

Министерство науки и высшего образования Российской Федерации
САНКТ-ПЕТЕРБУРГСКИЙ
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

П. Д. Зегжда М. О. Калинин

ОСНОВЫ
ИНФОРМАЦИОННОЙ БЕЗОПАСНОСТИ
ВВЕДЕНИЕ В ПРОФЕССИОНАЛЬНУЮ
ДЕЯТЕЛЬНОСТЬ
ЛАБОРАТОРНЫЙ ПРАКТИКУМ

*Рекомендовано СЗРО учебно-методического объединения
в системе высшего образования
по УГСНП 10.00.00 «Информационная безопасность»
в качестве учебного пособия для студентов
высших учебных заведений, обучающихся по направлению
«Информационная безопасность»
по программам подготовки бакалавров,
магистров, специалистов*



ПОЛИТЕХ-ПРЕСС

Санкт-Петербургский
политехнический университет
Петра Великого

Санкт-Петербург
2019

Выполняется загрузка документа

3: 100%

4: 100%

УДК 004.056(075.8)

347

Р е ц е н з е н т ы:

Доктор технических наук, доцент, заведующий кафедрой технологий защиты информации Санкт-Петербургского государственного университета аэрокосмического приборостроения *С. В. Бессатеев*

Доктор технических наук, профессор, профессор кафедры электротехники и автоматики Государственного университета морского и речного флота имени адмирала С. О. Макарова *И. А. Сикарев*

Зегжда П. Д. Основы информационной безопасности. Введение в профессиональную деятельность. Лабораторный практикум : учеб. пособие / П. Д. Зегжда, М. О. Калинин. – СПб. : ПОЛИТЕХ-ПРЕСС, 2019. – 112 с.

Содержит основные теоретические и практические сведения о современных способах защиты информации, технологиях и программных решениях, применяемых для защиты информационных и компьютерных систем, по учебной дисциплине «Введение в профессиональную деятельность», а также методики проведения лабораторных работ по учебной дисциплине «Основы информационной безопасности» для студентов кафедры информационной безопасности компьютерных систем Санкт-Петербургского политехнического университета Петра Великого.

Предназначен для студентов специальностей 10.05.01, 10.05.03, 10.05.04, а также при подготовке студентов, обучающихся другим техническим специальностям в области информационных технологий.

Табл. 13. Ил. 14. Библиог.: 15 назв.

Печатается по решению

Совета по издательской деятельности Ученого совета
Санкт-Петербургского политехнического университета Петра Великого.

ISBN 978-5-7422-6436-1

© Зегжда П. Д., Калинин М. О., 2019

© Санкт-Петербургский политехнический
университет Петра Великого, 2019

СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	4
<i>Лабораторная работа 1. Математические примитивы криптографии.....</i>	<i>7</i>
<i>Лабораторная работа 2. Основы частотного криptoанализа.....</i>	<i>17</i>
<i>Лабораторная работа 3. Изучение программных уязвимостей типа "переполнение буфера"</i>	<i>21</i>
<i>Лабораторная работа 4. Защита от встраиваемых потайных ходов...</i>	<i>26</i>
<i>Лабораторная работа 5. Анализ вредоносных программных средств.....</i>	<i>29</i>
<i>Лабораторная работа 6. Защита программного обеспечения от нелегального использования.....</i>	<i>40</i>
<i>Лабораторная работа 7. Кодирование и упаковка данных.....</i>	<i>48</i>
<i>Лабораторная работа 8. Основы стеганографической защиты информации.....</i>	<i>63</i>
<i>Лабораторная работа 9. Методы контроля целостности.....</i>	<i>70</i>
<i>Лабораторная работа 10. Методы надежного хранения и передачи информации.....</i>	<i>77</i>
<i>Лабораторная работа 11. Механизм аутентификации пользователей.</i>	<i>83</i>
<i>Лабораторная работа 12. Система контроля доступа</i>	<i>89</i>
<i>Лабораторная работа 13. Защита web-сервера от несанкционированного доступа.....</i>	<i>94</i>
<i>Лабораторная работа 14. Защита от угроз нарушения безопасности типа "отказ в обслуживании".....</i>	<i>101</i>
РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА.....	108
ПРИЛОЖЕНИЕ. Описание генераторов компьютерных вирусов.....	109

ВВЕДЕНИЕ

Лабораторный практикум разработан для методического обеспечения учебной дисциплины "Основы информационной безопасности" кафедры информационной безопасности компьютерных систем Санкт-Петербургского политехнического университета Петра Великого в рамках подготовки студентов по специальностям 10.05.01 "Компьютерная безопасность", 10.05.03 "Информационная безопасность автоматизированных систем", 10.05.04 "Информационно-аналитические системы". Практикум также содержит теоретический материал о современных способах защиты информации, технологиях и программных решениях, применяемых для защиты информационных и компьютерных систем, который может быть использован в учебной дисциплине "Введение в профессиональную деятельность", читаемой для студентов тех же специальностей.

Цель практикума – приобретение базовых профессиональных знаний об основных направлениях информационной безопасности, получение и закрепление навыков по применению современных алгоритмов, методов и средств защиты информации. В издании рассмотрены характерные задачи информационной безопасности, решаемые при внедрении современных цифровых технологий.

Практикум состоит из 14 лабораторных работ, посвященных изучению базовых технологий защиты информации. Лабораторные работы сгруппированы в две параллельные образовательные траектории (нечетные и четные номера работ). В парных работах студенты приобретают эквивалентные навыки на разных объектах изучения. Такой подход позволяет гибко подойти к формированию вариантов заданий. В процессе освоения практического материала допустимо следовать одной из предложенных траекторий или составлять индивидуальную комбинацию работ. Разнообразие затронутых тем позволяет гибко изменять сложность и объем практического курса с учетом начального уровня знаний студентов, требований и задач обучения.

Первая пара работ знакомит студентов с криптографическим направлением профессиональной деятельности. Цель лабораторной работы 1 – понимание и навык выполнения основных математических

примитивов, используемых в криптоалгоритмах и криптопротоколах. Лабораторная работа 2 вырабатывает умение применять криptoанализ на базе частотного раскрытия шифrogramм.

Вторая пара работ нацелена на изучение способов борьбы с уязвимостями в программном обеспечении. В лабораторной работе 3 исследуется уязвимость типа "переполнение буфера", способы ее эксплуатации, методы анализа и устранения. В лабораторной работе 4 изучаются специально встраиваемые дефекты программ, обеспечивающие нарушителям потайные ходы (backdoor) к удаленному управлению операционной системой, способы их реализации и устранения.

Третья пара лабораторных работ направлена на приобретение навыков специалиста по информационной безопасности в аналитической деятельности, исследовании исполняемых кодов и восстановлении алгоритмов. Лабораторная работа 5 нацелена на изучение проблемы вирусного конструирования, заражения и распространения вредоносного кода. В лабораторной работе 6 рассматриваются и изучаются простейшие методы проверки легальности авторских программных продуктов, базовые способы как обхода, так и усиления такой защиты. В обеих работах выполняется низкоуровневый анализ бинарных кодов.

Четвертая пара работ знакомит студентов с методами кодированного сокрытия информации. В лабораторной работе 7 изучаются методы кодирования информации, эффект сжатия и свойства цифровых кодов. В лабораторной работе 8 исследуются стеганографические методы, изучаются возможности и свойства стегоконтейнеров.

Пятая пара работ посвящена ознакомлению с методами обеспечения достоверности информации в ненадежных цифровых средах. В лабораторной работе 9 изучаются контрольное суммирование, методы и свойства циклических кодов, которые используются при проверке целостности. Лабораторная работа 10 рассматривает задачу помехоустойчивого избыточного кодирования, которое позволяет за счет изменения структуры представления данных обнаруживать в них ошибки и восстанавливать исходную информацию.

Шестая пара работ посвящена практике применения штатных средств защиты в операционных системах. При выполнении лабораторной работы 11 изучается механизм аутентификации, способы его

компрометации и усиления. Лабораторная работа 12 знакомит с принципами контроля доступа и управления полномочиями пользователей.

Седьмая пара работ знакомит с направлением сетевой безопасности. Лабораторная работа 13 посвящена обеспечению безопасности при доступе к сайтам на примере управления защитными механизмами web-сервера Apache. Лабораторная работа 14 посвящена анализу угроз доступности компьютерных систем, исследованию свойств компьютерных атак типа "отказ в обслуживании" и изучению методов защиты от таких воздействий.

Основными задачами практикума являются выработка у обучающихся понимания актуальных угроз безопасности и способов их осуществления, используемых злоумышленниками, изучение базовых технологий защиты информации, приобретение практических навыков в создании и применении аналитического и математического инструментария, в эксплуатации средств защиты информации, в решении современных проблем компьютерной безопасности.

С целью предотвращения каких-либо воздействий на реальные вычислительные системы лабораторный практикум выполняется в изолированной среде виртуальной учебной лаборатории под руководством преподавателя.

Лабораторная работа 1

МАТЕМАТИЧЕСКИЕ ПРИМИТИВЫ КРИПТОГРАФИИ

Цель работы – приобретение расчетных навыков в модульной арифметике, используемой в криптографических алгоритмах и протоколах, ознакомление с математическими вычислениями, используемыми для сокрытия сообщений на примерах алгоритма шифрования RSA и ранцевой крипtosистемы Меркля-Хеллмана.

Теоретические сведения

Модульная арифметика

Основная задача целочисленной арифметики – определение делимости чисел, то есть делится ли одно число на другое без остатка. Число a делится на число b (a кратно b), если можно подобрать такое число c , чтобы выполнялось равенство $a=bc$ ($a, b, c \in \mathbb{Z}, b \neq 0$).

В общем случае деления с остатком для натуральных чисел a и b (делимого и делителя) единственным образом находятся числа q и r (частное и остаток), обладающие свойствами:

$$a = b \times q + r, \quad 0 \leq r < b.$$

Число r называется вычетом числа a по модулю b , что записывается как

$$a = r(\text{mod } b) \quad \text{или} \quad a \equiv r(b)$$

и читается как " a сравнимо с r по модулю b ". В языках программирования известны операции нахождения частного и остатка от деления целых чисел. Условие $0 \leq r < b$ показывает, что остаток неотрицателен и меньше делителя. При $r=0$ получается определение делимости чисел.

Примеры: $853 = 20 \times 43 + 13$, то есть $853 = 13(\text{mod } 20)$; $5 = 0(\text{mod } 5)$; $2 = 2(\text{mod } 5)$.

Шифр Цезаря

Пример применения модульной арифметики в криптографии – *шифр замены*. Предположим, что все шифруемые числа неотрицательны, но меньше некоторого числа m , и таким же условиям удовлетворяют числа, получаемые в результате шифрования. Это позволяет считать, что числа являются элементами кольца вычетов $\mathbb{Z}/m\mathbb{Z}$. Шифрующая функция при

этом может рассматриваться как взаимно однозначное отображение колец вычетов $f: \mathbb{Z}/m\mathbb{Z} \rightarrow \mathbb{Z}/m\mathbb{Z}$.

Число $f(x)$ представляет собой сообщение x в зашифрованном виде.

Шифр замены соответствует отображению $f: x \rightarrow x+k \pmod{m}$ при фиксированном целом k , где m – размер алфавита; k – сдвиг по алфавиту. Подобный шифр использовал Юлий Цезарь (латинский алфавит, $k=3$).

Алгоритм Евклида

В математике и криптографии одной из востребованных прикладных задач является поиск ответа на вопрос, имеет ли данный набор чисел общий делитель, отличный от единицы – нетривиальный делитель, названный так, поскольку единица является делителем любого набора чисел. Числа, которые не имеют нетривиального общего делителя, называются *взаимно простыми*.

Обычно ищут *наибольший общий делитель* (НОД), который делится на все остальные делители и, таким образом, содержит их в себе. Например, набор чисел 72, 84, 132, 144 имеет общие делители 2, 3, 4, 6, 12. Наибольший общий делитель равен 12 (записывается "НОД=12"). Он делится на все общие делители.

Известно решение задачи поиска НОД через разложение каждого из чисел на простые множители. Этот алгоритм становится неприемлемым при увеличении разрядности чисел. Например, для разложения числа, превышающего миллиард, на простые множители потребуется таблица первых сотен или тысяч простых чисел, а построение такой таблицы – задача более трудоемкая, чем исходная. Кроме того, хранение таблицы простых чисел сопряжено с затратами ресурсов. Следовательно, данный метод неприменим в криптографии.

Существует достаточно эффективный метод нахождения НОД – алгоритм Евклида с вычитанием. Из набора выбираются любые два ненулевых числа, и большее из них (или любое, если числа равны) заменяется разностью этих чисел. Этот процесс повторяется до тех пор, пока не останется одно ненулевое число. Это число и будет НОД исходного набора, состоящего из натуральных чисел.

Выбирая для вычитания различные пары, можно получить разные алгоритмы. Все эти алгоритмы будут решать поставленную задачу. Для

того чтобы в этом убедиться, необходимо обосновать корректность алгоритмов, то есть доказать, что каждый такой алгоритм обязательно закончит свою работу, что полученный в результате работы алгоритма набор будет содержать только одно ненулевое число и что это число будет наибольшим общим делителем исходного набора.

Корректность метода доказывается с помощью двух утверждений: если числа a_1 и a_2 делятся на b , то и их разность делится на b , и числа набора остаются неотрицательными. Процесс изменения набора обязательно закончится, так как после каждого вычитания сумма чисел набора уменьшается. Эта сумма – всегда положительное целое число, следовательно, процесс не может длиться бесконечно (очевидно, что число шагов не превышает суммы чисел исходного набора).

Простые и составные числа

Одна из важных задач криптографии – определение того, является ли число простым или составным. Известен способ убедиться в этом, не разлагая число на множители.

Согласно малой теореме Ферма, если число N простое, то для любого целого a , не делящегося на N , выполняется сравнение $a^{N-1} \equiv 1 \pmod{N}$. Если при каком-то a сравнение нарушается, можно утверждать, что N – составное.

Проверка не требует больших вычислений. Проблема в том, как найти для составного N целое число a , не удовлетворяющее сравнению $a^{N-1} \equiv 1 \pmod{N}$. Можно применить полный перебор на отрезке $(1, N)$, но такой подход не всегда результативен. Имеются составные числа N , обладающие указанным свойством для любого целого a с условием $\text{НОД}(a, N)=1$. Такие числа называются *числами Кармайкла*.

В 1976 г. Миллер предложил следующий способ. Если N – простое число, $N-1 = 2^s t$, где t нечетно, то согласно малой теореме Ферма для каждого a с условием $\text{НОД}(a, N)=1$ хотя бы одна из скобок в произведении

$$(a^t - 1)(a^t + 1)(a^{2t} + 1) \dots (a^{2^{s-1}t} + 1) = a^{N-1} - 1$$

делится на N . Обращение этого свойства можно использовать, чтобы отличить составные числа от простых.

Пусть N – нечетное составное число, $N-1 = 2^s t$, где t нечетно. Назовем целое число a , $1 < a < N$, "хорошим" для N , если нарушается одно из двух условий:

N не делится на a ;

$a^t \equiv 1 \pmod{N}$ или существует целое k , $0 \leq k < s$, такое что $a^{2^{k-1}t} \equiv -1 \pmod{N}$.

Следовательно, для простого N не существует хороших чисел a . Если же N составное, то, как доказал Рабин, их существует не менее $3(N-1)/4$. Это позволяет построить вероятностный алгоритм, отличающий составные числа от простых:

1. Выберем случайным образом a , $1 < a < N$, и проверим для него указанные выше свойства.

2. Если хотя бы одно из них нарушается, то число N – составное.

3. Если выполнены оба условия, возвращаемся к шагу 1 алгоритма.

4. Составное число не будет определено как составное после однократного выполнения шагов 1...3 с вероятностью не большей $1/4$. А вероятность не определить его после k повторений не превосходит $1/4^k$.

Поиск числа, обратного по модулю

Поиска числа, обратного по модулю некоторому заданному, широко применяется в криптографии. Для каждого целого числа c и положительного числа d найдется единственная пара целых чисел Q (частное) и s (остаток) таких, что $c = dQ + s$, где $0 \leq s < d$. Частное обозначается как $Q = [c/d]$.

Для нахождения НОД двух заданных положительных чисел s и t ($s > t$) применяется рассмотренный ранее алгоритм Евклида:

$$s = s_0 = Q_1 t_0 + t_1,$$

$$t = t_0 = Q_2 t_1 + t_2,$$

$$t_1 = Q_3 t_2 + t_3, \dots,$$

$$t_{n-2} = Q_n t_{n-1} + t_n,$$

$$t_{n-1} = Q_{n+1} t_n,$$

где остановка наступает при получении нулевого остатка. Последний ненулевой остаток t_n равен НОД.

В матричном представлении шаги алгоритма Евклида выглядят следующим образом $\begin{pmatrix} s \\ t \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -Q_r \end{pmatrix} \begin{pmatrix} s_{r-1} \\ t_{r-1} \end{pmatrix}$, где $Q_r = [s_r - 1/t_r - 1]$.

