Ильин И.Д., Кульков И.В., Болдырев А.Г. Особенности оценки безопасности реализации алгоритмов сетевой аутентификации методом статического анализа исходного кода. // Проблемы информатики в образовании, управлении, экономике и технике: Сб. статей XVII Междунар. научно-техн. конф. – Пенза: ПДЗ, 2017. – С. 113-121.

УДК 004

ОСОБЕННОСТИ ОЦЕНКИ БЕЗОПАСНОСТИ РЕАЛИЗАЦИИ АЛГОРИТМОВ СЕТЕВОЙ АУТЕНТИФИКАЦИИ МЕТОДОМ СТАТИЧЕСКОГО АНАЛИЗА ИСХОДНОГО КОДА

И.Д. Ильин, И.В. Кульков, А.Г. Болдырев

FEATURES OF SECURITY STATIC ANALYSIS OF NETWORK AUTHENTICATION ALGORITHMS

I.D. Ilin, I.V. Kulkov, A.G. Boldyrev

Аннотация. В настоящей статье рассматриваются особенности применения статических анализаторов для выявления типовых программных ошибок, влияющих на безопасность программных компонентов, реализующих протокол аутентификации.

Ключевые слова: аутентификация, статический анализ, безопасность.

Abstract. This article describes aspects of static analyzers application for detection of typical errors that affect security of the authentication modules.

Keywords: authentication, static analysis, security.

Рассмотрим типовой вариант сетевой инфраструктуры, в которой доступ внешних клиентов к ресурсам, расположенным на файл-сервере защищаемой сети, предоставляется исключительно после успешного прохождения процедуры аутентификации. Упрощенная схема представлена ниже, на рисунке 1.

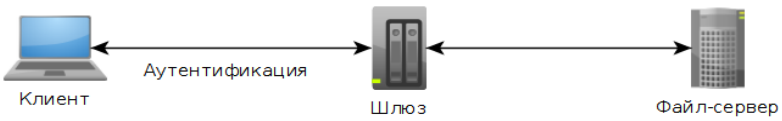

Рис. 1. Схема сети

Наиболее распространенной схемой реализуемой в протоколах аутентификации является Challenge-response authentication [1]. В соответствии с этой схемой взаимодействие Клиента, желающего получить доступ к ресурсам, и Шлюза, ограничивающего доступ к ним, происходит следующим образом:

1. Клиент инициирует запрос на аутентификацию со Шлюзом.
2. Шлюз отправляет в сторону Клиента запрос, содержащий сгенерированную уникальную информацию.
3. Клиент отправляет в сторону Шлюза ответ, содержащий свои идентификационные данные (логин) и некоторый хэш, вычисленный на основе уникальной информации, принятой в запросе от Шлюза и известного пароля Клиента.
4. Шлюз также производит аналогичный расчет хэша на основе тех же данных: эталонный пароль и уникальная информация, переданная Клиентом.
5. Если полученный Шлюзом от Клиента хэш и хэш, рассчитанный Шлюзом самостоятельно, совпадают, то Шлюз делает вывод о том, что Клиент обладает корректными аутентификационными данными (паролем) и предоставляет ему доступ к ресурсам.

На рисунке 2 приведена упрощенная иллюстрация данной схемы:

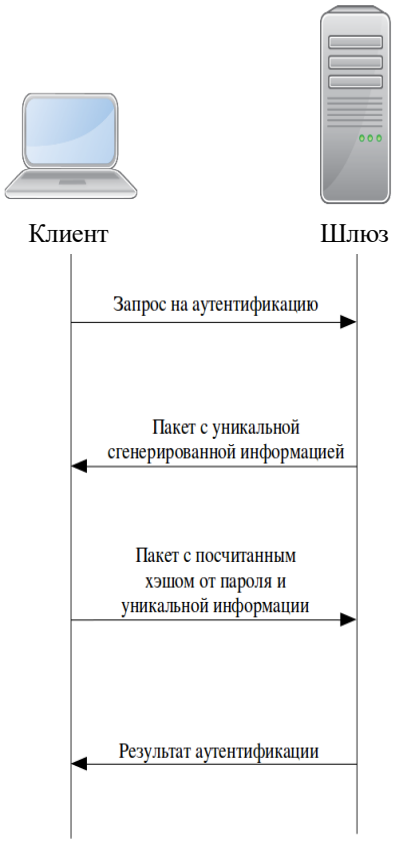

Рис. 2. Схема аутентификации

Очевидным преимуществом данного метода аутентификации является то, что пароль не передается по каналам связи в открытом виде.

