**Министерство образования и науки Российской Федерации**

**ФГАОУ ВО «УрФУ имени первого Президента России Б.Н. Ельцина»**

Департамент математики, механики и компьютерных наук ФГАОУ ВО УрФУ ИЕНиМ

Оценка работы\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Руководитель от УрФУ Гибилинда Р.В.

**СИСТЕМА ПОИСКА ЗАГРУЖЕННЫХ WEB-ШЕЛЛОВ   
НА WEB-СЕРВЕРЕ**

ОТЧЕТ

Производственная практика

Производственная практика, преддипломная

Руководитель практики от предприятия Гибилинда Р.В.

Студент Арцыбашев Г.И.

Специальность (направление подготовки) 02.03.01 Математика и компьютерные науки

Группа МЕН-400206

Екатеринбург 2024

**ОГЛАВЛЕНИЕ**

[ВВЕДЕНИЕ 3](#_Toc167882428)

[ПОСТАНОВКА ЗАДАЧИ 4](#_Toc167882429)

[СИСТЕМА ПОИСКА ЗАГРУЖЕННЫХ WEB-ШЕЛЛОВ НА WEB-СЕРВЕРЕ 5](#_Toc167882430)

[1 Характерные особенности web-шеллов 5](#_Toc167882431)

[2 Подозрительные функции и команды 6](#_Toc167882432)

[3 Преобразование над текстом программы 7](#_Toc167882433)

[3.1 Обоснование выбора характеристики 7](#_Toc167882434)

[3.2 Способы выявления преобразования 8](#_Toc167882435)

[3.3 Информационная энтропия 9](#_Toc167882436)

[3.4 Избыточность языка 10](#_Toc167882437)

[3.5 Избыточность преобразованного текста 11](#_Toc167882438)

[4 Обфускация кода программы 12](#_Toc167882439)

[4.1 Определение 12](#_Toc167882440)

[4.2 Способы выявления обфускации 13](#_Toc167882441)

[4.3 Расстояние Левенштейна 14](#_Toc167882442)

[5 Реализация алгоритма обнаружения преобразования 15](#_Toc167882443)

[5.1 Описание алгоритма 15](#_Toc167882444)

[5.2 Поиск и удаление непреобразованной части текста 16](#_Toc167882445)

[5.3 Расчёт энтропии 17](#_Toc167882446)

[5.4 Вынесение вердикта по значению энтропии 18](#_Toc167882447)

[6 Реализация алгоритма обнаружения обфускации 19](#_Toc167882448)

[6.1 Поиск имён 19](#_Toc167882449)

[6.2 Поиск похожего слова в словаре 20](#_Toc167882450)

[6.3 Обнаружение обфускации 22](#_Toc167882451)

[7 Реализация алгоритма поиска подозрительных функций и команд 23](#_Toc167882452)

[7.1 Классификация подозрительных инструкций 23](#_Toc167882453)

[7.2 Поиск подозрительных инструкций 25](#_Toc167882454)

[8 Проверка корректности и эффективности работы алгоритмов 26](#_Toc167882455)

[8.1 Описание выборки для алгоритма обнаружения преобразования 26](#_Toc167882456)

[8.2 Проверка работы алгоритма обнаружения преобразования 27](#_Toc167882457)

[8.3 Описание выборки для алгоритма обнаружения обфускации 28](#_Toc167882458)

[8.4 Проверка работы алгоритма обнаружения преобразования 29](#_Toc167882459)

[ЗАКЛЮЧЕНИЕ 30](#_Toc167882460)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 31](#_Toc167882461)

# ВВЕДЕНИЕ

Целью работы является исследование методов поиска загруженных web-шеллов на web-сервере и их программная реализация на языке Python 3.

Web-шелл (веб-оболочка) – командная оболочка (программа или скрипт для управления устройством с помощью команд) для удаленного управления веб-сервером [1]. Веб-шелл может использоваться для выполнения легитимных задач, однако чаще всего применяется в кибератаках, поскольку позволяет установить обратную связь с атакующим, давая ему полный контроль над сервером. Веб-шелл может быть написан на любом языке программирования, поддерживаемом сервером, например, PHP. Для внедрения веб-шеллов злоумышленники используют различные методы, эксплуатируя уязвимости как самого сервера, так и связанных с ним веб-сайтов или приложений. После установки и запуска веб-шелл функционирует как бэкдор, предоставляя преступнику доступ к файловой системе и базам данных сервера, позволяет прослушивать трафик и удаленно выполнять команды [1].