Таким образом, если $t_n = 0$, то $\begin{pmatrix} s_n \\ 0 \end{pmatrix} = \left\{ \prod_{k=n}^1 \begin{pmatrix} 0 & 1 \\ 1 & -Q_k \end{pmatrix} \right\} \begin{pmatrix} s \\ t \end{pmatrix} = A_n \begin{pmatrix} s \\ t \end{pmatrix}$.

Число d , обратное по модулю m числу e (то есть $ed \equiv 1 \pmod{m}$), является элементом A_{12} матрицы A_n , где $s=m$, $t=e$.

Алгоритм шифрования RSA

Кольцо вычетов Z/nZ , где n – произведение двух больших простых чисел, $n = pq$, является коммутативным кольцом с единицей.

Мультиликативная группа $(Z/nZ)^*$ этого кольца является абелевой и циклической, состоит из ненулевых чисел, меньших n и взаимно простых с n . Остатки от деления образующей группы $(Z/nZ)^*$ на p и q равны соответственно образующим мультиликативным группам полей F_p^* и F_q^* . Поэтому образующая группы $(Z/nZ)^*$ может быть найдена по китайской теореме об остатках.

Порядок группы равен значению функции Эйлера от n : $\#(Z/nZ)^* = \phi(n) = \phi(p)\phi(q) = (p-1)(q-1)$. По определению порядка группы для любого элемента x группы имеет место равенство $x^{\phi(n)} \equiv 1 \pmod{n}$.

Любой элемент кольца $a \in (Z/nZ)$ может быть единственным образом представлен в виде $a \pmod{p}$ и $a \pmod{q}$ и обратно по китайской теореме об остатках.

Если $a \pmod{p} = 0$, то a принадлежит идеалу (p) и не является элементом группы $(Z/nZ)^*$, при этом элемент a образует в (Z/nZ) мультиликативную группу, изоморфную F_q^* . Единичным элементом в этой группе является элемент, сравнимый с 1 по модулю q и сравнимый с 0 по модулю p .

Сложность нахождения порядка группы полиноминально эквивалентна сложности разложения числа n .

Алгоритм шифрования RSA – криптосистема с открытым ключом. Ключ содержит две составляющие: открытую, которую можно публиковать, и секретную. Открытый ключ содержит число n и показатель шифрования e . Секретный ключ содержит показатель дешифрования d .

Показатели шифрования и дешифрования связаны между собой равенством $ed = 1 \pmod{\phi(n)}$.

Любой блок текста x , представленный числом, меньшим n и взаимно простым с n , может быть зашифрован любым пользователем с помощью открытого ключа: $x^e \equiv y \pmod{n}$.

Дешифрование криптограммы y осуществляется только владельцем секретного ключа d : $y^d \equiv x^{ed} \equiv x^{\phi(n)+1} \equiv x^{\phi(n)}x \equiv x \pmod{n}$.

Цифровая подпись выполняется аналогично шифрованию и дешифрованию. Ключ создания подписи является конфиденциальной информацией данного пользователя и содержит показатель d . Соответствующий ему ключ проверки подписи содержит число n и показатель e . Этот ключ является общедоступным.

Для вычисления подписи s для текста x , представленного числом, меньшим n и взаимно простым с n , подписывающий возводит его в степень d : $s \equiv x^d \pmod{n}$. Подписаный текст представляет собой пару (x, s) .

Для проверки подписи проверяющий вычисляет обратное преобразование и проверяет равенство $x \equiv s^e \pmod{n}$. Если равенство выполняется, то подпись верна.

Поскольку операция возведения в степень коммутативна, то есть $(a^x)^y \equiv (a^y)^x \equiv a^{xy} \pmod{n}$, то можно использовать алгоритм Диффи-Хеллмана для установления сеансового ключа в кольце вычетов по модулю составного ключа. Для этого пользователи договариваются об общем числе n и образующей а циклической группы большого простого порядка.

Пользователь A вырабатывает случайное число $x < \phi(n)$, вычисляет $a^x \pmod{n}$ и посыпает пользователю B .

Пользователь B вырабатывает случайное число $y < \phi(n)$, вычисляет $a^y \pmod{n}$ и посыпает пользователю A .

Затем каждый из пользователей возводит полученное сообщение в степень со своим показателем и получает число $(a^x)^y \equiv (a^y)^x \equiv a^{xy} \pmod{n}$, являющееся сеансовым ключом.

Ранцевая криптосистема Меркля-Хеллмана

В 1978 г. Меркль и Хеллман предложили использовать задачу об укладке ранца (рюкзака) для асимметричного шифрования. Она относится к классу NP-полных задач и формулируется следующим образом.

Дано N предметов, V – вместимость рюкзака, $W = \{w_1, \dots, w_N\}$ – набор положительных целых весов предметов, $P = \{p_1, \dots, p_N\}$ – набор положительных целых стоимостей предметов. Необходимо найти набор бинарных величин $B = \{b_1, \dots, b_N\}$, где $b_i = 1$, если предмет n_i включен в набор, $b_i = 0$, если предмет n_i не включен в набор.

При этом $b_1w_1 + \dots + b_Nw_N \leq V$ и сумма $b_1p_1 + \dots + b_Np_N$ максимальна.

Допустим, заданы веса предметов $\{2, 7, 9, 11, 15, 22, 34, 45\}$. Рюкзак вместимостью 18 можно упаковать, складывая предметы весом 2, 7 и 9, но невозможно упаковать рюкзак вместимостью 19.

Предметы из набора выбираются с помощью блока открытого текста, длина которого (в битах) равна количеству предметов в наборе. Биты открытого текста соответствуют значениям b , а текст является полученным суммарным весом. Пример шифrogramмы, полученной с использованием алгоритма укладки ранца, представлен в табл. 1.

В качестве закрытого ключа применяется сверхвозрастающая последовательность. Сверхвозрастающей называется последовательность, в которой каждый последующий член больше суммы всех предыдущих. Например, последовательность $\{2, 3, 6, 13, 27, 52, 105, 210\}$ сверхвозрастающая.

Таблица 1
Пример шифrogramмы на основе укладки ранца

Открытый текст	1 1 0 0 1 1 1 0	1 1 0 0 1 0 0 0	1 1 0 0 0 0 0 1
Рюкзак (ключ)	2 7 9 11 15 22 34 45	2 7 9 11 15 22 34 45	2 7 9 11 15 22 34 45
Шифrogramма	80 (2+7+15+22+34)	24 (2+7+15)	54 (2+7+4)

Открытый ключ представляет собой несверхвозрастающую (нормальную) последовательность и создается на основе закрытого ключа. Чтобы получить открытый ключ, необходимо все значения закрытого ключа умножить на число n по модулю m . Значение m должно быть больше суммы всех чисел последовательности. Например, дана последовательность $\{1, 3, 5, 10, 21, 42, 84, 167\}$. Число $m = 335$, множитель $n = 19$ (взаимно простое с 335). Результат получения закрытого и открытого ключа представлен в табл. 2.

Для того чтобы расшифровать сообщение получателю необходимо найти число n^{-1} , такое что $nn^{-1} \pmod m = 1$. Для вычисления обратных чисел

по модулю применяется алгоритм Евклида. После определения обратного числа каждое значение шифrogramмы c_i умножается на n^{-1} по модулю m и с помощью закрытого ключа определяются биты открытого текста.

Таблица 2

Получение закрытого и открытого ключа

Закрытый ключ k_i	1	3	5	10	21	42	84	167
Открытый ключ $(k_i n) \bmod m$	19	57	95	190	64	128	256	158

Предположим, существует секретное сообщение "ОИБ". Символы данного сообщения представлены в бинарном виде в соответствии с кодировкой Windows 1251. Результат шифрования представлен в табл. 3.

Для того чтобы расшифровать сообщение получателю необходимо найти число n^{-1} , такое что $nn^{-1} \pmod{m} = 1$. Для вычисления обратных чисел по модулю применяется алгоритм Евклида. После определения обратного числа каждое значение шифrogramмы c_i умножается на n^{-1} по модулю m и с помощью закрытого ключа определяются биты открытого текста.

Таблица 3

Результат шифрования открытым ключом

Сообщение		Расчет весов	Шифrogramма c_i
Символ	Бинарный код		
О	1100 1110	19+57+64+128+256	524
И	1100 1000	19+57+64	140
Б	1100 0001	19+57+158	234

В рассматриваемом примере сверхвозрастающая последовательность $\{1, 3, 5, 10, 21, 42, 84, 167\}$, $m = 335$, $n = 19$. Значение $n^{-1}=194$, т.к. $19 \times 194 \pmod{335} = 1$. Пример расшифрования представлен в табл. 4.

Таблица 4

Пример расшифрования

Шифrogramма c_i	$(c_i n^{-1}) \bmod m$	Расчет весов	Сообщение	
			Бинарный код	Символ
524	151	1+3+21+42+84	1100 1110	О
140	25	1+3+21	1100 1000	И
234	162	1+3+158	1100 0001	Б

Порядок выполнения работы

1. Вычислить число по формуле: $((N_{ep.} + N_{cn.})^{11} + \Phi_3)(mod\ 11)$, где $N_{ep.}$ – номер учебной группы, $N_{cn.}$ – порядковый номер в списке группы, Φ_3 – порядковый номер в алфавите третьей буквы фамилии (эти же обозначения будут присутствовать в последующих пунктах).
2. Выбрав и зафиксировав число k в функции шифрования алгоритма Цезаря, зашифровать строку, составленную из фамилии, имени и отчества, записанных кириллицей. Число m – размерность алфавита.
3. Вычислить число $A = (N_{ep.}(8+N_{cn.}(mod\ 7)))^2$. Рассчитать число $B = \text{ЧЧММГГГГ}$, где ЧЧ, ММ и ГГГГ – число, месяц и год рождения. Найти $\text{НОД}(A, B(mod\ 95)+900)$, $\text{НОД}(A, (B+50)(mod\ 97)+700)$, $\text{НОД}(A, (B+20)(mod\ 101)+1500)$, $(B-40)(mod\ 103)+2500$. Указать в отчете последовательность вычислений методом Евклида.
4. Выбрать составное число N , $N > 10000$ (например, любое четное число). Методом Миллера доказать, что число составное. Выбрать простое число N_1 , $100 < N_1 < 1000$ и методом Миллера показать, что оно простое. Провести сравнительный анализ временной сложности обоих доказательств путем подсчета количества шагов и расчетных вероятностей для обоих чисел.
5. Для изучения генерации ключей в алгоритме RSA выбрать два любых больших ($10^2 < N < 10^6$) простых числа p и q : $\text{НОД}(p, q) = 1$. Вычислить число $n = pq$. Вычислить порядок группы $\phi(n)$. Выбрать показатель шифрования e такой, что $\text{НОД}(e, p-1) = \text{НОД}(e, q-1) = 1$. Вычислить, используя матричный способ, показатель d по формуле $ed = 1(mod\ \phi(n))$.
6. Выбрать произвольный текст x (использовать ASCII-кодировку символов) Зашифровать текст x . Для этого вычислить $x^e(mod\ n)$, получив тем самым зашифрованный текст y . Расшифровать текст y с помощью ключа d , вычислив $y^d(mod\ n)$. Сравнить расшифрованный текст с исходным.
7. Подписать текст x цифровой подписью s , где $s = xd(mod\ n)$. Проверить подпись, вычислив обратное преобразование и проверив равенство $x = se(mod\ n)$.
8. Для моделирования процесса установления сеансового ключа выбрать любое число a . Смоделировать действия пользователя A : выбрать случайным образом число x , $x < \phi(n)$; найти число $A = a^x(mod\ n)$.

Смоделировать действия пользователя B : выбрать случайное число y , $y < \phi(n)$; вычислить число $B = a^y \pmod{n}$. Установить сеансовый ключ: каждый из пользователей возводит полученное сообщение в степень со своим показателем: пользователь A вычисляет $B^x \pmod{n}$, пользователь B – $A^y \pmod{n}$. Полученные значения образуют сеансовый ключ. Проверить равенство полученных ключей пользователей. Вычислить $a^x y \pmod{n}$ и проверить равенство $(ax)y = (ay)x = a^x y \pmod{n}$.

9. Разработать утилиту шифрования и дешифрования с помощью алгоритма Меркля-Хеллмана.

Содержание отчета

1. Исходные данные для каждого пункта работы.
2. Произведенные математические расчеты и результаты вычислений.
3. Листинги программ.
4. Ответы на контрольные вопросы.
5. Выводы по работе.

Контрольные вопросы

1. Что такое вычет? На чем основан алгоритм шифрования Цезаря?
2. Каковы особенности чисел Кармайкла?
3. Перечислите основные свойства мультипликативной группы кольца вычетов по модулю pq .
4. Почему порядок группы $(\mathbb{Z}/n\mathbb{Z})^*$ должен иметь большой простой делитель?
5. Опишите алгоритм расчета кодов символов при декодировании шифrogramм согласно алгоритму Меркля-Хеллмана.

Лабораторная работа 2

ОСНОВЫ ЧАСТОТНОГО КРИПТОАНАЛИЗА

Цель работы – приобретение навыков криптоанализа, ознакомление со способом дешифрования криптограмм на примере применения метода частотного криптоанализа.

Теоретические сведения

Шифр моноалфавитной подстановки является одним из самых древних шифров (например, шифр Цезаря). Выбирается нормативный алфавит – набор символов, которые будут использоваться для составления сообщений (например, буквы русского алфавита и пробел). Затем определяется алфавит шифрования путем взаимооднозначного соответствия с символами нормативного алфавита. Алфавит шифрования может состоять из произвольных символов, в том числе и из символов нормативного алфавита. Чтобы зашифровать исходное сообщение, вместо символа открытого текста подставляется соответствующий ему символ алфавита шифрования (например, зашифрованное с помощью подстановки, указанной в табл. 5, слово "ЗВЕЗДА" записывается как "ИАТИЗН").

Таблица 5

Пример моноалфавитной подстановки

Нормативный алфавит	А	Б	В	Г	Д	Е	Ж	З	И	Й	К
Алфавит шифрования	Н	К	А	Л	З	Т	П	И	О	Р	Б

Все естественные языки имеют характерное частотное распределение символов. Например, буква "О" встречается в русском языке чаще других, а буква "Ъ" – самая редкая (рис. 1). Моноалфавитные подстановки не нарушают частот появления символов, характерных для данного языка. Частотный характер использования букв в криптограммах, использующих моноалфавитную подстановку, позволяет криптоаналитику получить открытый текст при помощи частотного анализа.

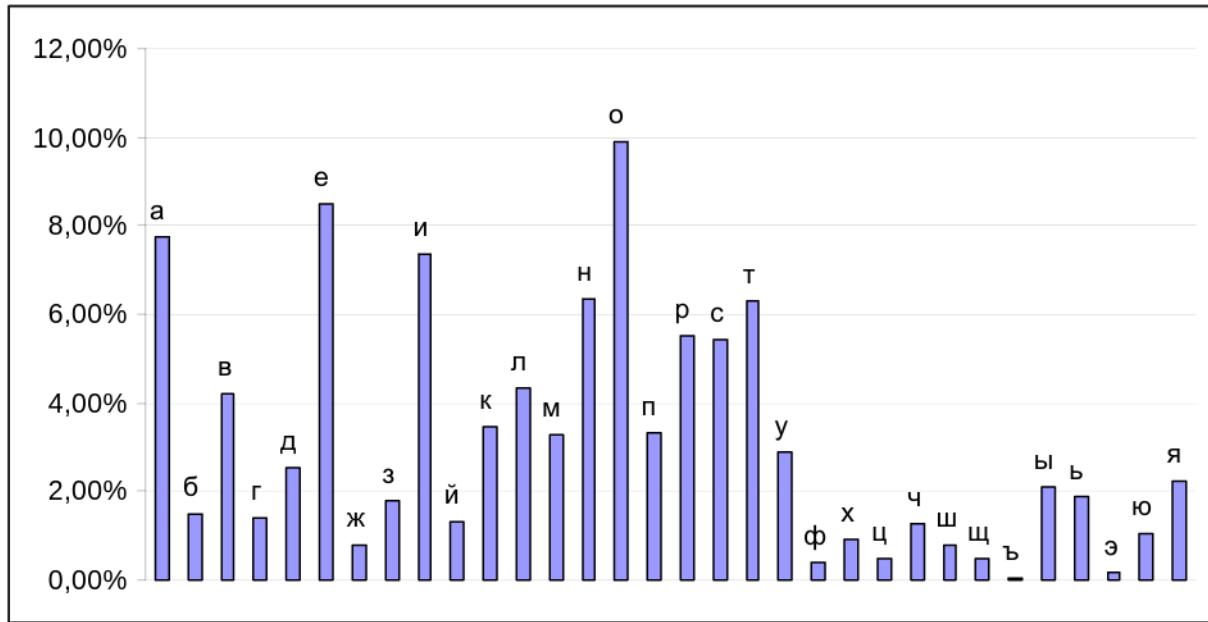


Рис. 1. Частотный спектр русского алфавита

Для этого необходимо сопоставить частоты появления символов шифра с вероятностями появления букв используемого алфавита. После этого наиболее частые символы криптограммы заменяются на наиболее вероятные символы алфавита. Остальные замены производятся на основе вероятных слов и знания синтаксических правил языка.