Задача проверки безопасности компонентов, реализующих алгоритмы сетевой аутентификации, является одной из ключевых задач при анализе защищенности сетевой инфраструктуры. Актуальность этой проблемы подтверждается статистикой обнаружения уязвимостей в модулях аутентификации широко распространенных сетевых решений. В качестве примера (см. таблицу 1) приведен перечень уязвимостей [2], обнаруженных в сервисе FreeRadius, использующем схему аутентификации challenge-response.

Таблица 1

Уязвимости, обнаруженные в FreeRadius

№	Уязвимость	CWE (Тип)	Описание	Результат
1	CVE-2017-10987	CWE-119	Отсутствует проверка границ буфера	Отказ сервиса
2	CVE-2017-9148	CWE-787	Существует возможность записи данных в произвольную область памяти	Выполнение произвольного кода Отказ сервиса
3	CVE-2017-3697	CWE-399	Ошибка управления ресурсами	Отказ в обслуживании
4	CVE-2009-2505	CWE-287	Ошибки в логике работы	Превышение прав доступа

Анализируя уязвимости в различных сервисах, можно выделить основные типовые программные ошибки, которые допускаются при создании компонентов сетевой аутентификации. Наиболее распространенные из них приведены в таблице 2.

Таблица 2

Основные типы ошибок

Ошибка	Опасное событие
Переполнение буфера	Отказ в доступе для санкционированного пользователя, возможность исполнения вредоносного кода
Неверная обработка исключений	Отказ в доступе для санкционированного пользователя
Висячий указатель	Возможность получения злоумышленником конфиденциальной информации, возможность воздействия на данные злоумышленником (искажение, удаление, подмена)
Обращение по нулевому указателю	Отказ в доступе для санкционированного пользователя, возможность исполнения вредоносного кода

Статический анализ исходного кода программного обеспечения не требует реального выполнения исследуемых программ и позволяет провести первичную оценку безопасности разрабатываемого кода на ранних этапах разработки.

В большинстве случаев статический анализ выполняется путем обзора программного кода (частный случай “code review”) с использованием автоматизированного средства (статический анализатор) и поиска потенциально опасных программных конструкций.

Статические анализаторы осуществляют выявление потенциально опасных программных конструкций на основе правил, которые зачастую можно конфигурировать вручную, применяя их к различным участкам кода.

Существуют различные методы обнаружения программных ошибок при проведении статического анализа. Рассмотрим два наиболее распространенных метода:

1. Анализ с использованием символического выполнения.
2. Анализ потока данных.

В первом случае анализатор строит граф достижимых состояний программы (см. рисунок 3), который учитывает все пути ее выполнения. Основной проблемой является появление слишком большого количества путей для анализа (“комбинаторный взрыв”), что может привести к сбоям в работе статического анализатора.

Во втором случае строится граф потока выполнения (см. рисунок 4), отличающийся тем, что он использует слияние разных ветвей во избежание большого количества вариаций. В зависимости от алгоритма слияния данные статические анализаторы могут пропускать ошибки, допущенные в какой-либо ветке.

Для примера разберем следующий код [3]:

```
1:    #include <iostream>
2:    void test_func(int unknown_value) {
3:          int check_var;
4:          if (unknown_value)
5:                check_var = 5;
6:          std::cout<<"hi"<<std::endl;
7:          if (!unknown_value)
8:                std::cout<<check_var;
9:    }
```