Все методы поиска, рассматриваемые в работе, основаны на анализе исходного кода web-приложений на различных языках программирования для выявления в нём характерных для веб-шеллов особенностей.

# ПОСТАНОВКА ЗАДАЧИ

Задача представляет собой несколько подзадач:

1. изучить примеры исходных кодов web-шеллов;
2. выявить и описать характерные особенности типичных web-шеллов;
3. описать и реализовать алгоритмы анализа исходного кода web-приложений (на языках программирования PHP, JavaScript, Python, Ruby, C#) для поиска выявленных особенностей;
4. проверить корректность и эффективность работы алгоритмов.

# СИСТЕМА ПОИСКА ЗАГРУЖЕННЫХ WEB-ШЕЛЛОВ НА WEB-СЕРВЕРЕ

## **1 Характерные особенности web-шеллов**

На основании проведённого исследования были определены следующие индикаторы того, что рассматриваемый файл может быть веб-шеллом:

* подозрительные функции и команды;
* наличие преобразования (подстановки, шифрование) над текстом программы;
* обфускация (один из видов преобразования) текста программы.

Обоснование отдельного рассмотрения обфускации будет приведено позже.

Важно отметить, что и легитимные файлы могут иметь указанные особенности. Поэтому однозначно определить, является ли определённый файл веб-шеллом, может быть затруднительно.

### **1.1 Подозрительные функции и команды**

К подозрительным функциям можно отнести функции языков программирования, которые могут быть использованы злоумышленником для:

1. преобразования текста (например, обратное преобразование текста с помощью функции *base64\_decode* в PHP);
2. вычисления и выполнения некоторого выражения во время работы программы (например, передача команды форматирования диска для командной оболочки ОС с помощью функции *system* в PHP);
3. взаимодействия с файловой системой и файлами в частности;
4. загрузки и отправки файлов по сети;
5. взаимодействия с функциями ОС (например, обращения к переменным среды).

Также в исходном коде web-шелла могут содержаться команды для оболочки ОС сервера, позволяющие, например, взаимодействовать с файлами. К таким можно отнести, например, *ls*, *rmdir*, *touch*. Также подозрительными можно считать команды *arp*, *awk*, *ipconfig* и другие.

Все указанные типы функций и команд сами по себе абсолютно безвредны и часто используются в легальных программах, как, например, функции для преобразования текста. Однако некоторые из функций достаточно редко встречаются в исходных кодах приложений, чего нельзя сказать про web-шеллы. Яркий тому пример – функции для вычисления и выполнения некоторого выражения во время работы программы. Чем шире набор подозрительных функций и команд в исходном коде программы, тем выше вероятность, что это web-шелл.

### **1.2 Преобразование текста программы**

#### **1.2.1 Обоснование выбора характеристики**

Для затруднения обнаружения вредоносного кода злоумышленник может преобразовать его так, что его предназначение становится непонятным для человека или программы-анализатора. На практике web-шеллы часто содержат в себе преобразованные фрагменты кода, над которыми совершается обратное преобразование, после чего они выполняются.

Таким образом, поиск подозрительных функций или команд может дать нулевой результат при подобной организации web-шелла. Поэтому необходимо уметь выявлять наличие различного рода преобразований в коде программы.

Важно отметить, что преобразование само по себе не является признаком вредоносной программы. Преобразование может использоваться при сжатии информации (архивы, изображения, данные), а также при передаче легальных веб-скриптов по сети. Поэтому обнаружение преобразования в коде ещё не говорит о том, что он вредоносен.

#### **1.2.2 Способы выявления преобразования**

Для выявления преобразования будет использоваться два подхода: простой поиск hex-кода и вычисление энтропии текста.