Например, имеется криптограмма "КЩРНСЙШЩХДТ РБУТЦПФЮСНЫ ШЙ ҮЙЛБ НСЙСТНСТОБНДЩМЩ ЙШИЙЖТЛЙ СБДНСЙ ШЩ СЩЖЕДЩ БНЖТ ЩШ РЩНСЙСЩОЩ РЖТШШИГ". Статистический анализ данной криптограммы представлен в табл. 6.

На основании статистики заменим самую частую букву криптограммы самой частой буквой русского языка: "Ш" заменим на "о" (прописные буквы будут обозначать символы криптограммы, а строчные – расшифрованные символы): "КоРНСЙШоХДТ РБУТЦПФЮСНЫ ШЙ ҮЙЛБ НСЙСТНСТОБНДоМо ЙШИЙЖТЛЙ СБДНСЙ Шо СоЖЕДо БНЖТ оШ РоНСЙСоОШо РЖТШШИГ".

Рассмотрим слова: "Шо" и "оШ", "Ш" может обозначать либо "т" либо "н", иначе одно или оба этих слова не имеют значения. Сделаем замену "Ш" на "н". Если позднее появится противоречие, то мы вернемся на этот шаг и сделаем другую замену: "Ш" на "т": "КоРНСЙноХДТ РБУТЦПФЮСНЫ нЙ ҮЙЛБ НСЙСТНСТОБНДоМо ЙнЙЖТЛЙ СБДНСЙ но СоЖЕДо БНЖТ он РоНСЙСоОно РЖТннИГ".

Таблица 6

Пример частотного анализа криптограммы

Символ	Частота
Щ	0,111
С	0,101
Й	0,091
Н	0,081
Ш	0,081
Т	0,071
Б	0,051
Р	0,040
Д	0,040
Ж	0,040

Можно предположить, что пятое слово оканчивается на "ОГО" и является прилагательным или причастием. Заменим "М" на "г".

Слово, следующее за прилагательным или причастием, скорее всего является существительным и оканчивается на "А". Заменим "Й" на "а": "КоРНСаноХДТ РБУТЦПФЮСНЫ на ЪалБ НСаСТНСТОБНДого анаЖТЛа СБДНСа но СоЖЕДо БНЖТ он РоНСаСоОно РЖТнниГ".

Четвертым словом является предлог "на", поэтому следующее слово оканчивается, скорее всего, на букву "е". Заменяем "Б" на "е".

Шестое слово имеет вид: "ана---а". Это может быть слово "анализа". Заменим "Ж" на "л", "Т" на "и", "Л" на "з": "КоРНСаноХДи РeУиЦПФЮСНЫ на Ъазе НСаСиНСиОеHДого анализа СеДНСа но СолЕДо еНли он РоНСаСоОно РлинниГ".

Слово "е-ли" является словом "если", а последняя подстрока криптограммы "-линн--" похоже на слово "длинный". Сделаем соответствующие замены: "Ь" на "б", "Н" на "с", "Р" на "д", "И" на "ы", "Г" на "й": "КодсСаноХДи дeУиЦПФЮСсы на базе сCaCисСиOесДого анализа СеДсСа но СолЕДо если он досCaСоОно длинный".

Последующие замены выполняются аналогично. В итоге, получаем дешифрованный текст: "подстановки дешифруются на базе статистического анализа текста но только если он достаточно длинный".

Ключ шифрования представлен в табл. 7 (сверху расположены символы нормативного алфавита, снизу – алфавита шифрования).

Таблица 7

Пример ключа шифрования

А	Б	В	Г	Д	Е	З	И	Й	К	Л	Н	О	П	Р	С	Т	У	Ф	Ч	Ш	Ы	Ь	Ю	Я
Й	Ь	Х	М	Р	Б	Л	Т	Г	Д	Ж	Ш	Щ	К	П	Н	С	Ф	Ц	О	У	И	Е	Ю	Ы

Порядок выполнения работы

1. Разработать программу, реализующую функции инструмента криptoаналитика. Программа должна выполнять следующие функции:

анализ частоты букв во входном файле и вывод предполагаемых замен в соответствии с частотами распределения букв русского алфавита;

вывод на экран всех слов, сгруппированных по количеству букв;

вывод на экран всех слов, сгруппированных по количеству нерасшифрованных на данный момент букв;

отображение криптограммы с указанием расшифрованного на данный момент текста;

возможность замены букв в криптограмме;

хранение и откат истории замены букв в криптограмме;

интерфейс взаимодействия с пользователем.

С использованием разработанной программы расшифровать криптограмму, выданную преподавателем.

2. Доработать программу, реализовав функцию автоматической замены букв. Проверить и оценить результат ее работы.

Содержание отчета

1. Исходная криптограмма.
2. Листинг разработанной программы.
3. Распределение частот букв в криптограмме.
4. Результаты работы программы с обоснованием последовательности анализа.
5. Дешифрованное сообщение, таблица подстановки.
6. Ответы на контрольные вопросы.
7. Выводы по работе.

Контрольные вопросы

1. Что такое шифр моноалфавитной подстановки?
2. Укажите недостатки шифра моноалфавитной подстановки.
3. Какова сложность дешифрации методом прямого перебора для сообщения, зашифрованного шифром моноалфавитной подстановки?
4. Какие условия упрощают частотный анализ?
5. Получится ли правильно дешифрованная криптограмма, если произвести все замены в соответствии с частотами появления букв в русском языке? Ответ обосновать.

Лабораторная работа 3

ИЗУЧЕНИЕ ПРОГРАММНЫХ УЯЗВИМОСТЕЙ ТИПА "ПЕРЕПОЛНЕНИЕ БУФЕРА"

Цель работы – приобретение навыков по прикладному анализу программных уязвимостей типа "переполнение буфера" и по предотвращению их эксплуатации.

Теоретические сведения

Переполнение буфера (buffer overflow) – одна из наиболее распространенных уязвимостей программного обеспечения, возникающих из-за отсутствия контроля границ массивов со стороны компилятора и операционной системы.

Память в программе адресуется относительно точки входа в программу. Память сегментирована и состоит из сегмента кода, сегмента данных и сегмента стека. Сегмент кода содержит инструкции, исполняемые процессором, где адрес следующей выполняемой инструкции указан в регистре EIP. Сегмент данных содержит данные, используемые командами программы. В сегменте стека хранятся переменные функций, переменные окружения и аргументы, которые передаются программе.

Стек – структура данных, в которой последний помещенный в стек объект извлекается из него первым (принцип LIFO, last in-first out,

"последним пришел-первым вышел"). Для доступа к сегменту стека используется указатель стека – регистр ESP, содержащий адрес вершины (наименьший адрес) стека. В архитектуре Intel стек "растет" вниз, то есть используются адреса памяти в сторону уменьшения номеров адресов.

В большинстве языков программирования стек используется при передаче данных в вызываемую процедуру. При вызове процедуры (с помощью инструкции CALL) в стек помещается текущее значение регистра EIP, а по окончании работы процедуры (с помощью инструкции RET) это значение восстанавливается, и процессор продолжает работу с того места, где он остановился перед вызовом процедуры.

Функция –часть кода программы, которая вызывается, выполняется и затем возвращается к предыдущему процессу исполнения. Если функция содержит аргументы, то под них в стеке выделяется память.

Например, в коде:

```
#include <string.h>
int sample_overflow(int argc, char *argv[])
{
    char buf[500]; //переменная
    if(argc > 1)
        strcpy(buf, argv[1]); //уязвимая функция
    return 0;
}
```

под переменную *buf* в стеке выделяется 500 байт (стек при нормальном выполнении программы представлен на рис. 2). Уязвимость типа "переполнение буфера" заключается в том, что аргумент, передаваемый функции и копируемый в *buf*, может быть больше 500 байт. В этом случае, по причине отсутствия механизмов проверки, данные, содержащиеся в аргументе, переполняют буфер и затирают данные в стеке, которые следуют за буфером (рис. 3).

На рис. 3 данные из буфера перекрывают адрес возврата из функции. Таким образом, злоумышленник может составить определенный код, при котором на месте адреса возврата в стеке окажется новый адрес, указывающий на инструкции созданного им шелл-кода, управление на который передается в результате переполнения.



Рис. 2. Структура стека на момент вызова функции *sample_overflow*

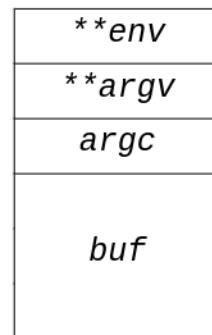


Рис. 3. Структура стека после переполнения массива buf в функции *sample_overflow*

Переполнение буфера вызывает различные нарушения в работе программы:

аварийное завершение (отказ) программы;

зависание программы;

выполнение произвольного кода с правами пользователя, от имени которого выполняется программа

В общем виде программные уязвимости типа "переполнение буфера" эксплуатируются следующим образом:

1. Нарушителем вычисляется длина буфера. Для этого определяется, при какой длине входного сообщения возникнет ошибка переполнения. Длина входной строки увеличивается до тех пор, пока не появляется системное сообщение об ошибке (*segmentation fault*). Как только оно появляется, фиксируется адрес, по которому произошла ошибка. Зафиксированный адрес – адрес, на который должен быть перезаписан адрес возврата из функции. Таким образом становятся известны байты входной последовательности, которые при переполнении перезапишут адрес возврата. Байты адреса располагаются в обратном порядке в силу особенностей адресации в архитектуре Intel (формат записи "*big endian*").

2. Нарушитель создает шелл-код. В простейшем случае разрабатывается программа на языке C,зывающая необходимую функцию. Полученный EXE-файл открывается с помощью дизассемблера (например, *OllyDbg*) и из него переносится шелл-код, включающий набор ассемблерных инструкций, сгенерированных компилятором для вызова

нужной функции, в передаваемый буфер (строку) сразу после адреса возврата.

3. Нарушитель организует передачу управления на шелл-код. В регистре ESP хранится указатель на начало стека, то есть указатель на байт, который является следующим после адреса возврата. Для передачи управления на шелл-код адрес возврата меняется на команду CALL ESP (FF D4) в системной библиотеке (*.DLL), которая автоматически подгружается к любому процессу (например, KERNEL32.DLL). Для этого с помощью дизассемблера (например, *IDA Disassembler*) выполняется поиск последовательности байт "FF D4" в библиотеке KERNEL32.DLL. Полученный адрес записывается на место байт, которые затирают адрес возврата.

Стандартные функции языка С, такие как *strcpy*, *sprintf*, *gets*, работают со строками без указания их размеров. Для создания безопасного кода следует использовать безопасные версии библиотечных функций, например, *strncpy*, *snprintf*, в параметрах которых указывается длина обрабатываемых строк, что позволяет контролировать границы массивов.

Существуют программные средства, которые способны автоматически выполнять действия, имитирующие переполнение буфера на этапе отладки программы. Существуют и применяются специальные утилиты для автоматического анализа исходных текстов программ на предмет поиска фрагментов кода, уязвимых к атакам типа "переполнение буфера".

Порядок выполнения работы

Во избежание потери информации и нарушения работоспособности используемых систем все действия следует выполнять в виртуальной среде.

1. Разработать уязвимую к "переполнению буфера" программу, которая принимает в качестве аргумента строку и копирует ее в буфер.
2. Создать шелл-код, который удаляет значение заданного ключа системного реестра.
3. Разработать программу, которая по известному размеру буфера уязвимой программы компонует адрес возврата и шелл-код (с возможностью выбора удаляемого ключа системного реестра) и затем

запускает уязвимую программу, передавая в качестве аргумента сгенерированную входную строку, которая вызывает переполнение буфера и удаление заданного ключа системного реестра.

4. Запустить разработанную программу и убедиться в удалении ключа системного реестра.

5. Установить утилиту статического анализа исходных кодов, например, *CPPcat*, поддерживающий языки C/C++.

6. Проверить с помощью утилиты статического анализа исходных кодов программу, разработанную в п. 3. Зафиксировать результаты анализа.

7. Используя полученный в п. 6 отчет утилиты анализа, устраниТЬ ошибки в исходном коде разработанной программы.

8. Повторить выполнение п. 6 и убедиться в исправлении ошибок.

9. Запустить модифицированную программу и убедиться в том, что шелл-код не срабатывает и ключ системного реестра не удаляется.

Содержание отчета

1. Листинг уязвимой программы.
2. Листинг программы-генератора шелл-кода.
3. Листинг программы, модифицированной по результатам статического анализа безопасности исходного кода.
4. Шелл-код в байтовом виде и в виде ассемблерных инструкций.
5. Описание стека в штатном и уязвимом режимах функционирования программы.
6. Результаты наблюдений за поведением программы до и после эксплуатации уязвимости.
7. Отчет, сгенерированный утилитой статического анализа исходных кодов.
8. Ответы на контрольные вопросы.
9. Выводы по работе.

Контрольные вопросы

1. Какие основные этапы выделяют при реализации атаки, эксплуатирующей уязвимость типа "переполнение буфера"?

2. Почему стек является уязвимым при передаче параметров функции?

3. Однаков ли адрес инструкций CALL ESP в различных версиях операционной системы Windows?

4. Могут ли инструкции быть выполнены где-либо кроме сегмента кода? Ответ обоснуйте.

5. Опишите работу механизма защиты Data Execution Prevention. Каким образом он позволяет защитить систему от эксплуатации уязвимостей типа "переполнение буфера"?

Лабораторная работа 4

ЗАЩИТА ОТ ВСТРАИВАЕМЫХ ПОТАЙНЫХ ХОДОВ

Цель работы – приобретение навыков по анализу структуры, функциональности и угроз специально встраиваемого дефекта программного продукта – потайного хода (backdoor), а также изучение методов защиты от уязвимости такого вида.

Теоретические сведения

Программа-шпион (spyware) – программное обеспечение, скрытно собирающее информацию о пользователе (персональные данные, настройки операционной системы, статистику работы и пр.).

Шпионское программное обеспечение применяется для проведения маркетинговых исследований, распространения целевой рекламы, деструктивных воздействий. Информация о пользователе и системе может существенно упростить последующую компьютерную атаку.

Шпионские программы распространяются в результате:

посещения web-сайтов (для установки шпионских программ активно используются ActiveX-компоненты);

установки бесплатных и условно-бесплатных программ (например, кодек DivX содержит утилиту для скрытной загрузки и установки SpyWare.Gator).

Большинство шпионских программ не уведомляют об этом пользователей, эксплуатируя встроенные в программу дефекты потайные ходы (backdoor) – программные компоненты, специально оставленные или встроенные в программу с целью скрытого получения несанкционированного доступа к данным или для удаленного управления вычислительной системой.

Категории потайных ходов:

потайные ходы, построенные по технологии клиент-сервер. Такой потайной ход состоит из двух и более программ – небольшого агента, скрытно устанавливаемого на компьютер-жертву, и программы управления на компьютере злоумышленника;

потайные ходы, использующие для удаленного управления Telnet-, web- или IRC-серверы. Для управления таким потайным ходом не требуется специальное клиентское программное обеспечение, так как оно уже входит в состав операционных систем или пакетов прикладных программ, установленных на компьютерах пользователей.

Потайной ход, например, позволяет копировать файлы с пораженного компьютера и на него, получать удаленный доступ к реестру, выполнять системные операции (перезагрузку системы, создание новых сетевых ресурсов, модификацию паролей и т.п.). Опасность потайных ходов увеличилась в последнее время в связи с тем, что многие современные сетевые черви или содержат в себе компоненты, обеспечивающие срабатывание потайного хода, или устанавливают их после заражения для последующей эксплуатации. Эти действия позволяют использовать компьютер пользователя, устройства Интернета вещей, киберфизические системы для сканирования сетей, проведения с них массированных сетевых атак, скрытого сбора информации и телеметрии, а также для дальнейшего распространения по другим узлам сети.

В большинстве случаев потайной ход после установки открывает сетевой порт и ожидает входящего соединения. Соответственно, основными защитными мерами, направленными на борьбу с угрозами эксплуатации потайных ходов, являются контроль активных процессов и регулярная проверка сетевых портов. При этом следует учитывать, что потайные ходы могут маскировать свое присутствие в системе, открывать

сетевые порты только в заранее определенное время и самостоятельно инициировать исходящее соединение с удаленным сервером.

Порядок выполнения работы

Во избежание потери информации и нарушения работоспособности используемых систем все действия следует выполнять в виртуальной среде.

