1] = (**float**) (i / 10.0);  
 }  
 *//обратное преобразование Фурье  
 //получаем массив xChange* f1.dft(x, yReal, yImag);  
 *//обратное преобразование Фурье  
 //получаем массив xChange* f2.dftInverse(yReal, yImag, xChange);  
 *//вычисляем максимальную относительную погрешность преобразований* **for** (**int** i = 0; i < ***N***; i++) {  
 **float** tmp = Math.*abs*(x[i] - xChange[i])/x[i] \* 100;  
 **if**(delta < tmp) delta = tmp;  
 }  
 System.***out***.printf(**"%30.15f --- %30.20f\n"**, alpha, delta);  
 }  
 } **catch** (ClassNotFoundException e) {  
 e.printStackTrace();  
 } **catch** (IllegalAccessException e) {  
 e.printStackTrace();  
 } **catch** (InstantiationException e)   
 e.printStackTrace();  
 keyBackgroundCheck.interrupt();  
 hasp.logout();  
 }  
}

Исходный код, полученной в результате программы, представлен в *Приложении В.* В результате получили, защищенную от несанкционированного доступа программу, для доступа к которой без ключа придется расшифровать алгоритм AES со 128-битным ключом. При наличии ключа узнать исходный код программы не составит никакого труда.

**Результаты тестирования**

Выполним тестирование полученной программы и исследуем зависимость времени выполнения от количества шифруемых классов и их от из размера относительно исходной программы.

Наши классы зашифрованы с помощью алгоритма шифрования AES со 128-битным ключом. Так как этот алгоритм достаточно надежен, то есть с чисто математической точки зрения подобрать одни правильный пароль из всех возможных – трудноосуществимая задача, а каждый класс может быть зашифрован отдельным ключом, то делаем вывод, что доступ к классам без ключа не получить. Так как сам ключ алгоритма хранится в аппаратном ключе, то и с помощью программной подмены ключа не добиться результата.

При наличии ключа, для опытного пользователя Java при затрачивании некоторого времени, не составит труда добраться до исходного кода зашифрованных классов, которые уже загружены и выполняются в JVM. Методы получения информации могут доходить даже до модификации самой JVM.

Доступ к остальным незашифрованным классам программы, можно получить при использовании декомпилятора Java Decompiler. Их можно изменять любым образом, но без наличия кода зашифрованных классов, эти меры бесполезны.

Выполнение программы останавливается после того, как из компьютера вынули ключ, так как в алгоритме предусмотрены и реализованы в программе шаги по проверке наличия ключа.

Исследуем время выполнения исходной и защищенной программы при различных условиях и выясним, как добавление в программу защиты повлияет на время ее выполнения. Результаты проверки защищенной программы представлены в Таблице 2.

Таблица 2

Время выполнения защищенной программы

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество классов | Размер классов, мс | | | |
| 1КБ | 2 КБ | 4 КБ | 8 КБ |
| 2 | ~400 | ~415 | ~405 | ~420 |
| 4 | ~600 | ~610 | ~615 | ~615 |
| 6 | ~820 | ~810 | ~830 | ~815 |
| 8 | ~1020 | ~1030 | ~1020 | ~1035 |
| 10 | ~1240 | ~1220 | ~1230 | ~215 |

Исходная программы все тесты выполнила примерно за 40-60 мс. Из таблицы видно, что количество файлов сильно влияет на время выполнения, в то время как объем классов почти не вносит изменений. Делаем вывод, что лучше писать объемные классы, но в меньшем количестве. В крайнем случае можно часть реализации класса перенести в абстрактный класс.

Заключение

В результате выполнения курсовой работы разработан алгоритм надежной защиты Java-программы с помощью аппаратного ключа Sentinel HASP. Проведена программная реализация алгоритма на тестовой программе. Проведено тестирования защиты и выполнено исследование зависимости времени выполнения программы.

**Литература**

1. Шилдт Г. Java 8. Полное руководство. – М.:Вильямс, 2017. – 1376 с.
2. Software Protection and Licensing Guide[Электронный ресурс] - <http://sentinelldk.safenet-inc.com/LDKdocs/SPNL> .
3. Trusted computing[Электронный ресурс] - <https://en.wikipedia.org/wiki/Trusted_Computing>.
4. Trusted platform module[Электронный ресурс] -https://ru.wikipedia.org/wiki/Trusted\_Platform\_Module .
5. Tamper resistance[Электронный ресурс] -https://en.wikipedia.org/wiki/Tamper\_resistance.
6. Атака по сторонним каналам[Электронный ресурс] -https://en.wikipedia.org/wiki/Side-channel\_attack.
7. Атака по энергопетреблению[Электронный ресурс] -https://en.wikipedia.org/wiki/Power\_analysis.
8. Java Decompiler [Электронный ресурс] - http://jd.benow.ca/