Нередко используется достаточно простое преобразование, когда исходный текст преобразуется в советующий ему hex-код. Такое преобразование легко распутать, однако текст в таком виде становится нечитаемым и непонятным, а именно этого и хочет добиться злоумышленник. Обнаружить hex-код нетрудно ввиду особенностей его написания в коде программы, для этого будет достаточно обычного поиска по тексту специфических последовательностей символов.

Более сложные преобразования (например, base64) обнаружить таким образом куда сложнее, в этом случае используется другой подход. Предполагается, что преобразование изменяет статистические свойства текста, и, следовательно, энтропия текста может быть использована для его выявления. Энтропия в данном контексте относится к статистической мере неопределенности символов в тексте.

Ниже будет приведено обоснование применения энтропии в контексте обнаружения преобразований.

#### **1.2.3 Информационная энтропия**

Информационная энтропия представляет собой меру неопределенности в системе, особенно в отношении непредсказуемости появления символов первичного алфавита. В контексте передачи сообщений, энтропия численно соответствует количеству информации, приходящемуся на каждый символ сообщения при отсутствии информационных потерь.

Для расчета энтропии в работе будет использоваться формула Шеннона [2]. Пусть для каждого символа алфавита определена вероятность его появления () в информационном сообщении. Тогда энтропия сообщения, построенного на алфавите , будет вычисляться по формуле:

,

где – некоторая положительная константа.

Это соотношение было получено Шенноном для определения среднего количества информации в сообщении с произвольными вероятностями появления значений символов [3].

#### **1.2.4 Избыточность языка**

Избыточность языка – это статистическая величина, обозначающая избыточность информации, содержащейся в тексте на определённом языке. Для формального определения избыточности введем понятия нормы (энтропии) и абсолютной нормы (абсолютной энтропии) языка [4].

Для данного языка норма (энтропия) языка равна:

,

где – энтропия сообщения , – длина сообщения. Для примера, норма обычного английского языка при больших принимает различные значения от 1,0 до 1,5 бит на букву.

Абсолютная норма (абсолютная энтропия) языка равна максимальному количеству битов, которое может быть передано каждым символом при условии, что все последовательности символов равновероятны. Если в алфавите языка используется различных букв, то абсолютная норма равна:

.

Для английского языка с 26 буквами в алфавите абсолютная норма равна  ~ 4,7 бит на букву. Действительная норма английского языка намного меньше абсолютной, поскольку естественные языки обладают высокой избыточностью.

Избыточность языка, обозначаемая , определяется так:

.

Принимая норму английского языка за 1,3, избыточность составит 3,4 бит на букву, то есть каждая английская буква содержит 3,4 бита избыточной информации. Следовательно, у сообщения ASCII (где каждый символ кодируется одним байтом), состоящего только из английских букв, количество избыточной информации на каждый байт составляет 6.7 бит.

#### **1.2.5 Избыточность преобразованного текста**

Избыточность достаточна высока в текстах, написанных на естественных языках. В тексте, представляющем собой исходный код программы, где гораздо активнее используются знаки пунктуации, цифры, а также специальные символы, избыточность ниже. Но еще ниже избыточность в преобразованных (например, зашифрованных или сжатых) текстах, поскольку одно из характерных свойств алгоритма преобразования – перераспределение частот встречаемости значений байтов таким образом, что вероятность появления каждого из них примерно одинакова.

Более-менее одинаковые частоты встречаемости приводят к заметному росту энтропии, что означает сильное уменьшение избыточности. И чем больше символов будет использоваться для преобразования текста, тем ближе к нулю будет избыточность и тем ближе будет значение энтропии к значению абсолютной энтропии.

На этом факте и основана идея обнаружения преобразования. Чем меньше избыточность текста, тем вероятнее, что он преобразован.

### **1.3 Обфускация кода программы**

#### **1.3.1 Определение**

Обфускация – процесс изменения кода программы, в результате которого он приобретает вид, трудный для понимания – при этом программа сохраняет свои функции [5]. Обфускация может применяется в следующих целях:

* защита интеллектуальной собственности (сокрытие алгоритмов и логики работы программы);
* конфиденциальность разработки (предотвращение кражи и дальнейшего анализа кода);
* предотвращение кибератак (сокрытие уязвимостей, которые могут быть использованы злоумышленниками для взлома системы).