1. Реализовать клиент-серверную программу, использующую сокеты для сетевого соединения. Клиент при запуске открывает определенный порт и ожидает входящего соединения. Сервер соединяется с клиентом и передает ему имя файла, который должен быть удален.
2. Запустить программу-клиент на одном компьютере, а программу-сервер на другом. Удаленно выполнить удаление файла на компьютере, на котором установлена программа-клиент.
3. Расширить функциональность клиента, добавив процедуру маскировки. Для этого модифицировать код программы-клиента таким образом, чтобы она копировала себя в системный каталог, автоматически запускалась при старте операционной системы, не имела окна.
4. Запустить модифицированную программу-клиент. Запустить *Диспетчер задач*, убедиться в присутствии процесса клиента в списке активных задач. Повторить выполнение п. 2.
5. Изменить функциональность программ клиента и сервера таким образом, чтобы программа-клиент при запуске самостоятельно осуществляла регулярные попытки соединения с программой-сервером, а программа-сервер ожидала удаленного подключения от программы-клиента. Повторить выполнение п. 2.
6. Установить программный межсетевой экран (например, *Commodo Internet Security*) на компьютер, на котором работает программа-клиент. Зафиксировать список открытых сетевых портов.
7. Повторить выполнение п. 2, зафиксировав поведение межсетевого экрана. Разрешив сетевое взаимодействие программам клиента и сервера, зафиксировать сетевые порты, которые ими использовались.
8. Средствами межсетевого экрана заблокировать данные сетевые порты, повторить выполнение п. 2, отметив произошедшие изменения в работе программ клиента и сервера.

Содержание отчета

1. Листинги разработанных программ.
2. Описание методов маскировки работы программы-клиента.
3. Описание добавления программы-клиента в автозагрузку.
4. Результаты наблюдений за поведением межсетевого экрана до и после блокировки сетевых портов.
5. Список открытых и блокируемых сетевых портов.
6. Ответы на контрольные вопросы.
7. Выводы по работе.

Контрольные вопросы

1. Какие угрозы несут потайные ходы?
2. Как можно обнаружить потайной ход, если он открывает порт только на короткие промежутки времени, которые заранее не известны?
3. Можно ли использовать потайные ходы для организации распределенной атаки типа "отказ в обслуживании"?
4. Каким образом можно удалить процесс программы-клиента из списка задач *Диспетчера задач*?
5. Опишите схему работы потайного хода, использующего для удаленного управления Telnet-сервер.

Лабораторная работа 5

АНАЛИЗ ВРЕДОНОСНЫХ ПРОГРАММНЫХ СРЕДСТВ

Цель работы – приобретение практических навыков по противодействию вредоносным программным средствам на примере компьютерных вирусов.

Теоретические сведения

Компьютерный вирус – программа, которая может заражать другие программы, модифицируя их посредством добавления своей, возможно измененной, копии. Заражая программы, вирус распространяется в

компьютерной системе или сети, используя системные или пользовательские полномочия. Зараженная программа также действует как вирус, что способствует быстрому инфицированию.

Однозначно идентифицирует вредоносное программное обеспечение его сигнатура – характерная информационная последовательность (например, текстовая строка, цепочка выполняемых функций, специфический фрагмент графа передачи управления). Выявление таких участков в коде программы до сих пор является основным способом обнаружения вирусов, хотя сигнатуры могут и не содержаться в явной форме (например, полиморфные вирусы изменяют свой код при каждом заражении). В этом случае применяются методы обнаружения, отличные от сигнатурного (например, эвристический, вероятностный методы).

К категории вредоносных программных средств помимо вирусов относятся троянские кони – программы, которые скрытым образом осуществляют несанкционированные действия, например, чтение конфиденциальных данных. Однако такие программы, несмотря на то, что используют похожие деструктивные функции, не являются вирусами, так как они не могут размножаться. Разновидностью троянских программ являются логические бомбы, которые выполняют разрушающие воздействия при наступлении определенных условий (например, при работе в операционной системе определенной версии, при наступлении заданного времени, при создании определенного файла).

При выполнении зараженной программы управление передается коду вируса. Если вирус является резидентным, то при получении управления он проверяет оперативную память на наличие своей копии и инфицирует память компьютера, если копия не найдена. Если же вирус нерезидентный, то он ищет незараженные файлы в заданных каталогах файловой системы, а затем заражает обнаруженные файлы.

Вирус может выполнять дополнительные деструктивные функции: визуальные или звуковые эффекты, удаление файлов, шифрование разделов жесткого диска и т.д. В последнее время вирусные технологии обычно не выдают своего присутствия в системе эффектами, а их деструктивные функции заключаются в сборе информации, угнетении ресурсов, перенаправлении активности, организация сбоев и пр.

Для поиска заражаемых файлов и выполнения дополнительных функций вирус перехватывает одно или несколько системных функций. При инфицировании файла вирус обычно производит маскирующие действия, например, обработку атрибута "read-only", восстановление исходного размера и даты последней модификации зараженного файла. После этого вирус возвращает управление программе, поэтому внешне функциональность зараженной программы не изменяется.

Наиболее часто заражаемые – файлы формата EXE. EXE-файлы появились еще в системе MSDOS, затем с небольшими изменениями были перенесены в системы Windows. EXE-файлы содержат несколько программных сегментов, включая сегмент кода CS, сегмент данных DS и сегмент стека SS. Файл типа EXE содержит информацию для загрузчика для инициализации программы. Формат EXE-файла для системы Windows называется PE (переносимый исполняемый, Portable Executable), что показывает независимость исполняемого файла от архитектуры процессора. PE-формат включает заголовки, содержащие описание свойств файла и его структуры (рис. 4). Секции исполняемого файла содержат информацию, размещаемую в адресном пространстве процесса при загрузке файла в память. Структура DOS-заголовка (рис. 5):

Sign – подпись файла (4D5Ah, то есть отметка "MZ");

PartPag – длина неполной последней страницы;

PageCnt – длина образа загружаемой программы в 512-байтновых страницах, включая заголовок;

ReloCnt – число элементов в таблице перемещений;

HdrSize – длина заголовка в 16-байтовых параграфах;

MinMem – минимум требуемой памяти за концом программы;

MaxMem – максимум требуемой памяти за концом программы;

ReloSS – сегментное смещение сегмента стека (для установки SS);

ExeSP – значение регистра SP (указателя стека) при запуске;

ChkSum – контрольная сумма;

ExeIP – значение регистра IP (указателя команд) при запуске;

ReloCS – сегментное смещение кодового сегмента;

Tabloff – смещение для первого элемента перемещения;

Overlay – номер оверлея (0 для главного модуля).

DOS-заголовок включен в файл PE-формата для поддержания совместимости со старыми версиями операционной системы.

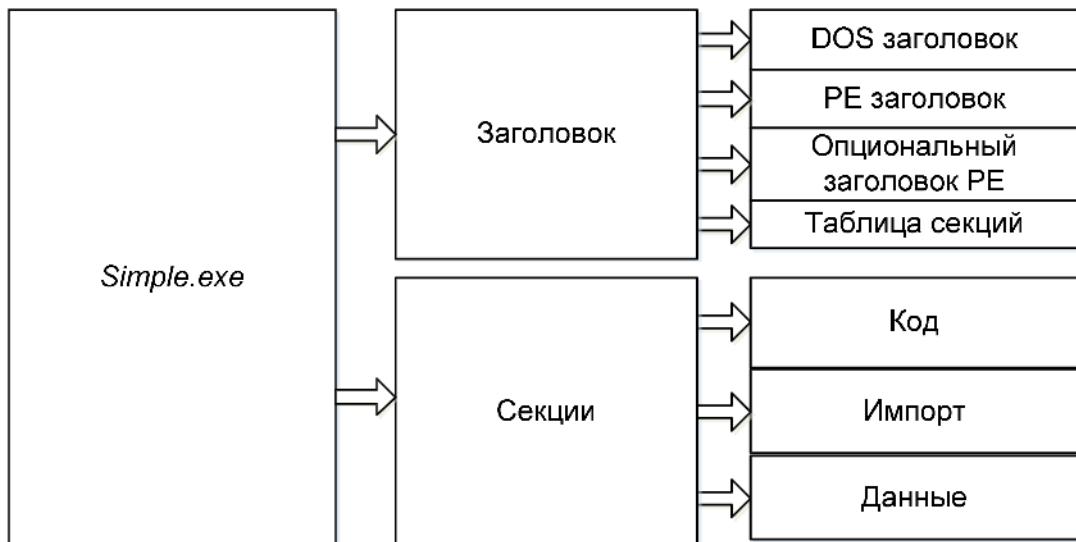


Рис. 4. Структура исполняемого EXE-файла

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00h	Sign	PartPag	PageCnt	ReloCnt	HdrSize	MinMem	MaxMem	ReloSS								
10h	ExeSP	ChkSum	ExeIP	ReloCS	Tabloff	Overlay	Rel. Table E1									
20h	сегм.	смеш.	сегм.	смеш.	
30h	
...	
1F0h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
200h	Начало кода EXE-программы															

Рис. 5. DOS-заголовок EXE-файла

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F								
00h	Signature		Major Link Ver	Minor Link Ver	CodeSize				InitDataSize				UnInitDataSize											
10h	EntryPointAddr				CodeBase				DataBase				ImageBase											
20h	SectionAlign				FileAlign				Major OSVer	Minor OSVer	Major ImageVer	Minor ImageVer	Major	Minor	ImageVer	ImageVer								
30h	Major SubSysVer	Minor SubsysVer	Reserved				ImageSize				HeaderSize													
40h	CheckSum				Subsystem	DLLFlags	StackReserveSize				StackCommitSize													
50h	HeapReserveSize				HeapCommitSize				LoaderFlags				NumberOfRvaAndSizes											
60h	DataDirector																							
...	...																							
140h	DataDirector																							

Рис. 6. PE-заголовок EXE-файла

Опциональный PE-заголовок (рис. 6) используется при загрузке программ в среде Windows. Структура PE-заголовка:

`Signature` – постоянная сигнатура;

`MajorLinkVer` – старшая цифра номера версии сборщика;

`MinorLinkVer` – младшая цифра номера версии сборщика;

`CodeSize` – размер всех секций, содержащих исполняемый код;

`InitDataSize` – сумма размеров всех секций, содержащих инициализированные данные;

`UnInitDataSize` – сумма размеров всех секций, содержащих неинициализированные данные;

`EntryPointAddr` – RVA точки запуска программы (для драйвера – адрес `DriverEntry`, для DLL – адрес `DllMain` или 0);

`CodeBase` – RVA начала кода программы;

`DataBase` – RVA начала данных программы;

`ImageBase` – предпочтительный базовый адрес программы в памяти, кратный 64 Кб (по умолчанию 0x00400000);

`SectionAlign` – выравнивание в байтах для секций при загрузке в память, большее или равное `FileAlign` (по умолчанию равно размеру страницы виртуальной памяти для данного процессора);

`FileAlign` – выравнивание в байтах для секции внутри файла, должно быть степенью числа 2 от 512 до 64Кб включительно (по умолчанию равно 512). Если `SectionAlign` меньше размера страницы виртуальной памяти, то `FileAlign` должно с ним совпадать;

`MajorOSVer` – старшая цифра номера версии операционной системы;

`MinorOSVer` – младшая цифра номера версии операционной системы;

`MajorImageVer` – старшая цифра номера версии данного файла;

`MinorImageVer` – младшая цифра номера версии данного файла;

`MajorSubSysVer` – старшая цифра версии подсистемы;

`MinorSubSysVer` – младшая цифра версии подсистемы;

`Reserved` – зарезервировано (всегда равно нулю);

`ImageSize` – размер файла в памяти, включая все заголовки (кратен `SectionAlign`);

`HeaderSize` – суммарный размер заголовка и заглушки DOS, заголовка PE и заголовков секций, выравненных на границу `FileAlign`, задает смещение от начала файла до данных первой секции;

`CheckSum` – контрольная сумма файла (равен нулю);

`SubSystem` – исполняющая подсистема Windows для данного файла (0 – неизвестная подсистема, 1 – не требует подсистему, 2 – Windows GUI, 3 – консоль);

`DllFlags` – дополнительные атрибуты файла;

`StackReserveSize` – размер стека стартового потока программы в байтах виртуальной памяти, при загрузке в физическую память отображается только `StackCommitSize` байт, в дальнейшем отображается по одной странице виртуальной памяти (по умолчанию равен 1 Мб);

`StackCommitSize` – начальный размер стека программы в байтах (по умолчанию равен 4 Кб);

`HeapReserveSize` – максимальный возможный размер локальной кучи (по умолчанию равен 1 Мб);

`HeapCommitSize` – начальный размер кучи программы в байтах (по умолчанию равен 4 Кб);

`LoaderFlags` – не используется;

`NumberOfRvaAndSizes` – количество описателей каталогов данных (в текущий момент всегда равно 16);

`DataDirSize` – описатели каталогов данных.

В процессе загрузки EXE-файла считывается информация из его заголовков и выполняется настройка адресов сегментов. Затем управление передается загрузочному модулю посредством инструкции дальнего перехода (FAR JMP) по адресу CS:IP, извлеченному из заголовка. После загрузки программы регистры DS и ES указывают на PSP. Регистры CS, IP, SS и SP инициализированы значениями, указанными в заголовке файла.

Обычно файловый вирус дописывается к заражаемому файлу (рис. 7 a). При внедрении вируса в середину файла (рис. 7 b) он может скопировать свой код в таблицу настройки адресов, в область стека (при этом размер зараженного файла не меняется, что затрудняет обнаружение вируса). Вирус может "раздвинуть" файл или переписать часть программы в конец файла, а свой код – на освободившееся место. При внедрении в начало файла (рис. 7 c) вирус либо переписывает начало заражаемого файла в конец, а сам копируется в освободившееся место, либо создает в оперативной памяти свою копию, дописывает к ней заражаемый файл и сбрасывает новый образ файла на диск.

Существуют вирусы, которые записывают себя поверх заражаемого кода (рис. 7 ε). Они не передают управление программе, так как ее код разрушен перезаписью. Программы, зараженные такими вирусами, неизлечимы.

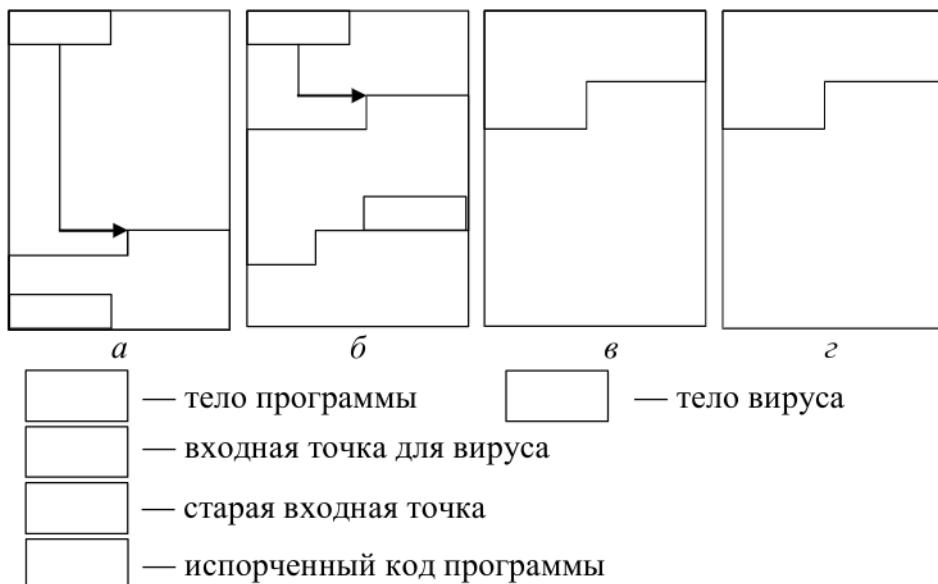


Рис. 7. Способы внедрения вируса в тело программы

EXE-файлы заражаются двумя способами:

1. В заголовке EXE-файла точка входа в программу изменяется таким образом, чтобы она указывала на тело вируса.
2. Точка входа в программу не меняется, но вирус заменяет команды по адресу точки входа таким образом, чтобы они передавали ему управление.

Разновидностью файловых вирусов являются резидентные файловые вирусы, которые, перехватив необходимые прерывания, способны заражать программы не только при их исполнении, но и при открытии файлов. Такие вирусы постоянно находятся в памяти, благодаря чему они могут маскировать изменения при просмотре списка файлов.

Современная ситуация характеризуется распространением полиморфных вирусов с широким применением автоматических генераторов и динамических конструкторов вирусов.

Копии полиморфных вирусов создаются в результате целенаправленной мутации кода и отличаются друг от друга благодаря

шифрованию кода вируса с переменным ключом, что затрудняет сигнатурное обнаружение.

Генератор вирусов – программа, позволяющая задавать характеристики вирусов (например, тип, способ распространения, причиняемый вред) и получать на выходе код нового вируса. Сложные генераторы вирусов (вирусные фабрики) могут быть автоматическими, самораспространяющимися и создавать вирусы в любой системе с учетом ее особенностей, условий функционирования, настроек среды, обстоятельств запуска, состава и параметров работы средств защиты.

Примеры самых простых программ-генераторов компьютерных вирусов: *GVDG* и *Raptor Virus Generator* (*Приложение*).