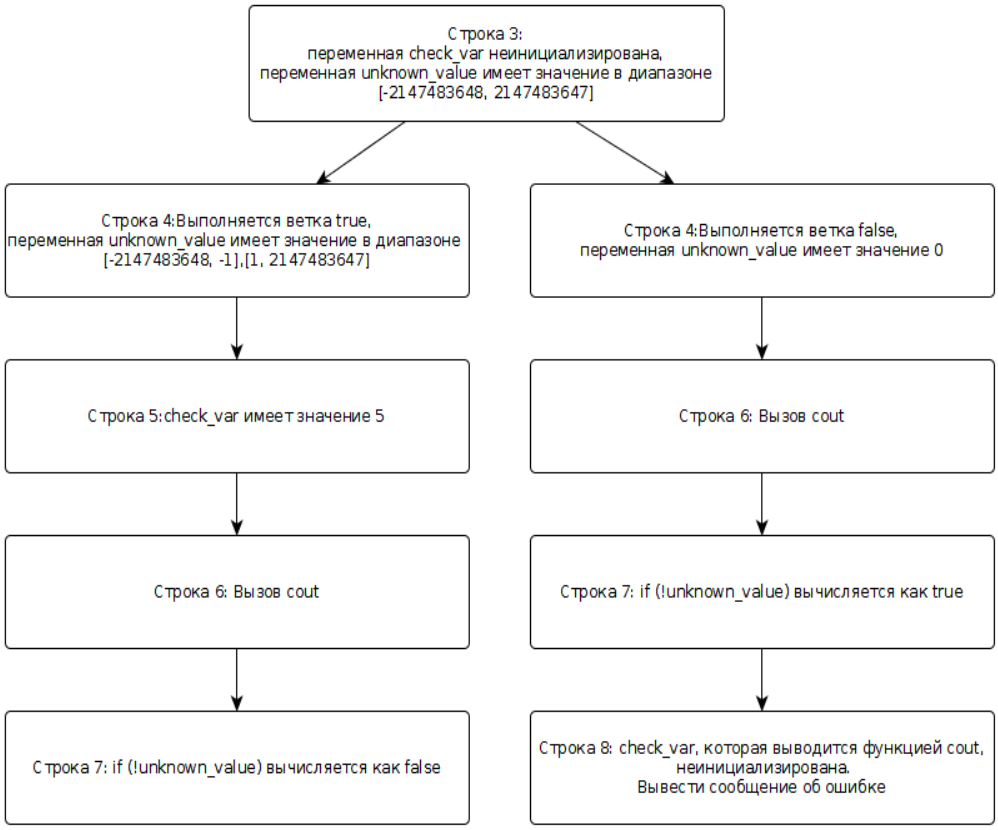

Рис. 3. Граф достижимых состояний

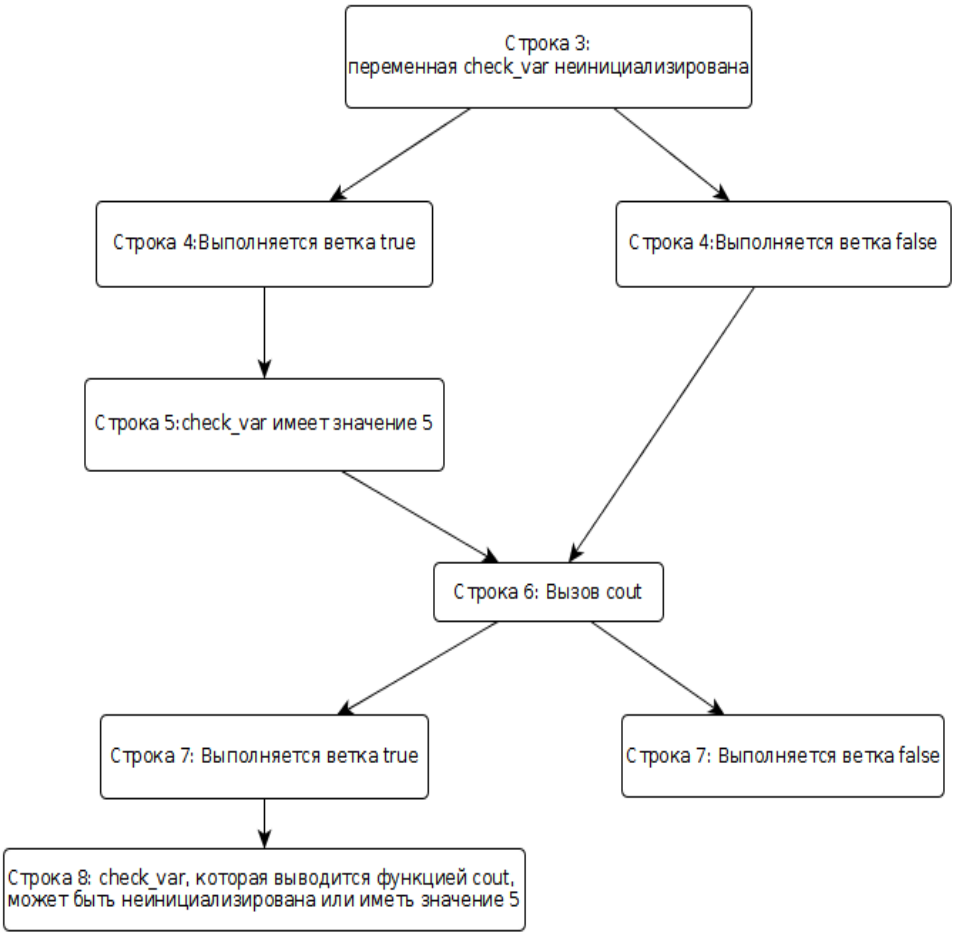

Рис. 4. Граф потока выполнения

Анализируя работу упомянутых методов применительно к коду примера, можно сделать следующие выводы:

- при построении графа достижимых состояний формируются две независимые ветви графа, по каждой из которых производится анализ на использование неинициализированной переменной;

- при построении графа потока выполнения выполняется слияние в одной точке (стр. 6), и дальнейший анализ может не учитывать отсутствие инициализации переменной.