Ранее уже было отмечено, что преобразования могут быть использованы злоумышленником для сокрытия вредоносного кода. Обфускация является преобразованием, которое сохраняет вычисляемую программой функцию, но при этом придает программе такую форму, что извлечение из текста программы ключевой информации об алгоритмах и структурах данных, реализованных в этой программе, становится трудоёмкой задачей [6]. Это обстоятельство делает обфускацию одним из методов, применяемых в разработке вредоносных программ, в том числе и web-шеллах.

Обфусцирование кода может иметь разную степень сложности. В простейшем случае, это изменение имён переменных, функций, методов, классов и т.п. Более сложные способы обфускации в работе рассматриваться не будут.

#### **1.3.2 Способы выявления обфускации**

Для обнаружения обфускации можно так же использовать расчёт энтропии, однако в ходе выполнения работы было установлено, что в общем случае значение энтропии текста, преобразованного, например, популярными алгоритмами кодирования base64 или ASCII85, отличается от значения энтропии обфусцированного кода в большую сторону. Для отделения преобразованных текстов от непреобразованных необходимо установить границу значений энтропии, переходя которую, текст будет считаться преобразованным. Устанавливая границу такой, чтобы преобразованными считались в том числе и обфусцированные коды программ, довольно немалая часть непреобразованных текстов (кодов) будут признаваться преобразованными, то есть будет много ложных срабатываний. Чтобы уменьшить их число, для обнаружения обфусцированного кода будет использоваться другой подход.

Изменение имён выглядит следующим образом: оригинальное имя, составленное, как правило, из осмысленных слов английского алфавита, преобразуется в бессмысленный набор символов, допустимых для именования. Поскольку изменение имён почти всегда используется при обфускации, именно оно и послужит её признаком.

Способ обнаружения обфускации заключается в следующем:

1. простым поиском найти в коде имена переменных, констант, функций, методов и классов;
2. попытаться вычленить из каждого имени осмысленные слова;
3. если имя состоит преимущественно из осмысленных слов, обфускация к нему не применялась;
4. в конечном счёте по количеству обфусцированных имен станет ясно, была ли примерена обфускация к коду программы, или нет.

Применение отдельного подхода к поиску обфусцированного кода позволяет повысить точность работы алгоритмов.

#### **1.3.3 Расстояние Левенштейна**

Для определения того, является ли отдельно взятое слово осмысленным, будет использоваться следующий подход: в заранее подготовленном словаре будет искаться слово, похожее на рассматриваемое. Степень похожести определяется допустимым расстоянием Левенштейна между словами.

Расстояние Левенштейна – это метрика, измеряющая по модулю разность между двумя последовательностями символов. Она определяется как минимальное количество односимвольных операций (а именно вставки, удаления, замены), необходимых для превращения одной последовательности символов в другую [7].

Пусть и – два слова длиной и соответственно, тогда расстояние Левенштейна рассчитывается по следующей рекуррентной формуле:

, где

*deleteCost*, *insertCost*, *replaceCost* – стоимости удаления, вставки и замены символа соответственно. В нашем случае нет смысла выставлять приоритет операций, поэтому все стоимости будут равны единице.

## **2. Реализация**

### **2.1 Реализация алгоритма обнаружения преобразования**

#### **2.1.1 Описание алгоритма**

На практике анализируемый текст может быть преобразован лишь частично, например, на 5%. Эта часть текста будет характеризоваться высокой энтропией. Остальные 95% будут иметь невысокую энтропию. В результате может получится так, что энтропия по тексту в целом тоже будет невысокой, так как 95% «затрут» остальные 5%. Во избежание подобной ситуации было придумано два решения: расчёт энтропии методом «скользящего» окна и игнорирование при расчёте всех английских[[1]](#footnote-1) слов и некоторых синтаксических конструкций используемого языка. Это позволит локализовать преимущественно высокоэнтропийные области в тексте.

Работу алгоритма можно поделить на следующие этапы:

1. удаление из текста всех символов, не относящихся к преобразованным фрагментам;
2. вычисление энтропии текста в целом;
3. вычисление энтропии текста методом «скользящего» окна;
4. вынесение вердикта по результатам вычисленных значений энтропии.