Генератор *GVDG* создает вирусы-паразиты, перезаписывающие вирусы и вирусы-компаньоны, скрывает сигнатуры путем упаковки вируса с помощью архиватора исполняемого кода *GDS* и генерирует мутирующие вирусы. Генератор также составляет исходные коды вирусов на языке Pascal, что при необходимости позволяет функционально дополнять возможности вируса.

Генератор *Raptor Virus Generator* создает вирусы и троянские программы с возможностью задания иконки результирующего исполняемого файла, что маскирует вирус под популярные форматы данных и программ (например, графический файл, инсталлятор).

Для борьбы с компьютерными вирусами применяются средства антивирусной защиты:

программы-детекторы – антивирусные средства, определяющие наличие вирусов, но не удаляющие их. Они сканируют память, файлы, загрузочные секторы дисков с целью обнаружения вирусов. Детекторы реализуют поиск сигнатур в коде программы или проверку фрагментов кода программы, получающих управление, на совпадение с известными командами вируса;

программы-фаги – антивирусные средства, способные обнаруживать и восстанавливать (если это возможно) пораженные программы, удаляя из них тело вируса;

ревизоры – антивирусные средства, реализующие сопоставление текущей и эталонной информации о файлах, загрузочных секторах и

других параметрах системы. Сопоставление позволяет выявлять изменения, производимые вирусами, и зараженные компоненты;

интеллектуальные средства антивирусной защиты, использующие различные эвристические и самообучающиеся алгоритмы, автоматически расширяющие список обнаруживаемых вирусов.

Сегодня наиболее распространены антивирусы-фаги (например, *ESET NOD32*, *Dr.Web*, *Kaspersky*), комбинирующие и сигнатурный поиск, и анализ программ, что позволяет им выявлять новые вирусы.

Порядок выполнения работы

Во избежание потери информации и нарушения работоспособности используемых систем все действия следует выполнять в виртуальной среде.

Краткое описание пользовательских интерфейсов генераторов вирусов *GVDG* и *Raptor Virus Generator* приведено в *Приложении*.

1. Подготовка к работе.

1.1. Выбрать локальный диск и создать в нем каталоги *virus_test* и *virus_disk_test*.

1.2. Для тестирования генератора вирусов *Raptor Virus Generator*, смонтировать каталог *virus_disk_test* в корневой уровень файловой системы, например, с помощью команды *SUBST* консоли ОС Windows:

```
SUBST X: c:\virus_disk_test.
```

2. Исследование механизма заражения.

2.1. Запустить генератор вирусов *GVDG*.

2.2. Используя управляющие элементы интерфейса генератора вирусов *GVDG*, создать вирус-паразит с несколькими деструктивными функциями.

2.3. Изучить полученный исходный код на языке Pascal (файл с расширением PAS) и построить блок-схему алгоритма работы вируса.

2.4. Скопировать полученный EXE-файл вируса в каталог *virus_test*, скопировать туда же несколько произвольных EXE-файлов для демонстрации заражения, предварительно сохранив их эталонные копии.

2.5. Запустить скопированный компьютерный вирус.

2.6. На основании сравнения по содержимому (например, с помощью встроенных средств сравнения файлов файл-менеджера *Total*

Commander) демонстрационных EXE-файлов и их сохраненных эталонных копий сформулировать гипотезу о способе внедрения вируса в тело файла.

2.7. Скопировать один из ранее сохраненных эталонных EXE-файлов в каталог *virus_test*, запустить на исполнение один из уже зараженных EXE-файлов. Убедиться в заражении нового EXE-файла в результате запуска зараженного EXE-файла.

2.8. С помощью антивируса, встроенного в генератор *GVDG*, попытаться вылечить исполняемые файлы в каталоге *virus_test*.

2.9. Повторить пп. 2.5 и 2.6, после чего запустить проверку любым сторонним антивирусом (например, *ESET NOD32*, *Dr.Web*, *Kaspersky*). При обнаружении вирусов включить опцию шифрования в настройках генератора и повторить процедуру снова. Сравнить полученные результаты.

3. Работа с генератором вирусов *Raptor Virus Generator*.

3.1. Скопировать в каталог *virus_disk_test* несколько произвольных EXE-файлов для демонстрации заражения, предварительно сохранив их эталонные копии.

3.2. С помощью программы-генератора *Raptor Virus Generator* сгенерировать вирус, обладающий тремя деструктивными функциями и выполняющим заражение всех файлов на диске (в качестве диска-жертвы указать диск, созданный при выполнении команды *SUBST*).

3.3. Запустить утилиту *ProcMon*, задать в ней фильтр по имени процесса вируса. В данном случае он должен совпадать с именем созданного исполняемого файла вируса. Оставив активной программу *ProcMon*, запустить исполняемый файл вируса. Зафиксировать протокол работы вируса, полученный программой *ProcMon*.

3.4. Очистить папку *virus_disk_test* и повторить п. 3.1.

3.5. Запустить утилиту *apiMonitor*, задать фильтр по имени процесса вируса. Иницировать слежение с помощью утилиты *apiMonitor* (сочетание клавиш *CTRL+E*) и запустить созданный файл вируса в папке *virus_disk_test*. Зафиксировать протокол работы вируса, полученный программой *apiMonitor*.

3.6. На основании полученных протоколов работы вируса (пп. 3.3 и 3.5) построить функциональную схему сгенерированного вируса, выделив деструктивные функции.

- 3.7. Очистить папку *virus_disk_test* и повторить п. 3.1.
- 3.8. Выполнить заражение исполняемых файлов с помощью созданного вируса. Запустить программу-дизассемблер (например, *W32Dasm*, *ollyDBG*). Дизассемблировать код зараженного EXE-файла. Используя полученный ассемблерный код и построенную функциональную схему вируса, выделить участки программы, которые содержат тело вируса, проверяют сигнатуру вируса, осуществляют заражение и деструктивные функции, передают управление программе-носителю.

Содержание отчета

1. Опции генераторов вирусов и характеристики созданных вирусов.
2. Характеристики тестовых исполняемых файлов до и после заражения, а также до заражения и после лечения антивирусом: размер файла, дата его модификации, изменение содержимого каждого файла.
3. Вывод о способе заражения файла, полученный с помощью визуального сравнения эталонного и зараженного файла.
4. Исходные коды сгенерированных вирусов.
5. Блок-схема алгоритма заражения и функциональная схема вируса.
6. Дизассемблированный код зараженной программы, в котором выделено тело вируса и его функции: проверка сигнатуры вируса, поиск и заражение исполняемых файлов, деструктивные функции, возврат управления программе-носителю (уровень детализации – файловые, математические, текстовые операции).
7. Ответы на контрольные вопросы.
8. Выводы по работе.

Контрольные вопросы

1. Перечислите виды компьютерных вирусов.
2. Для чего используется DOS-заголовок исполняемого файла в операционной системе Windows?
3. Какие действия выполняет вирус при заражении файла?
4. Укажите методы обнаружения вирусов, отличные от сигнатурного.
5. Как при отсутствии программ-детекторов и фагов можно вылечить зараженную программу?

Лабораторная работа 6
**ЗАЩИТА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ
ОТ НЕЛЕГАЛЬНОГО ИСПОЛЬЗОВАНИЯ**

Цель работы – приобретение навыков по защите приложений от нелегального использования, по анализу исполняемых кодов в отсутствие исходных текстов и по применению способов защиты программ от дизассемблирования и отладки.

Теоретические сведения

Простейшим способом проверки легальности копии программы является запрос у пользователя учетных данных (например, пароля, номера лицензии, идентификатора, ключа), выданных ему при покупке. Если пользователем введены корректные данные, функция проверки разрешает продолжить работу с программой.

В общем виде такая схема выглядит следующим образом:

```
if( !Password() ) /* Ввод и проверка пароля */
{
    pError("Access denied!");
    abort();
}
```

Данный код транслируется в набор ассемблерных инструкций:

```
CALL Password
OR AX,AX
JZ continue
PUSH offset str_access_denied
CALL pError
.
.
.
continue:
.
.
```

Программа продолжает полноценно работать после метки **continue**. Такой защитный механизм может быть легко обойден заменой одного байта. Заменив условный переход на безусловный, то есть

инструкцию `JZ` на `JMP`, злоумышленник получает полностью работоспособную копию программы независимо от сложности алгоритма, реализованного в функции *Password*.

Вторым способом защиты программ от взлома является запрет на открытое хранение пароля во внешнем файле. Таким образом, например, реализовано хранение паролей в операционных системах семейства UNIX. Доступ к файлу паролей не дает никакой информации о самих паролях. Пароли хранятся в зашифрованном виде или в виде соответствующих хэш-значений. Хэш – код, вычисленный с помощью необратимой математической функции. Знание алгоритма шифрования или хэширования не дает предположений об исходном пароле.

Простейшая схема скрытия кода программ в целях затруднения ее анализа и модификации – использование логического "исключающего ИЛИ" (XOR). Таблица истинности функции XOR имеет вид:

$$\begin{aligned} 0 \text{ XOR } 0 &= 1 \text{ XOR } 1 = 0, \\ 0 \text{ XOR } 1 &= 1 \text{ XOR } 0 = 1. \end{aligned}$$

Основное свойство функции: $a \text{ XOR } b \text{ XOR } a = b$, поэтому шифратор и дешифратор кода имеют одно строение.

На языке программирования чрезвычайно трудно создать что-нибудь устойчивое ко взлому, поэтому в большинстве коммерческих приложений применяется механизм "психологического давления", направленного на систематическое прерывание работы пользователя, не приобретающего лицензию на авторский продукт.

Пример такой защиты – реализация всплывающих окон (*nag screen*), которые часто активизируются в ходе сеанса пользовательской работы и напоминают о необходимости зарегистрировать программу. Окно может содержать поле ввода регистрационных данных, рекламу или просьбу нажать некоторую клавишу. Слово "nag" переводится как "назойливый", что отражает характер воздействия данного механизма: постоянные прерывания и напоминания, препятствующие работе, подталкивающие к приобретению легальной копии программы, в которой данный механизм деактивирован.

Для обхода механизмов защиты и для анализа поведения программ с целью выявления нарушений применяются программы-дизассемблеры и отладчики (например, *IDA*, *W32Dasm*, *ollyDBG*, *SoftICE*, *Periscope*, *Bubble Chamber*, *Sourcer*). В простейшем случае, как и при проверке пароля, нарушителю необходимо исправить один байт, что свидетельствует о чрезвычайной ненадежности такого способа защиты программ.

Для того чтобы противодействовать модификации исполняемого файла, разработчики используют проверку целостности программ (например, с помощью контрольного суммирования). Кроме того, существует множество методов по усложнению дизассемблирования и анализа алгоритмов программ.

Простейшим методом защиты от дизассемблирования является применение оптимизирующего компилятора. В оптимизированном исполняемом файле с целью повышения быстродействия программы или сокращения объема исполняемого файла нарушается прямой ("логичный") порядок вызова команд и использования регистров. Это несколько усложняет процесс анализа функций и передачи управления внутри кода программы. В современных компиляторах оптимизация исполняемого кода включена по умолчанию.

Традиционный способ защиты программ – применение шифрования или упаковки распространяемого программного обеспечения. В этом случае собственно исполняемый код не соответствует тексту программы в открытом виде. Однако если программа во время исполнения сама себя дешифрует или распаковывает, то она же содержит код, выполняющий действия по дешифрации или распаковке. Причем этот код доступен злоумышленнику, и, следовательно, выполнив его, можно получить в памяти исходный исполняемый код. Поэтому в программах, требующих хорошего уровня защиты от дизассемблирования, используется метод неполного динамического шифрования/дешифрования, таким образом, программа никогда не содержится в памяти, расшифрованной целиком.

В большинстве случаев получение исходного ассемблерного кода не дает достаточно информации о механизме защиты. Тогда с помощью специальных утилит-отладчиков можно запустить анализируемую исполняемую программу в режиме отладки. Простейшим методом защиты

от запуска в режиме отладки является проверка наличия отладчика (например, функция *isDebugPresent*).

Существует метод защиты программного обеспечения, основанный на запутывании программного кода, – обфускация (*obfuscate* – сбивать с толку). При этом исходный текст или исполняемый код программы приводится к виду, сохраняющему ее функциональность, но затрудняющему анализ, понимание и модификацию алгоритма при дизассемблировании. Для создания обфусцированного кода могут применяться специализированные компиляторы, использующие неочевидные и недокументированные возможности, или специальные программы-обфускаторы.

При программировании на интерпретируемых скрипт-языках (например, JavaScript, VBScript) пользователю доступен исходный текст программы. Обфускация в таком случае сводится к форматированию текста и к замене имен переменных и функций, что направлено на то, чтобы сделать текст менее читаемым, и, соответственно, менее понятным нарушителям. Например, код, созданный на JavaScript:

```
function MyClass(){
    this.foo = function(argument1, argument2){
        var addedArgs =
            parseInt(argument1)+parseInt(argument2);
        return addedArgs;
    }
    var anonymousInnerFunction = function(){}
}
```

может быть обфусцирован в функционально эквивалентную, но неочевидную последовательность команд:

```
eval(function(p,a,c,k,e,d){e=function(c){return
c};if(!''.replace(/\^/,String)){while(c--
){d[c]=k[c]||c}k=[function(e){return
d[e]}];e=function(){return'\\w+'};c=1};while(c--
){if(k[c]){p=p.replace(new RegExp('\\\b'+e(c)+'\\\b','g'),
k[c])}}return p}('4 0="3 5!";9 2(1){6(1+"\\\7"+0)}2("8");
',10,10,'a|msg|MsgBox|Hello|var|World|alert|n|OK|
function'.split('|'),0,{}))
```

Обфускация на уровне машинного кода заключается в добавлении блоков с необязательным исполнением, в перемешивании и внесении "мусорных" функциональных блоков программы с целью осложнения анализа кода, но не нарушения логики работы программы. Код независимых функциональных блоков вместо тиражирования может, наоборот, объединяться. Таким образом, граф исполнения программы принимает новый вид, в котором передача управления выполняется через новую цепочку блоков и совсем не очевидным образом. В этой связи негативное следствие обфускации – замедление работы программы.

При обфускации машинного кода также используется скрытие констант и данных, когда константа формируется во время исполнения и не встречается в открытом виде. Константные данные могут скрываться при помощи шифрования.

В настоящее время ведутся разработки обфускаторов, использующих виртуальные процессоры. В этом случае создается случайная виртуальная машина со случайным набором инструкций, и весь код программы трансформируется под нее.

Обфускация помогает сделать программу более защищенной от отладки и анализа, но ни один из существующих обфускаторов не гарантирует абсолютной невозможности восстановления логики работы программы.

Порядок выполнения работы

В ходе выполнения работы студенту предоставляется два исполняемых файла *Nag.exe* и *Guard.exe*, моделирующих различные методы защиты легального программного обеспечения. Файл *Nag.exe* является примером применения механизма всплывающих окон, файл *Guard.exe* – защиты от дизассемблирования и отладки. Исполняемый файл *Guard.exe* упакован и использует функции проверки наличия отладчика.

1. Создать простую программу на языке С, запрашивающую пароль. Пароль должен храниться в открытом виде в отдельном файле. Проверку пароля реализовать в виде отдельной функции.

2. Скомпилировать программу. Сохранить эталонную копию полученного исполняемого файла.

3. Убедиться в том, что программа запрещает доступ пользователям, неправильно вводящим пароль.

4. Пользуясь исходным текстом программы и утилитой *NIEW*, отыскать в исполняемом файле команду перехода при проверке правильности пароля (вариант ассемблерной инструкции *JMP*) и изменить ее таким образом (например, заменой на инструкцию *NOP*), чтобы при вводе любого пароля программа принимала пользователя как авторизованного. При этом следует иметь в виду, что команды ближнего перехода используют относительное смещение. Поэтому при их замене необходимо следить за постоянностью размера команды перехода до и после изменения, так как изменение размера команды может привести к непредсказуемым действиям программы.

5. Проверить работу исполняемого файла при вводе любых данных в качестве пароля.

6. Подготовить компилятор к новому режиму генерации исполняемого кода, включив в его настройках опцию оптимизации кода компилируемой программы, если она выключена, или – наоборот – выключив ее, если она включена.

7. Скомпилировать программу с новыми настройками компилятора. Повторить выполнение пп. 3 и 4. Зафиксировать различия полученной программы и сохраненной в п. 2 ее эталонной копии, а также возникшие трудности при поиске необходимых инструкций ассемблера.

8. Модифицировать свою программу таким образом, чтобы исключить открытое хранение пароля. Для этого следует вставить в свой исходный код функцию, отвечающую за простейшее шифрование пароля: *пароль XOR константа*. Повторить выполнение пп. 3 и 4.

9. Получить у преподавателя тестовую программу *Nag.exe*, работающую в среде *Win32*.

10. Запустить на выполнение программу *Nag.exe*. Начать вводить в окне редактирования произвольный текст (в течение 20...25 с.). Дождаться появления и завершения работы всплывающего окна, нажать на кнопку "OK" во всплывающем окне, продолжить редактирование вводимого текста. Убедиться в многократности появления такого счетчика и определить период его появления. Зафиксировать наблюдаемые эффекты.