Таким образом, при рассмотрении и выборе статических анализаторов необходимо обращать внимание на метод обнаружения программных ошибок и отдавать предпочтение анализатору, способному использовать сразу оба указанных метода.

Различные статические анализаторы могут обладать функциональными особенностями, такими как:

- возможность и необходимость интеграции со сборочной средой;

- наличие ознакомительного периода для анализаторов, не распространяющихся свободно;

- возможность проведения анализа с использованием символьного выполнения;

- возможность кастомной конфигурации правил анализирования;

- возможность доработки алгоритмов проверок.

Приведем сводную таблица функциональных особенностей для наиболее популярных в настоящий момент статических анализаторов:

Таблица 2

Характеристики статических анализаторов

	Необходимость внедрения в среду сборки	Наличие ознакомительного периода	Возможность доработки алгоритмов проверок	Возможность проведения анализа с использованием символьного выполнения	Возможность кастомной конфигурации правил анализирования	Ограничения на использование
CppCheck	-	-	+	-	+	Нет
Coverity	-	-			+	Отсутствует бесплатная версия
Clang Static Analyzer	+		+	+	+	Нет
Klocwork Static Code Analysis	-	+			+	Ограниченное время бесплатного использования
PVS-Studio	+	+			+	Ограниченная поддержка UNIX-систем
Флаги gcc	+	-	-	-	-	Нет

Если не принимать во внимание прочие характеристики статических анализаторов, то по совокупности выбранных критериев и наличию ограничений на использование оптимальными статическими анализаторами можно считать: CppCheck и Clang Static Analyzer.

Эти статические анализаторы (в соответствии с техническим описанием) могут выявить следующие основные программные ошибки:

1. CppCheck:
a. выход за пределы массива;
b. утечки памяти;
c. разыменовывание NULL-указателей;
d. использование неинициализированной переменной;
e. проверка обработки исключительных ситуаций на безопасность.
2. Clang Static Analyzer:
a. деление на ноль;
b. разыменование NULL-указателей;
c. использование неинициализированной переменной;
d. утечки памяти.

Рассмотрим особенности анализа безопасности выбранными статическими анализаторами (CppCheck и Clang Static Analyzer) следующих тестовых примеров:

Пример 1 - Разыменование нулевого указателя.
core.NullDereference:

```
1.  class C {
2.  public:
3.    int x;
4.  };
5.
6.  void test() {
7.    C *pc = 0;
8.    int k = pc->x; // warn
9.  }
```

Результат: CppCheck и Clang Static Analyzer обнаруживают данную ошибку.

Пример 2 - Деление на ноль.
core.DivideZero:

```
1.  void test(int z) {
2.    if (z == 0)
3.      int x = 1 / z; // warn
4.  }
```

Результат: Clang Static Analyzer обнаруживает, а Cppcheck не обнаруживает данную ошибку.

Пример 3 - Использование неинициализированной индексной переменной.
core.uninitialized.ArraySubscript:

```
1.  void test() {
2.    int i, a[10];
3.    int x = a[i]; // warn: array subscript is undefined
4.  }
```

Результат: CppCheck и Clang Static Analyzer обнаруживают данную ошибку.

Таким образом, можно сделать вывод, что при оценке безопасности программных реализаций алгоритмов аутентификации, с учетом присущих им основных типовых ошибок, наиболее эффективным является совместное применение разнородных статических анализаторов.

Библиографический список

1. H. C. Grossman, Challenge – Response / Clemson, Computer Science 420/620 (cpsc420) [Электронный ресурс]: 2012. URL: http://www.cs.clemson.edu/course/cpsc420/material/Authenication/Challenge-Response.pdf (дата обращения 20.09.2017)

2. Перечень уязвимостей FreeRadius / Национальная база уязвимостей NIST [Электронный ресурс]. URL: https://nvd.nist.gov/vuln/search/results?adv_search=false&form_type=basic&results_type=overview&search_type=all&query=FreeRadius (дата обращения 20.09.2017)

3. Бруно Кардо Лопес, Рафаэль Аулер. LLVM: инфраструктура для разработки компиляторов / пер. с англ. А.Н. Киселев. М.: ДМК Пресс, 2015.

Ильин Игорь Данияpович
ООО “Открытые решения»,
г. Пенза, Россия

Ilin I.D.
LLC Otkrytye resheniya,
Penza, Russia

Кульков Иван Васильевич
ООО “Открытые решения»,
г. Пенза, Россия

Kulkov I.V.
LLC Otkrytye resheniya,
Penza, Russia

Болдырев Александр Геннадьевич
ООО “Открытые решения»,
г. Пенза, Россия

Boldyrev A.G.
LLC Otkrytye resheniya,
Penza, Russia