В следующих разделах каждый этап алгоритма будет рассмотрен отдельно.

#### **2.1.2 Поиск и удаление непреобразованной части текста**

Для поиска непреобразованной части текста используется словарь английских слов объёмом примерно 3800 слов.

Алгоритм поиска и удаления заключается в следующем:

* + - 1. весь текст разбивается на токены;
      2. все буквенные токены разбиваются на слова, которые в свою очередь сопоставляются со словарём слов: если большая часть слов была найдена в словаре, то значение токена признается непреобразованным, в противном случае значение определяется как преобразованное;
      3. формируется последовательность токенов:
         1. если предыдущий и текущий буквенные токены были признаны преобразованными, то эти токены, а также все небуквенные токены между ними, будут сохранены;
         2. если один из двух указанных буквенных токенов был признан непреобразованным, сохранены будут только те токены, которые ближе к преобразованному;
         3. в случае, когда оба буквенных токена определены как непреобразованные, то они, а также все небуквенные токены между ними, будут проигнорированы, и их значения не попадут в результирующий текст;
      4. из последовательности токенов формируется итоговый вариант текста.

Токен представлен классом *Token*, хранит в себе значение (последовательность байт) и тип (буквы, цифры, символ нижнего подчеркивания, прочие символы). Вычлененное из токена слово оборачивается в класс *Word*. Для формирования последовательности токенов и вычленения из них слов используются классы *TokenExtractor* и *WordExtractor.*

#### **2.1.3 Взаимодействие со словарём**

Взаимодействие со словарём осуществляется посредством класса *WordDictService*, позволяющий эффективно (за ) проверить наличие слова в словаре.

#### **2.1.4 Расчёт энтропии**

Класс *EntropyCalculator* производит расчёт энтропии. Класс *EntropyAnalyzer* использует *EntropyCalculator* для расчёта энтропии по тексту в целом и методом «скользящего» окна. Размер окна составляет часть от размера текста, величина сдвига окна – от его размера. Метод *EntropyAnalyzer.window\_analyze* двигает окно вдоль всего текста, в каждом «окне» рассчитывает энтропию попавших в него символов и выдает результаты расчётов в виде массива.

Указанные размер окна и величина его сдвига были получены в ходе «обучения» программы и признаны оптимальными.

#### **2.1.5 Вынесение вердикта по значению энтропии**

Для обнаружения преобразования в тексте используется следующий подход. Сначала проверяется уровень энтропии текста в целом. Если он оказывается ниже определённого значения, при котором текст считался бы преобразованным, проверяется, сколько «окон» можно отнести к преобразованным, то есть какой процент «окон» содержит в себе текст с высокой энтропией. Ниже приведены следующие поля класса *EncryptionDeterminatorByEntropy*, который непосредственно отвечает за обнаружение преобразования:

* *\_window\_encryption\_border* – все «окна» с энтропией равной или большей значения этого поля будут считаться «окнами» с высокой энтропией;
* *\_unconditional\_lower\_bound\_of\_entropy* – текст с энтропией равной или большей значения этого поля будет считаться преобразованным;
* *\_conditional\_lower\_bound\_of\_entropy* – текст с энтропией равной или большей значения этого поля будет считаться преобразованным, если процент окон с высокой энтропией будет не меньше, чем *\_percent\_of\_entropy\_vals\_for\_window*;
* *\_percent\_of\_entropy\_vals\_for\_window* – см. выше;
* *\_upper\_bound\_of\_entropy\_strict* и *\_upper\_bound\_of\_entropy\_optimal* – текст с энтропией равной или большей значения одного из этих полей (какое из полей будет выбрано, определяется режимом работы алгоритма: строгим или оптимальным) будет считаться непреобразованным; эти границы необходимы, чтобы исключить ложные срабатывания на высокоэнтропийных, но легальных файлах (например, сжатые архивы или изображения).

Значения полей, указанных выше, получены в ходе «обучения» программы и были признаны оптимальными на основании метрик.