11. Создать резервную копию тестовой программы *Nag.exe* для ее восстановления в случае необратимой модификации исполняемого кода.

12. С помощью программы-дизассемблера *W32Dasm* (или любой другой) и без помощи исходных текстов программы найти в исполняемом файле команды, отвечающие за увеличение счетчика, выводимого на кнопке всплывающего окна. Зафиксировать найденный фрагмент кода.

13. Модифицировать найденный код таким образом, чтобы надпись "OK" появлялась сразу же после активизации всплывающего окна. Проверить работоспособность программы.

14. Создать копию программы *Nag.exe* из сохраненной резервной копии.

15. В полученной копии при помощи дизассемблера найти исполняемый код, отвечающий за активизацию всплывающего окна. Для этого найти функцию активации таймера (*SetTimer*) и изменить код таким образом, чтобы окно счетчика больше не блокировало работу программы.

16. Проверить работоспособность программы. Зафиксировать модифицированные участки исполняемого кода.

17. Получить у преподавателя демонстрационную программу *Guard.exe*, работающую в среде *Win32*.

18. Запустить программу *Guard.exe* на выполнение и попытаться ввести произвольный пароль.

19. Проверить работу программы при вводе произвольных данных.

20. Выявить, с помощью какой утилиты запакован исполняемый файл (упаковщики *AS-PACK*, *FSG*, *GHF*, *MEW*, *NSPACK*, *PeCompact*, *UPX*, *WinUPack*, *Xitech* и т.д.). Распаковать файл с помощью специализированной утилиты. Для определения упаковщика используются специальные утилиты, например, *Reid*. Зафиксировать распакованный код исполняемого файла.

21. С учетом того, что весь исполняемый код полностью распаковывается в момент запуска и код распаковщика доступен, открыть исполняемый файл отладчиком *ollyDBG* (или любым другим), зафиксировать весь доступный исполняемый код и отследить окончание его выполнения. Зафиксировать распакованный исполняемый код, сравнить его с распакованным кодом, полученным в п. 20.

22. Запустив распакованный исполняемый файл, полученный в п. 20, на исполнение в отладчике *ollyDBG* (или в другом), найти команды, реализующие проверку использования отладчика и вывод сообщения.

23. Модифицировать найденный код программы таким образом, чтобы проверка использования отладчика не выполнялась.

24. В коде программы найти команды, отвечающие за проверку пароля, и модифицировать их так, чтобы выполнялся обход проверки.

25. Для разработанной в п. 8 программы реализовать обфускатор на уровне исходных кодов. Созданный обфускатор должен удалять излишние пробельные символы (пробелы, табуляцию и переносы строки), удалять комментарии, заменять имена переменных и функций на произвольные нумерованные и/или строковые переменные, перемешивать функции, вносить "мусорные" переменные, циклы и функции, сохраняя первоначальную логику работы программы. Параметры обфускации должны задаваться в отдельном конфигурационном файле обфускатора.

Содержание отчета

1. Листинги реализованных программ до и после модификации.
2. Пароль и примеры работы созданной программы.
3. Найденные участки исполняемого кода, содержащие команды переходов, в дизассемблированном виде с указанием адресов команд.
4. Все внесенные в ходе работы изменения и примеры работы тестовых программ с произвольным вводом пароля.
5. Выводы по сравнению исходной и оптимизированной программ.
6. Для программы *Nag.exe*: примеры работы программы; участки кода программы, содержащие вызов *SetTimer*; модифицированные участки кода; пример работы программы, не блокируемой счетчиком.
7. Для программы *Guard.exe*: примеры работы программы; распакованный исполняемый код; участки кода программы, содержащие команды перехода на вывод сообщений об ошибке ввода данных и о наличии отладчика; модифицированные участки кода; пример работы программы с произвольным вводом пароля и без проверки использования отладчика.
8. Для разработанного обфускатора исходных текстов: исходные тексты обфускатора; конфигурационные параметры обфускатора с

примерами программного кода до и после обfuscации; блок-схема алгоритма работы обfuscируемой программы до и после обfuscации.

9. Ответы на контрольные вопросы.

10. Выводы по работе.

Контрольные вопросы

1. Какие методы применяются для защиты коммерческих программ от ввода некорректных учетных данных?
2. По каким признакам можно найти интересующий код сопоставления с паролем в программе?
3. Какие существуют основные методы защиты от дизассемблирования и отладки программ?
4. Как реализуется обfuscация с помощью виртуальных машин?
5. Предложите методы усиления механизмов защиты программ от нелегального использования, исследованные в данной работе.

Лабораторная работа 7

КОДИРОВАНИЕ И УПАКОВКА ДАННЫХ

Цель работы – приобретение навыков по защите информации с помощью методов кодирования, получение прикладных знаний в области исследования и реализации алгоритмов упаковки данных.

Теоретические сведения

Теория кодирования

Задача кодирования данных формально представляется следующим образом. Заданы алфавиты $A=\{a_1, \dots, a_n\}$ и $B=\{b_1, \dots, b_m\}$, а также функция $F: S \rightarrow B^*$, где S – подмножество в алфавите A , $S \subseteq A^*$, A^* – множество всех слов в алфавите A , B^* – множество всех слов в алфавите B .

Функция F называется *кодированием*, элементы множества S – сообщениями, элементы $\beta=F(\alpha)$, $\alpha \in S$, $\beta \in B^*$ – кодами соответствующих сообщений. Обратная функция F^{-1} (если она существует) называется

декодированием. Если $|B| = m$, то F – m -ичное кодирование, например, при $B = \{0, 1\}$ F – двоичное кодирование.

Типичная задача кодирования формулируется следующим образом: при заданных алфавитах A, B и множестве сообщений S найти кодирование F , которое обладает определенными свойствами, то есть удовлетворяет заданным ограничениям, и оптимально в некотором смысле. Критерий оптимальности, как правило, связан с минимизацией длин получаемых кодов. Таким образом, *кодирование* – преобразование информации, в том числе с защитной целью, а *упаковка* (сжатие, архивирование) является побочным эффектом кодирования.

Примеры задаваемых ограничений:

существование декодирования. Это естественное, но не всегда необходимое свойство (например, трансляция программы с языка высокого уровня в машинные команды – это кодирование, которое не требует однозначного декодирования);

помехоустойчивость – функция декодирования F^{-1} обладает таким свойством, что $F^{-1}(\beta)=F^{-1}(\beta')$, если β' в определенном смысле близко к β ;

заданная сложность кодирования и декодирования.

Кодирование F может сопоставлять код всему сообщению из множества S как единому целому или же строить код сообщения из кодов его частей. Пусть задано конечное множество $A=\{a_1, \dots, a_n\}$, которое называется алфавитом. Элементы алфавита называются буквами. Элементарной частью сообщения является буква алфавита A . Последовательность букв – слово. Множество слов в алфавите A обозначается A^* . Если слово $\alpha=a_1\dots a_k \in A^*$, то количество букв в слове называется длиной слова: $|\alpha|=|a_1\dots a_k|=k$. Пустое слово обозначается Λ : $\Lambda \in A^*, |\Lambda|=0, \Lambda \notin A$.

Если $\alpha=\alpha_1\alpha_2$, то α_1 называется началом (префиксом) слова α , а α_2 – окончанием (постфиксом) слова α . Если при этом $\alpha_1 \neq \Lambda$ (соответственно, $\alpha_2 \neq \Lambda$), то α_1 (соответственно, α_2) называется собственным началом (соответственно, собственным окончанием) слова α .

Алфавитное (или побуквенное) кодирование задается схемой (или таблицей кодов) $\sigma=(\alpha_1 \rightarrow \beta_1, \dots, \alpha_n \rightarrow \beta_n)$, $\alpha_i \in A$, $\beta_i \in B^*$. Множество букв $V=\{\beta_i\}$ называется множеством элементарных кодов.

Для практики важно, чтобы коды сообщений имели по возможности наименьшую длину. Алфавитное кодирование пригодно для любых сообщений, то есть $S=A^*$. Если больше про множество S ничего не известно, то точно сформулировать задачу оптимизации затруднительно. Однако на практике часто доступна дополнительная информация. Например, для текстов на естественных языках известно распределение вероятности появления букв в сообщении. Использование такой информации позволяет строго поставить и решить задачу построения оптимального алфавитного кодирования.

Таким образом, логично составить такую схему кодирования, при которой наиболее часто встречающейся в сообщении букве будет соответствовать самый короткий элементарный код, а наиболее редкой букве – самый длинный.

Пусть заданы алфавит $A=\{a_1, \dots, a_n\}$ и соответствующие вероятности появления букв в сообщении $P=\langle p_1, \dots, p_n \rangle$. Ценой кодирования называется математическое ожидание коэффициента увеличения длины сообщения при кодировании, которое составляет $I_o(P)=\sum_{i=1}^n p_i l_i$, где $l_i=|\beta_i|$ – длина кодовой последовательности β_i .

Обозначим $L=[\log_2(n-1)]+1$. Тогда существует такая схема кодирования, что $\forall i |\beta_i|=L$, которая называется равномерной.

Алфавитное кодирование, для которого $I_o(P)$ является минимальным при данном распределении вероятностей P , называется кодированием с минимальной избыточностью, или оптимальным кодированием.

Например, пусть $A=\{a,b\}$ и $\sigma=\langle a \rightarrow 0, b \rightarrow 01 \rangle$. $S_1="aababbbaba"$. $S_2="aaaabaaaaa"$. Для S_1 распределение вероятностей получается $\langle 0,5; 0,5 \rangle$, цена кодирования $0,5 \times 1 + 0,5 \times 2 = 1,5$ (закодированное сообщение S_1 : 000100101010010 – 15 знаков), а для S_2 распределение вероятностей – $\langle 0,9; 0,1 \rangle$ и цена кодирования $0,9 \times 1 + 0,1 \times 2 = 1,1$ (закодированное сообщение S_2 : 00000100000 – 11 знаков).

Сжатие (упаковка) данных – кодирование, позволяющее построить без потери данных код сообщения меньшей длины по сравнению с исходным. Качество сжатия определяется коэффициентом сжатия, показывающим, насколько закодированное сообщение короче исходного.

Различают статические методы сжатия и адаптивные. Статические методы создают отображение из множества букв алфавита во множество кодовых слов до начала процесса кодирования, а при сжатии каждая буква алфавита заменяется соответствующим кодовым словом. Для построения такого отображения до начала сжатия необходим предварительный проход по сообщению (файлу), чтобы собрать необходимую информацию о сообщении. При этом в выходной поток должны быть записаны данные о собранной статистике, чтобы декодер смог расшифровать сжатые данные.

Адаптивные методы не нуждаются в предварительном просмотре сообщения, так как они динамически меняют схему кодирования в зависимости от исходных данных. Такие алгоритмы не требуют передачи в выходной поток информации об использованном кодировании. Вместо этого декодер, считывая кодированный поток, изменяет схему кодирования, начиная с некоторой предопределенной. Адаптивное кодирование может дать большую степень сжатия, поскольку могут быть учтены локальные изменения частот.

Алгоритм Фано

Кодирование Фано выполняется следующим образом. N букв входного алфавита располагаются в порядке убывания их вероятностей. Далее последовательность букв разбивается на две группы так, чтобы суммарные вероятности букв в каждой из групп были как можно более близки друг другу. Буквам из одной группы в качестве первого символа кодового слова приписывается символ 0, буквам из второй группы – 1. По тому же принципу каждая из групп снова разбивается на две подгруппы, и это разбиение определяет значение второго символа кодового слова.

Процедура продолжается до тех пор, пока все множество групп не будет разбито на отдельные буквы. В результате каждой из букв будет сопоставлено кодовое слово из нулей и единиц. Чем более вероятно появление буквы во входном потоке, тем быстрее она образует самостоятельную группу и тем более коротким словом она будет закодирована. Это обстоятельство обеспечивает высокую экономность кода Фано. Данный алгоритм позволяет построить близкую к оптимальной схему кодирования.

Например, необходимо закодировать сообщение "aabcbada". Всего задано четыре буквы: a , b , c , d с вероятностями $P(a)=1/2$, $P(b)=1/4$, $P(c)=P(d)=1/8$. Множество букв разбиваем на две группы: в первой группе один элемент, а во второй – три элемента. Далее еще раз разобьем множество букв b , c , d на две равновероятные подгруппы. Первой, состоящей из одной буквы b , сопоставим 0, а второй, в которую входят c и d – 1. Наконец оставшуюся группу из двух букв снова разобьем на две подгруппы, содержащие соответственно c и d , сопоставив первой из них 0, а второй – 1.

Для буквы a сразу образована группа и ей сопоставлен код 0. Буква b образовала группу за два шага (на первом шаге ей сопоставлен символ 1, на втором – 0), поэтому ее код – 10. Коды c и d – 110 и 111.

В результате код сообщения – 00101101001110. Длина кода – 14 бит вместо исходных 64 бит. Вместе с закодированным сообщением передается схема кодирования для восстановления исходных данных.

Алгоритм Хаффмена

Алгоритм Хаффмена позволяет построить оптимальную схему кодирования. Алгоритм основан на свойствах оптимального набора:

чем больше вероятность появления буквы, тем короче длина кода;
две самые длинные кодовые комбинации имеют одинаковую длину;
если в наборе из n символов с вероятностями $p_1 \geq \dots \geq p_{n-1} \geq p_n$ объединить последние два символа, то получится набор из $n-1$ символа, одному из которых соответствует вероятность $p' = p_{n-1} + p_n$. Тогда оптимальный кодовый набор для n символов получается из оптимального кодового набора для $n-1$ символа удлинением на 1 бит кодовой последовательности для объединенного символа (он разбивается на два исходных, которым соответствует кодовая последовательность этого символа с добавлением 0 для первого и 1 для второго символа). Для набора из двух символов коды известны: 0 и 1.

Допустим, алфавит состоит из пяти символов: a , b , c , d , e , для которых известны соответствующие вероятности появления в тексте сообщения: 0,30, 0,22, 0,16, 0,14 и 0,11. Объединяя в один символ буквы d и e , получаем отсортированный набор вероятностей 0,37(a), 0,25(de), 0,22(b), 0,16(c). На следующем шаге получим 0,38(bc), 0,37(a), 0,25(de), а

затем $0,62(ade)$, $0,38(bc)$. Сопоставим групповому символу ade код 0, а bc – код 1 и выполним "обратный ход". Расщепим ade на a и de с кодами 00 и 01, а bc – на b и c с кодами 10 и 11. Наконец, расщепив de , получим коды $00(a)$, $10(b)$, $11(c)$, $010(d)$, $011(e)$.

Арифметическое кодирование

Алгоритм арифметического кодирования основан на соответствии букв алфавита вероятностным интервалам в промежутке $[0; 1)$, длины которых равны вероятностям букв.

Кодирование реализуется путем сужения исходного интервала в зависимости от приходящей буквы. При получении первой буквы сообщения кодер определяет, какой ей соответствует интервал, и сужает исходный интервал $[0; 1)$ до него. При получении следующей буквы кодер определяет ее интервал и ставит этому интервалу в соответствие новый, внутри суженного на предыдущем шаге. Новый интервал в пропорции к суженному будет таким же, как интервал буквы в пропорции к исходному интервалу $[0; 1)$.

При получении следующих символов интервал снова сужается. После сужения его в последний раз любое число, лежащее внутри полученного интервала, будет являться кодом сообщения. Естественно, что при увеличении длины сообщения увеличивается и точность вычислений, и, следовательно, количество бит, необходимое для кодировки сообщения.

При декодировании декодер определяет, в каком из интервалов, соответствующих буквам алфавита, лежит полученное при кодировании число (для этого декодеру должна быть передана информация о заданных интервалах). Первая раскодированная буква – буква, соответствующая полученному интервалу. Далее интервал $[0; 1)$ сужается до определенного на предыдущем шаге интервала. При получении следующего символа декодер определяет, в каком из пропорциональных исходным интервалам, соответствующим буквам алфавита, лежит кодовое число, и получает таким образом следующую букву.

При этом необходимо определить, когда следует закончить декодирование. Это можно реализовать двумя способами: передать кодеру длину закодированного сообщения или добавить в алфавит некую

завершающую букву, которая будет добавляться кодировщиком в конец сообщения (ей также будет соответствовать определенный интервал), а при декодировании распознаваться как терминальный знак. Эта буква определяется как служебная и только для кодера и декодера, и она не должна встречаться в кодируемых сообщениях.

Например, требуется закодировать сообщение "bab". Алфавит состоит из двух букв *a* и *b*. Добавим завершающую букву #, увеличив тем самым алфавит до трех букв, а сообщение – до пяти. Вероятности букв равны 0,2, 0,6 и 0,2. Интервал для *a* – [0; 0,2), *b* – [0,2; 0,8), # – [0,8; 1).

При получении первой буквы *b* исходный интервал [0; 1) сужается до $[0+0,2 \times 1; 0+0,8 \times 1] = [0,2; 0,8)$. Длина текущего интервала: $0,8 - 0,2 = 0,6$. Далее, получив *a*, кодер сужает интервал до $[0,2+0 \times 0,6; 0,2+0,2 \times 0,6) = [0,20; 0,32)$.

При получении буквы границы и размер интервала вычисляются по формулам:

$$\begin{aligned} newleft &= prevleft + currleft \times prevsize, \\ newsize &= currsize \times prevsize, \\ newright &= newleft + newsize, \end{aligned}$$

где *prevleft* и *prevsize* – левая граница и длина интервала, полученного на предыдущем шаге; *currleft* и *currsize* – левая граница и длина интервала, соответствующего кодируемой букве из переданного кодировщиком списка.

В результате кодирования получился интервал [0,27296; 0,28160) (рис. 8). Выбираем любое число из этого интервала. Для представления его в виде наиболее короткой последовательности бит надо, чтобы это число являлось произведением целого числа на отрицательную степень числа 2. Причем, чем меньше модуль степени этого числа, тем короче его код. В этот интервал попадает число $9/2^5 = 0,28125$. Ему соответствует двоичный код 0,01001 (то есть сдвигаем число $9_{(10)} = 1001_{(2)}$ вправо на 5 разрядов). Отбрасываем начальные нули, так как число всегда меньше 1 и больше 0. В результате получаем код $01001_{(2)}$ длиной 5 бит.

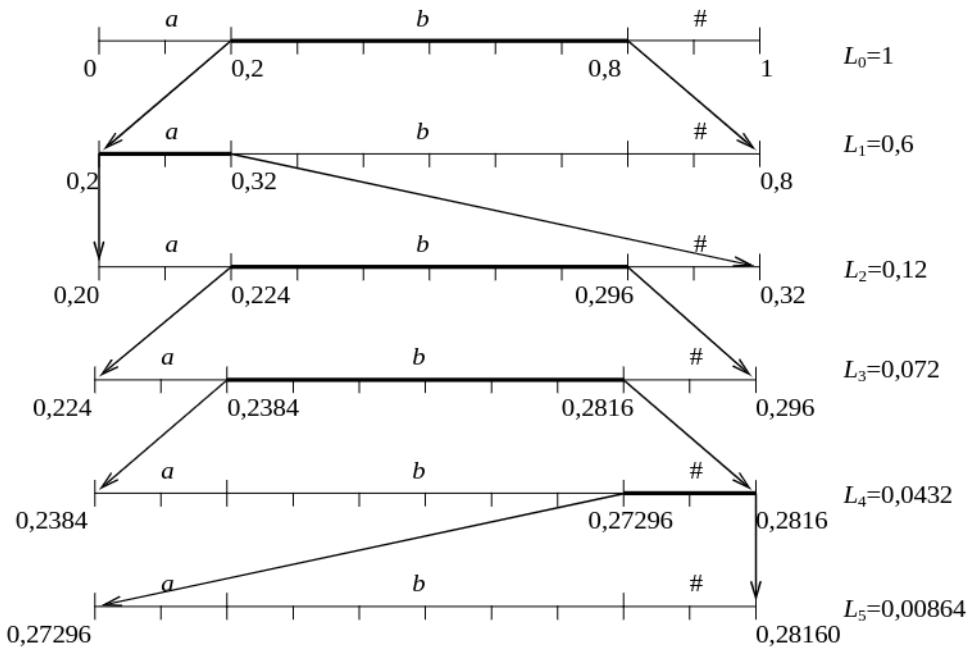


Рис. 8. Пример построения арифметических кодов

Для декодирования данного кода приписываем к нему "0," и проверяем, в каком интервале из переданного кодером списка лежит данное число: $0,2 \leq 0,28125 < 0,8$. Полученный интервал соответствует букве *b*, а значит, она является первой буквой сообщения.

Далее интервал сужается до $[0,2, 0,8)$. Проверяется, в каком из пропорциональных исходным интервалов (на втором шаге исходным интервалам $[0; 0,2)$, $[0,2; 0,8)$, $[0,8; 1)$ из $[0; 1)$ соответствуют интервалы $[0,20; 0,32)$, $[0,32; 0,68)$, $[0,68; 0,8)$ из $[0,2; 0,8)$) лежит данное число: $0,2 \leq 0,28125 < 0,32$. Число принадлежит первому интервалу, что соответствует букве *a* – второй букве сообщения.

Далее интервал сужается до $[0,2; 0,32)$. Определяется, в какой его части лежит кодовое число. Сужения интервалов проводятся точно такие же, как и при кодировании, причем, в той же последовательности. Декодируются еще две буквы *b* и завершающий маркер *#*. Символ *#* отбрасывается, так как он искусственно добавлен кодером и не является частью сообщения "bab".

Алгоритм Зива-Лемпеля (алгоритм LZ)

Алгоритм LZ – адаптивный метод кодирования и не требует предварительного просмотра сообщения.

Изначально в словаре определена пустая строка, имеющая номер 0. Поступающий первый символ приписывается к пустой строке, образуя тем самым новую строку. Эта строка заносится в выходной поток. Также строка записывается в словарь под номером 1. Фактически сохраняется не строка, а номер пустой строки и сам добавляемый символ. При поступлении следующего символа (то есть строки, состоящей из одного символа) смотрится, есть ли эта строка в словаре. Если она обнаружена в словаре, то к этой строке добавляется следующий символ из входного потока. Полученная в результате новая строка проверяется на наличие в словаре. Если ее нет, то она записывается в словарь под номером, следующим после последнего на текущий момент, в виде пары (номер образующей строки; дополнительный символ). Эта же пара записывается в выходной поток. Если же эта строка уже имеется в словаре, то к ней приписывается следующий символ из потока, и вновь проверяется наличие полученной строки в словаре. Так продолжается до тех пор, пока новая строка не будет записана в словарь. Затем читается следующий символ из входного потока и действия повторяются. Каждая новая пара, добавляемая в словарь, записывается в выходной поток.

При декодировании после получения первой пары (номер пустой строки (0); первый символ) декодер записывает ее в словарь, который изначально пуст, а в выходной поток записывает первый символ. После получения n -ой пары (номер составляющей строки k ; символ) декодер строит строку, состоящую из символа и стоящей перед ним строки, находящейся в словаре под номером k , которая в свою очередь тоже состоит из символа и стоящей перед ним строки. Декодер приписывает в начало строки по символу, пока не дойдет до строки, состоящей из пустой строки и символа. Этот символ ставится в начало строки, и полученная в результате строка записывается в выходной поток.

Данный метод эффективен для сжатия текстов, так как в них присутствует большое количество повторений одних и тех же последовательностей символов (окончаний, суффиксов слов и т.п.) и слов.

Для примера закодируем сообщение "*abaaababbbaabba*". Считывается первый символ *a* и записывается в словарь в виде пары (0*a*) под номером 1. Второй символ записывается в виде пары (0*b*) под номером 2, так как строки "*b*" еще нет в словаре. Прочитав третий символ,

убеждаемся, что строка "a" уже есть в словаре под номером 1. Тогда прочитывается следующий символ и он приписывается в конец строки, то есть формируется пара (1a). Убеждаемся, что такой пары в словаре нет. Она заносится в словарь под номером 3. В результате последующих действий получается схема кодирования, представленная в табл. 8.

Таблица 8

Схема LZ-кодирования строки "abaaababbbaabba"

Разбор сообщения	a	b	aa	ab	abb	ba	abba
Пары в словаре	0a	0b	1a	1b	4b	2a	5a
Номер пары в словаре	1	2	3	4	5	6	7

Если учесть, что каждая пара кодируется двумя байтами, то код будет занимать 14 байт, а исходное сообщение занимает 15. Слабый коэффициент сжатия объясняется тем, что текст короткий. На начальном этапе размер кода даже превышает размер исходного текста: один символ кодируется двумя. Но по мере поступления все новых и новых символов длина кодируемых двумя байтами строк увеличивается, так как используются уже имеющиеся в словаре пары.

При декодировании текста декодер получает пару (0a), заносит ее в словарь, а строку, состоящую из элементов этой пары, записывает в выходной поток. То же самое он делает с парой (0b). При получении пары (1a) он заносит ее в словарь, и составляет строку из строки с номером 1 ("a") и символа a. Полученную строку "aa" декодер записывает в выходной поток. То же самое он делает с парой 4. Получив пару 5, декодер заносит ее в словарь и формирует строку, состоящую из строки 4 (которая состоит из строки 1 и символа b) и символа b. Сформированная строка "abb" записывается в выходной поток. Так продолжается до тех пор, пока не будет считана последняя пара и не будет записана последняя строка в выходной поток. В результате получаем исходное сообщение "abaaababbbaabba".

Алгоритм Зива-Лемпеля-Велча (алгоритм LZW)

Алгоритм LZW – усовершенствованная версия алгоритма LZ.

Исходные символы получают свои кодовые номера, например, ASCII-коды. Эту информацию не надо передавать декодеру, так как она общеизвестна. Встретившиеся в тексте цепочки символов заносятся в кодовую таблицу для дальнейшего использования. Декодирование выполняется по схеме алгоритма LZ.

Например, закодируем сообщение "abaaababbbaabba". Изначально словарь состоит из 256 символов с номерами от 0 до 255. Алгоритм кодирования представлен в табл. 9.

Таблица 9

Схема LZW-кодирования строки "abaaababbbaabba"

	<i>a</i>				Букву <i>a</i> сохраняем как префикс
<i>a</i>	<i>b</i>	[<i>a</i>]	<i>ab</i>	256	Строка " <i>ab</i> " не встречалась, регистрируем ее с кодом 256, код буквы <i>a</i> выводим в результат, букву <i>b</i> запоминаем как префикс.
<i>b</i>	<i>a</i>	[<i>b</i>]	<i>ba</i>	257	Аналогично.
<i>a</i>	<i>a</i>	[<i>a</i>]	<i>aa</i>	258	Аналогично.
<i>a</i>	<i>a</i>				Строка " <i>aa</i> " есть в кодовой таблице, продолжаем кодовый процесс с увеличенным префиксом.
<i>aa</i>	<i>b</i>	258	<i>aab</i>	259	Регистрируем " <i>aab</i> ", выводим код строки " <i>aa</i> ".
<i>b</i>	<i>a</i>				Строка " <i>ba</i> " есть в кодовой таблице, продолжаем кодовый процесс с увеличенным префиксом.
<i>ba</i>	<i>b</i>	257	<i>bab</i>	260	Регистрируем " <i>bab</i> ", выводим код строки " <i>ba</i> ".
<i>b</i>	<i>b</i>	[<i>b</i>]	<i>bb</i>	261	Строка " <i>bb</i> " не встречалась, регистрируем ее с кодом 261, код буквы <i>b</i> выводим в результат, букву <i>b</i> запоминаем как префикс.
<i>b</i>	<i>b</i>				Строка " <i>bb</i> " уже есть в кодовой таблице, продолжаем кодовый процесс с увеличенным префиксом.
<i>bb</i>	<i>a</i>	261	<i>bba</i>	262	Регистрируем " <i>bba</i> ", выводим код строки " <i>bb</i> ".
<i>a</i>	<i>a</i>				Строка " <i>aa</i> " уже есть в кодовой таблице, продолжаем кодовый процесс с увеличенным префиксом.
<i>aa</i>	<i>b</i>				Строка " <i>aab</i> " уже есть в кодовой таблице, продолжаем кодовый процесс с увеличенным префиксом.
<i>aab</i>	<i>b</i>	259	<i>aabb</i>	263	Регистрируем " <i>aabb</i> ", выводим код строки " <i>aab</i> ".
<i>b</i>	<i>a</i>				Строка " <i>ba</i> " уже есть в кодовой таблице, продолжаем кодовый процесс с увеличенным префиксом.
<i>ba</i>	<i>eof</i>	257			Конец файла, выводим префикс " <i>ba</i> ".

В первой колонке приведен префикс рассматриваемой строки (в начале он пустой). Этот префикс представлен кодом, полученным ранее. Каждый последующий код в таблице представляется парой: (код префикса; код символа). Во второй колонке указан очередной символ входного текста. Если текущая строка – объединение префикса и символа – уже включена в кодовую таблицу, то она становится новым префиксом. В противном случае она регистрируется, (непустой) префикс выводится как результат (третья колонка), а символ заменяет его как новый префикс. В итоге код состоит из 9 чисел: кодов символов и строк в словаре (коды символов – стандартные ASCII-коды).

Псевдокод алгоритма кодирования LZW:

```

СТРОКА = очередной символ из входного потока
WHILE(входной поток не пуст)
{
    СИМВОЛ = очередной символ из входного потока
    IF(СТРОКА+СИМВОЛ есть в словаре)
        СТРОКА = СТРОКА+СИМВОЛ
    ELSE
    {
        вывести в выходной поток код для СТРОКА
        добавить в словарь СТРОКА+СИМВОЛ
        СТРОКА = СИМВОЛ
    }
}
вывести в выходной поток код для СТРОКА

```

Расшифруем полученный код (97, 98, 97, 258, 257, 98, 261, 259, 257). Числа 97 и 98 – стандартные ASCII-коды символов *a* и *b*.

Декодирование представлено в табл. 10. Поступающие на вход декодера коды записаны во втором столбце. В первом столбце указаны префиксы, которые являются строками, отправленными в выходной поток на предыдущем шаге. Отправляемые строки указаны в третьем столбце. Столбцы 4 и 5 – составление словаря, который пополняется на каждом шаге. Этот словарь идентичен составленному кодером, хотя не все его элементы были использованы при кодировании. Соединив все строки, получаем исходное сообщение "abaaababbbbaabba".

Таблица 10

Схема LZW-декодирования кодовой последовательности (97, 98, 97, 258, 257, 98, 261, 259, 257)

[a]	a				Букву <i>a</i> запоминаем как префикс и как постфикс, а также отправляем ее в выходной поток.
<i>a</i>	[<i>b</i>]	<i>b</i>	<i>ab</i>	256	Строка с кодом [<i>b</i>] уже есть в словаре, поэтому сразу отправляем ее в выходной поток и запоминаем как префикс. В таблицу записываем новый элемент, составленный из предыдущего префикса и первого символа полученной строки. Постфиксом становится прочитанный код [<i>b</i>].
<i>b</i>	[<i>a</i>]	<i>a</i>	<i>ba</i>	257	Строка с кодом [<i>a</i>] уже есть в словаре, то все выполняется аналогично предыдущему случаю. Записываем в словарь новый элемент. Постфиксом становится прочитанный код [<i>a</i>].
<i>a</i>	258	<i>aa</i>	<i>aa</i>	258	Строки с кодом 258 еще нет в словаре, поэтому составляем строку из префикса (<i>a</i>) и постфикса (<i>a</i>) и записываем ее в выходной поток. Записываем в словарь новый элемент, составленный из префикса и первого элемента полученной строки. Постфиксом становится прочитанный код 258. Он известен, так как его только что занесли в словарь ("aa").
<i>aa</i>	257	<i>ba</i>	<i>aab</i>	259	Строка с кодом 257 уже есть в словаре, записываем ее в выходной поток. Постфикс – <i>b</i> . Префикс – "ba".
<i>ba</i>	[<i>b</i>]	<i>b</i>	<i>bab</i>	260	Аналогично предыдущему случаю.
<i>b</i>	261	<i>bb</i>	<i>bb</i>	261	Строки с кодом 261 еще нет в словаре. Получаем строку из префикса и постфикс и записываем ее в выходной поток.
<i>bb</i>	259	<i>aab</i>	<i>bba</i>	262	Аналогично, как в случае с кодом 257.
<i>aab</i>	257	<i>ba</i>	<i>aabb</i>	263	Аналогично предыдущему случаю.

Псевдокод алгоритма декодирования LZW:

```

читать ТЕКУЩИЙ_КОД
вывести в выходной поток ТЕКУЩИЙ_КОД
СИМВОЛ = ТЕКУЩИЙ_КОД
WHILE( входной поток не пуст )
{
    читать из входного потока СЛЕДУЮЩИЙ_КОД
    IF( в словаре СЛЕДУЮЩИЙ_КОД отсутствует )
    {
        СТРОКА = строка, соответствующая ТЕКУЩЕМУ_КОДУ
                                         в словаре
    }
}
```

```

СТРОКА = СТРОКА+СИМВОЛ
}
ELSE
    СТРОКА = строка, соответствующая СЛЕДУЮЩЕМУ_КОДУ
                                в словаре
    вывести в выходной поток СТРОКУ
    СИМВОЛ = первый символ СТРОКИ
    добавить в словарь ТЕКУЩИЙ_КОД+СИМВОЛ
    ТЕКУЩИЙ _КОД = СЛЕДУЮЩИЙ_КОД
}

```

Порядок выполнения работы

1. Согласовать с преподавателем два разных алгоритма кодирования.
2. Создать рабочий файл, содержащий информацию, предназначенную для кодирования. Использовать алфавит из 3...5 символов. Размер файла – более 100 000 символов.
3. Разработать две утилиты, реализующие процедуры кодирования и декодирования для обоих алгоритмов. Ключ, определяющий режим работы утилиты (кодирование или декодирование), пути к входному и выходному файлам должны задаваться в командной строке запуска утилит. Программы должны содержать функции подсчета длительности кодирования/декодирования и коэффициента сжатия (отношения размера закодированного файла к размеру исходного).
4. Закодировать входной файл при помощи одной утилиты. Зафиксировать длительность кодирования и степень сжатия для первого алгоритма.
5. Закодировать тот же входной файл при помощи второй утилиты. Зафиксировать длительность кодирования и степень сжатия для второго алгоритма.
6. Декодировать файл, полученный на выходе первой утилиты. Зафиксировать длительность декодирования для первого алгоритма.
7. Декодировать файл, полученный на выходе второй утилиты. Зафиксировать длительность декодирования для второго алгоритма.
8. Полученный на выходе первой утилиты закодированный файл направить на вход второй утилиты кодирования и закодировать его.