### **2.2 Реализация алгоритма обнаружения обфускации**

#### **2.2.1 Поиск имён**

Для поиска имён переменных и констант, функций и методов, классов используются регулярные выражения, инкапсулированные в классы *VariableSearcher*, *FunctionSearcher* и *ClassSearcher* соответственно. Эти классы являются наследниками класса *AbstractSearcher*. Метод *AbstractSearcher.get\_name\_iter* принимает на вход анализируемый текст, а возвращает итератор по именам, найденным в нём. Все имена представлены экземплярами класса *Name*.

Прежде, чем передать имя (*Name*) на анализ алгоритму обнаружения обфускации, оно переводится в экземпляр класса *NameInfo* с помощью класса *NameProcessor*. *NameInfo* хранит в себе информацию о том, какие слова содержит имя, сколько в имени букв и цифр. Вычленение слов из имени осуществляется посредством описанных ранее классов: *Token*, *TokenExtractor*, *Word*, *WordExtractor*.

#### **2.2.2 Поиск похожего слова в словаре**

Ранее было описано, что для определения того, является ли слово осмысленным, для него ищется похожее слово в словаре, имеющее расстояние Левенштейна не больше указанного.

Расчёт метрики Левенштейна осуществляет класс *CalculatorLevenshteinMetric*, принимая на вход два слова (экземпляры класса *Word*) и возвращая целое число – редакционное расстояние между ними. Для расчёта используется алгоритм Вагнера-Фишера. Данная реализация имеет временную сложность и пространственную – , где – длины передаваемых слов.

Поиск похожего слова в словаре осуществляет класс *SearcherByLevenshteinMetric*. Алгоритм поиска:

1. проверить, есть ли проверяемое слово в словаре (сложность ), если да, алгоритм завершает работу;
2. если слово не нашлось, алгоритм продолжает работу: на основе длины проверяемого слова и заранее переданного множителя рассчитывается максимально допустимое редакционное расстояние (расстояние равно округленному до ближайшего целого значению произведения длины слова и множителя);
3. далее выстраивается порядок длин слов, среди которых будет искаться похожее, при этом в первую очередь будут рассматриваться слова, «близкие» к проверяемому слову (например, если проверяемое слово содержит 7 букв, а максимально допустимое расстояние равно 3, то будет выстроен следующий порядок: 7, 6, 8, 5, 9, 4, 10; то есть сначала будут рассмотрены все слова той же длины (7), что и проверяемое слово, затем слова, отличающиеся по длине на один символ (6 и 8), на два символа (5 и 9) и, наконец, на три символа (4 и 10); рассматривать слова меньшей/большей длины (начиная с 3 и 11 соответственно) не имеет смысла, поскольку расстояние Левенштейна для таких слов будет уже больше 3);
4. выстроив порядок длин слов, алгоритм в соответствии с ним перебирает все слова, пытаясь найти похожее;
5. как только будет найдено слово, с расстоянием не больше указанного, алгоритм завершит свою работу, вернув слово и расстояние; если же были рассмотрены все допустимые слова, и при этом не было найдено ни одного похожего, алгоритм вернет пустое значение, означающее, что похожих слов в словаре нет.

Словарь, используемый классом *SearcherByLevenshteinMetric*, это тот же словарь, который используется для поиска и удаления непреобразованной части текста в задаче обнаружения преобразования. Работа со словарём осуществляется с помощью класса *WordDictService*.

#### **2.2.3 Обнаружение обфускации**

Алгоритм обнаружения обфускации инкапсулирован в классе *ObfuscationDeterminator*. Класс содержит три поля, определяющих работу алгоритма:

* *\_obf\_text\_border* – если доля обфусцированных имен в коде окажется выше этого значения, текст признается обфусцированным;
* *\_obf\_name\_border* – если доля неосмысленных слов в имени окажется выше этого значения, имя признается обфусцированным;
* *\_max\_non\_obf\_count\_digits* – максимальное количество цифр в имени, при котором оно ещё не считается обфусцированным, и по нему будет проводится проверка на наличие осмысленных слов; если цифр в имени больше значения этого поля, оно признается обфусцированным.

Значения полей, указанных выше, получены в ходе «обучения» программы и были признаны оптимальными на основании метрик.

Работу алгоритма можно поделить на следующие этапы:

1. найти все имена в коде;
2. для каждого имени определить, является ли оно обфусцированным (для этого используется характеристика имени или поиск похожего слова в словаре);
3. по количеству обфусцированных имён определить, является ли код обфусцированным.

### **2.3 Реализация алгоритма поиска подозрительных функций и команд**

#### **2.3.1 Классификация подозрительных инструкций**

Для удобства все подозрительные инструкции были разнесены в различные группы. Для этого было придумано две классификации: по уровню угрозы и по типу или области действия.

Классификация по уровню угрозы:

* опасные (DANGEROUS) – инструкции, представляющие наибольшую опасность (в среде программистов существуют общие рекомендации не использовать подобные инструкции в коде своих программ во избежание возможного появления в них уязвимостей);
* подозрительные (SUSPICIOUS) – инструкции, не представляющие сами по себе какой-либо угрозы, но которые часто используются и применяются в web-шеллах;
* требующие внимания (PAY\_ATTENTION) – инструкции, которые очень часто используются в легальных программах, однако могут быть использованы злоумышленником (например, загрузка файлов по сети).

Классификация по типу/области действия:

* команды для оболочек ОС (COMMAND);
* преобразование (ENCRYPT);
* вычисление переданного выражения (EXECUTION);
* взаимодействие с файлами (FILES);
* подключение внешних скриптов/библиотек, вызывающих подозрение (IMPORT);
* загрузка/отправка файлов по сети (NET);
* взаимодействие с функциями ОС (OS);
* остальные подозрительные инструкции (GENERAL);

Ниже приведена таблица, раскрывающая связь приведённых классификаций.

|  |  |  |  |
| --- | --- | --- | --- |
|  | DANGEROUS | SUSPICIOUS | PAY\_ATTENTION |
| EXECUTION | + |  | + |
| COMMAND |  | + |  |
| ENCRYPT |  | + |  |
| FILES |  | + | + |
| IMPORT |  | + | + |
| NET |  | + | + |
| OS |  | + |  |
| GENERAL |  |  | + |

Таблица 7.1.1 Связь классификаций подозрительных инструкций

#### **2.3.2 Поиск подозрительных инструкций**

Задача поиска подозрительных функций и команд возложена на класс *SuspySearcher*. Непосредственно поиск осуществляется с помощью регулярных выражений. Метод *SuspySearcher.search* получает на вход анализируемый текст, после чего, перебирая шаблоны регулярных выражений, ищет все возможные совпадения тексте. Одновременно с обнаружением подозрительной инструкции проводится её классификация.

## **3 Проверка корректности и эффективности работы алгоритмов**

### **3.1 Алгоритм обнаружения преобразования**

#### **3.1.1 Описание выборки**

Для проверки работы алгоритма обнаружения преобразования была собрана выборка из 92 файлов, из которых 46 преобразованы.

В выборку непреобразованных файлов были включены файлы, представляющие собой исходные коды программ, реализованных с использованием следующих языков программирования: JavaScript, PHP, Python, Ruby, C#. Также представлены файлы HTML, CSS, SQL, XML, bash. Кроме того, в выборку были включены архивы, и сжатые (zip через LZMA) и не сжатые (zip, tar), а также файлы-изображения (png, webp).

В выборку преобразованных файлов были включены файлы, к которым были применены следующие алгоритмы:

* base32, base64, base85 (ASCII85), base122;
* rot13, rot18, rot47;
* AES;
* DES (triple).

Кроме этого проверялась обнаружение файла, содержащего hex-код.

При этом для каждого способа преобразования, кроме rotXX, было произведено несколько степеней преобразования, файлы были преобразованы на 5%, 10%, 30%, 50% и 100%.

#### **3.1.2 Проверка работы алгоритма**

Результат работы алгоритма по описанной выборке файлов (под срабатыванием подразумевается обнаружение преобразования, под пропуском – не обнаружение):

* верное срабатывание – 43 файла;
* ложное срабатывание – 15 файлов;
* верный пропуск – 31 файл;
* ложный пропуск – 3 файла.

Все сжатые файлы (архивы и изображения) были признаны непреобразованными, поскольку алгоритм работал в оптимальном режиме, при котором файлы с крайне высокой энтропией не включаются в множество преобразованных. Все ложные пропуски обусловлены файлами, преобразованными по алгоритму rotXX. Поскольку rotXX, в отличие от baseXX, AES или DES, используется для простой обфускации текста путём сдвига по алфавиту, энтропия оригинального и преобразованного текстов будет примерно одинакова. Вследствие этого алгоритм определяет файлы, преобразованные rotXX, как непреобразованные.

Метрики по выборке:

* точность – 0,74;
* полнота – 0,93;
* F-score – 0,83.

Следует отметить, что F-score мог бы быть чуть выше, однако параметры алгоритма подобраны так, чтобы полнота была наибольшей, вследствие чего снижается точность.

### **3.2 Алгоритм обнаружения обфускации**

#### **3.2.1 Описание выборки**

Для проверки работы алгоритма обнаружения преобразования была собрана выборка из 26 файлов, из которых 13 обфусцированы.

В выборку представлены файлы исходных кодов программ, реализованных с использованием следующих языков программирования: JavaScript, PHP, Python.

#### **3.2 Проверка работы алгоритма обнаружения преобразования**

Результат работы алгоритма по описанной выборке файлов (под срабатыванием подразумевается обнаружение обфускации, под пропуском – не обнаружение):

* верное срабатывание – 13 файлов;
* ложное срабатывание – 1 файл;
* верный пропуск – 12 файлов;
* ложный пропуск – 0 файлов.

Метрики по выборке:

* точность – 0,93;
* полнота – 1.00;
* F-score – 0,96.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения работы были выявлены характерные особенности web-шеллов, разработаны и реализованы подходы к их обнаружению.

Результатом выполнения работы является работающая программа, выполненная с помощью языка программирования Python, позволяющая обнаруживать различного рода преобразования (включая обфускацию) и подозрительные функции и команды в исходных кодах программ, выполненных на языках программирования PHP, JavaScript, Python, Ruby, C#. Также в работе представлены результаты проведения тестов и метрики, отражающие корректность и эффективность работы программы.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Энциклопедия «Касперского»: сайт. URL: <https://encyclopedia.kaspersky.ru/glossary/web-shell> (дата обращения: 28.05.2024).
2. Шеннон К. Работы по теории информации и кибернетике. — М.: [Издательство иностранной литературы](https://ru.wikipedia.org/wiki/%D0%98%D0%B7%D0%B4%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D1%82%D0%B2%D0%BE_%D0%B8%D0%BD%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D0%BD%D0%BE%D0%B9_%D0%BB%D0%B8%D1%82%D0%B5%D1%80%D0%B0%D1%82%D1%83%D1%80%D1%8B), 1963. — 829 с.
3. Цымбал В.П Теория информации и кодирование: Учебник. — 4-е изд., перераб. и доп. — К.: Вища шк., 1992. — 263 с.:ил.
4. [Шнайер Б.](https://ru.wikipedia.org/wiki/%D0%A8%D0%BD%D0%B0%D0%B9%D0%B5%D1%80,_%D0%91%D1%80%D1%8E%D1%81) Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке Си — М.: Триумф, 2002. — 816 с.
5. Энциклопедия «Касперского»: сайт. URL: [https://encyclopedia.kaspersky.ru/glossary/obfuscation](https://encyclopedia.kaspersky.ru/glossary/obfuscation/) (дата обращения: 28.05.2024).
6. Большая российская энциклопедия: сайт / учредитель АНО «Национальный научно-образовательный центр «Большая российская энциклопедия». URL: <https://bigenc.ru/c/obfuskatsiia-fe1e3c> (дата обращения: 28.05.2024).
7. Романовский И.В. Дискретный анализ: Учебное пособие для студентов, специализирующихся по прикладной математике и информатике. — 4-е изд., испр. и доп. — СПб.: Невский Диалект; БХВ-Петербург, 2008. — 336  с.:ил.

1. Поскольку алгоритм предназначен для поиска преобразований в исходных кодах программ, разумно «вырезать» из текста только английские слова, так как в большинстве случаев исходный код либо не содержит буквенные символы каких-либо других естественных языков, либо содержит в небольшом количестве, что не должно сильно влиять на энтропию. [↑](#footnote-ref-1)