9. Сначала с помощью второй, а затем первой утилиты декодировать полученный в п. 8 файл. Сравнить декодированный файл с исходным, убедиться в их идентичности.

10. Повторить выполнение пп. 4-9 для произвольного бинарного файла (например, файла в формате Word , Adobe) размером более 2 Мб.

11. Создать новый рабочий файл из одного символа – ноль в шестнадцатеричной кодировке. Повторить выполнение пп. 4-9.

12. Построив гистограммы, сопоставить характеристики двух исследованных алгоритмов (длительность кодирования, длительность декодирования, коэффициент сжатия) для текстового файла малого алфавита, для бинарного файла, для файла с одним символом. Обосновать полученные показатели. Отметить замечания и предложить рекомендации по оптимизации работы кодировщиков и декодировщиков.

Содержание отчета

1. Перечень реализованных алгоритмов.
2. Гистограммы, показывающие зависимости характеристик исследованных алгоритмов от типа кодируемого файла, размеров рабочего файла и алфавита.
3. Сравнительная оценка и обоснование характеристик алгоритмов для разных условий применения, рекомендации по доработке и оптимизации программ.
4. Ответы на контрольные вопросы.
5. Выводы по работе.

Контрольные вопросы

1. В чем заключаются преимущества и недостатки адаптивных методов кодирования по сравнению со статическими?
2. Опишите алгоритм получения арифметического кода?
3. Почему текстовые и бинарные файлы имеют разную степень сжатия?
4. В каких случаях кодирование не имеет смысла?
5. В чем заключается доработка алгоритма LZW по сравнению с алгоритмом LZ?

Лабораторная работа 8

ОСНОВЫ СТЕГАНОГРАФИЧЕСКОЙ ЗАЩИТЫ ИНФОРМАЦИИ

Цель работы – приобретение навыков исследования свойств стегоконтейнеров, разработки стегосистем и их применения для сокрытия данных при передаче с помощью графических изображений.

Теоретические сведения

Основные технологии, используемые для сокрытия информации, – это криптография и стеганография. Криптография применяется для защиты данных путем их шифрования, то есть применения криптографических алгоритмов. Стеганография позволяет скрывать сам факт наличия тайного сообщения. При этом скрываемое сообщение встраивается в некоторый не привлекающий внимания объект-контейнер, который может открыто пересыпаться адресату.

Развитие средств вычислительной техники дало толчок развитию компьютерной стеганографии, когда сообщение встраивают в различные цифровые данные: графические изображения, аудио- и видеозаписи.

Цифровая стеганография включает в себя следующие направления:
встраивание цифровых водяных знаков (ЦВЗ) (watermarking);
встраивание идентификационных номеров (fingerprinting);
встраивание заголовков (captioning);
встраивание информации с целью ее скрытой передачи.

Для интеллектуальной собственности, представленной в цифровом виде, актуальна проблема защиты авторских прав. Данная проблема решается с помощью встраивания в защищаемый объект невидимых меток – ЦВЗ. Такие знаки могут содержать некоторый аутентичный код, например, информацию о собственнике или авторе. Невидимые ЦВЗ анализируются специальным декодером, проверяющим их аутентичность.

Технология встраивания идентификационных номеров производителей имеет много общего с технологией ЦВЗ. Отличие заключается в том, что при встраивании номеров защищенная копия имеет

свой уникальный идентификатор. Этот номер позволяет разработчику отслеживать перемещения информации.

Встраивание невидимых заголовков служит для каталогизации и хранение разнородно представленной информации как единого целого. При этом нарушитель в явном виде отсутствует. Встраивание невидимых заголовков применяется, например, для подписи медицинских снимков.

Встраивание информации с целью ее скрытой передачи является старейшим способом применения стеганографии. Встраивание и выделение скрытых сообщений реализует стегосистема (рис. 9).

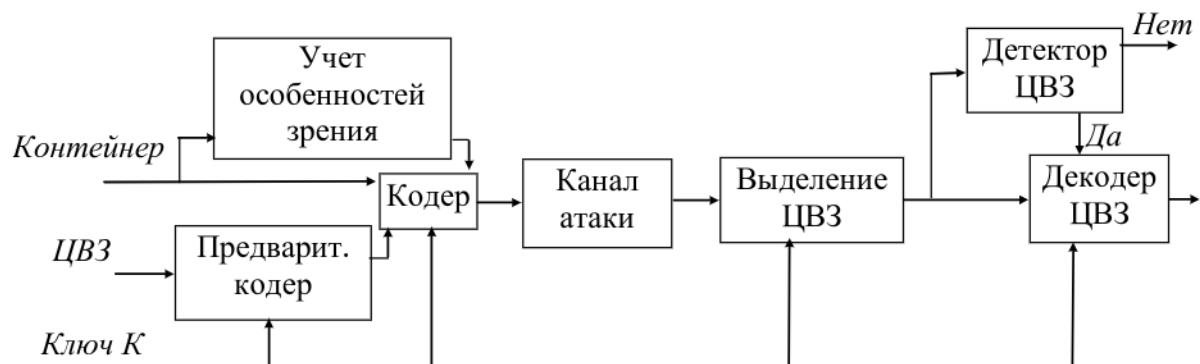


Рис. 9. Структура стегосистемы

Стегоконтейнер – информационная последовательность, в которой скрывается сообщение.

Различают два типа стегоконтейнеров:

потоковый. Непрерывно следующая последовательность бит вкладывается в контейнер в реальном масштабе времени. При этом в кодере заранее не известно, достаточно ли объема контейнера для передачи всего сообщения;

фиксированный. Размеры и характеристики контейнера известны заранее, что позволяет оптимально упаковывать данные.

В дальнейшем рассматриваются фиксированные контейнеры.

Предварительный кодер – устройство, предназначенное для преобразования скрываемого сообщения к виду, удобному для встраивания в стегоконтейнер. Например, если в качестве контейнера выступает изображение, то сообщение представляется как двумерный массив бит.

Предварительная обработка часто выполняется с использованием ключа K для повышения секретности встраивания.

Кодер – устройство, предназначенное для осуществления вложения сообщения в другие данные с учетом их модели, например, путем модификации младших значащих бит. Например, в изображении с 256 градациями серого (каждый пиксель изображения кодируется 8 битами), глаз человека не способен заметить различия в цвете на один разряд, поэтому младший бит используют для скрытия информации. Данный метод прост в реализации и эффективен.

Детектор – устройство, предназначенное для определения наличия стегосообщения.

Декодер – устройство, восстанавливающее скрытое сообщение. Данный узел может отсутствовать.

Стегосистема образует стегоканал, по которому передается заполненный стегоконтейнер. Данный канал считается подверженным воздействиям со стороны нарушителей или помех.

Нарушитель стегосистемы может быть пассивным, активным и злоумышленным. В зависимости от этого он может создавать различные угрозы. Пассивный нарушитель может обнаружить наличие стегоканала, при этом содержание секретного сообщения ему может быть неизвестно. Активный нарушитель своими действиями способен удалить или разрушить сообщение. Действия злоумышленного нарушителя наиболее опасны – он способен не только разрушать, но и создавать ложные стегоконтейнеры. Для осуществления той или иной угрозы нарушитель применяет атаки. Простейшая атака – субъективная: нарушитель внимательно рассматривает изображение, пытаясь определить "на глаз", имеется ли в нем скрытое сообщение.

В дальнейшем в качестве стегоконтейнера используется изображение. Практически любой способ обработки изображений может привести к разрушению части или всего встроенного сообщения. Например, операции низкочастотной фильтрации, операции обработки изображений (масштабирование, повороты, усечение, перестановка пикселей). Ситуация усугубляется тем, что преобразования стегосообщения могут осуществляться не только нарушителем, но и

законным пользователем, или являться следствием ошибок при передаче по каналу связи.

Атаки на стегосистемы:

атаки на сообщение. Направлены на извлечение стегосообщения без искажения контейнера. Примеры атак: очистка сигналов от шумов, нелинейная фильтрация;

атаки на стегоконтейнер. Направлены на удаление или порчу сообщения путем различных преобразований, проводимых над стегоконтейнером. Примеры атак: линейная фильтрация, сжатие изображений, добавление шума, изменение контрастности;

атаки на детектор. Направлены на затруднение работы или блокировку детектора. При этом сообщение в стегоконтейнере остается, но теряется возможность его приема. Примеры атак: масштабирование, сдвиги, повороты, усечение изображения, перестановка пикселей;

атаки на механизм использования стегосообщений. В основном связаны с созданием ложных стегосообщений, добавлением нескольких сообщений.

Один из способов решения проблемы прав собственности заключается во встраивании в стегоконтейнер некоторой временной отметки, предоставляемой третьей, доверенной стороной. В случае возникновения конфликта лицо, имеющее на изображении более раннюю временную отметку, считается настоящим собственником.

Для защиты от атак аффинного преобразования (масштабирование, сдвиг, поворот, усечение изображения) можно использовать дополнительные ЦВЗ. Эти ЦВЗ не несут в себе информации, но используются для регистрации выполняемых нарушителем преобразований. В этом случае атака может быть направлена именно против дополнительного ЦВЗ. Другой альтернативой является вложение ЦВЗ в визуально значимые области изображения, которые не могут быть удалены из него без существенной его деградации.

Другим методом защиты от подобных атак является блочный детектор. Модифицированное изображение разбивается на блоки размером 12×12 или 16×16 пикселей, и для каждого блока анализируются все возможные искажения, то есть пиксели в блоке подвергаются поворотам, перестановкам и т.п. Для каждого изменения определяется коэффициент

ЦВЗ. Преобразование, после которого коэффициент оказался наибольшим, считается реально выполненным нарушителем. Таким образом, появляется возможность обратить внесенные нарушителем искажения.

Безопасность стегосистем оценивается их стойкостью, то есть способностью стегосистемы скрывать от квалифицированного нарушителя факт передачи сообщений, способностью противостоять попыткам нарушителя разрушить, исказить, удалить скрытно передаваемые сообщения, а также способностью подтвердить или опровергнуть подлинность скрытно передаваемой информации.

Стегосистема является стойкой, если нарушитель, наблюдая информационный обмен между отправителем и получателем, не способен обнаружить, что под прикрытием контейнера передается сообщение.

По сравнению с криптографическими системами оценка безопасности стегосистем более сложная, что объясняется недостаточной теоретической и практической проработкой вопросов их стойкости.

Порядок выполнения работы

1. Создать произвольный графический файл формата BMP, который будет использоваться в качестве стегоконтейнера. Для этого следует с помощью графического редактора (например, *Paint*, входящего в штатный набор программ операционной системы Windows), создать произвольный рисунок и сохранить его в соответствующем формате.

2. Открыть файл изображения с помощью HEX-редактора (например, *HIEW*). Найти основные блоки, входящие в файл соответствующего формата. Отметить наличие или отсутствие глобальной палитры. При наличии глобальной палитры указать ее размер.

3. Создать программу-кодер, реализующую стеганографическое сокрытие "секретного" текстового файла в созданный графический файл. Программа должна соответствовать требованиям:

для сокрытия текста должен применяться метод замены наименее значимых бит в глобальной палитре и в содержательной части графического файла по алгоритму: в каждом байте палитры младший бит заменяется битом шифруемого сообщения и полученный байт записывается в результирующий файл;

помимо самого "секретного" сообщения, графический файл, используемый в качестве контейнера, должен включать в себя необходимую для восстановления информацию о скрываемом сообщении: тип скрываемого файла и его размер;

во входных параметрах программы должна быть обозначена степень упаковки скрываемого сообщения, которая указывала бы на количество бит, помещаемое в один байт графического файла, например, один бит сообщения на байт графической информации (три бита на точку в цветовой схеме RGB) или два бита на байт графической информации (шесть битов на точку);

если размеры скрываемого текста не позволяют полностью поместить этот текст в графический файл, при заданной степени упаковки, то процесс скрытия должен быть прерван с указанием причины завершения работы программы, размера текущего сообщения, максимальной вместимости стегоконтейнера, вычисленной по параметрам данного графического файла. Расчет производится по размерам глобальной палитры, скрываемого сообщения и соответствующей степени упаковки в графический файл.

4. Запустить разработанную программу-кодер на выполнение, указывая различные значения степени упаковки в качестве параметра командной строки исполняемого файла. При этом визуально сравнить полученные графические файлы, содержащие упакованное "секретное" сообщение, с исходным изображением. Сравнить размер графического файла (стегоконтейнера) до упаковки в нем стегосообщения и после. Отметить влияние степени упаковки на качество изображения, получаемого в результате сокрытия в нем стегосообщения.

5. Создать программу-декодер, осуществляющую извлечение "секретного" сообщения из стегоконтейнера.

6. Запустить разработанную программу-декодер, используя в качестве входного параметра графический файл, который был получен в п. 4. Извлечь "секретное" сообщение. Сравнить его с исходным.

7. Запустить программу-кодер на выполнение, задав в качестве входных параметров текстовый файл, размер которого превышает объем стегоконтейнера. Отметить полученные результаты.

8. Открыть с помощью графического редактора файл изображения, содержащий "секретное" сообщение. Выполнить преобразования изображения: растяжение, поворот, отражение. Отметить параметры преобразования, которые были выполнены над графическим файлом. Сохранить результирующее изображение в тот же файл.

9. Запустить программу-декодер, используя в качестве входного параметра графический файл, который был получен в результате преобразований в п. 8. Сравнить расшифрованное и исходное сообщения.

Содержание отчета

1. Описание формата файла изображения. Исходный размер файла изображения, наличие или отсутствие в нем глобальной палитры, размер палитры, если она присутствует, размер содержательной части графического файла.

2. Математический расчет максимальной длины скрываемого сообщения, помещаемого в созданный стегоконтейнер.

3. Листинги программы-кодера и программы-декодера.

4. Результаты анализа влияния степеней упаковки на полученное графическое изображение.

5. Описание преобразований, производимых с графическим файлом.

6. Результаты анализа влияния преобразований, производимых над графическим файлом на полученное декодированное сообщение.

7. Ответы на контрольные вопросы.

8. Выводы по работе.

Контрольные вопросы

1. Какие функции выполняет каждый элемент стегосистемы?

2. Перечислите основные виды атак на стегосистемы.

3. Что означает термин "стойкость стегосистемы"?

4. Какое влияние оказывает сжатие графических изображений на алгоритмы встраивания стегосообщений?

5. Каковы методы противодействия стеганографическим атакам?

Лабораторная работа 9

МЕТОДЫ КОНТРОЛЯ ЦЕЛОСТНОСТИ

Цель работы – приобретение навыков в разработке и использовании программных средств контроля целостности информации, анализ свойств алгоритмов расчета контрольных сумм и циклических кодов.

Теоретические сведения

Целостность – состояние информации, при котором отсутствует ее изменение либо изменение осуществляется только авторизованными субъектами. Наиболее простой способ проверки целостности – подсчет контрольных сумм.

Контрольная сумма – число, рассчитанное путем сложения всех кодов из потока входных данных. Если полученный результат превышает некоторое максимально допустимое значение, то контрольная сумма C – остаток от деления: $C = T \% (M + 1)$, где T – сумма входных данных, M – максимально допустимое значение контрольной суммы. Например, для последовательности ASCII-кодов

36 211 163 4 109 192 58 247 47 92

контрольная сумма – однобайтовая величина и ее максимально допустимое значение составляет 255. Для приведенного массива сумма кодов равна 1159, контрольная сумма – $1159\%256 = 135$. Если при пересылке контрольная сумма, рассчитанная отправителем, равнялась 135, а при получении она изменилась, то целостность информации нарушена.

Недостаток метода контрольных сумм заключается в том, что, хотя несовпадение значений сумм служит доказательством изменения информации, их равенство не гарантирует, что информация осталась неизменной. Например, перемена мест кодов в последовательности не приводит к изменению суммы, но при этом целостность очевидно нарушена. При использовании восьмиразрядной контрольной суммы вероятность того, что суммы двух строк совпадут, равна $1/2^8 = 0,4\%$. При увеличении разрядности переменной, выделенной под контрольную сумму, вероятность совпадения уменьшится, однако сам механизм остается чувствительным к такой ошибке.

Более совершенным способом проверки достоверности данных является вычисление *циклического избыточного кода* (cyclic redundancy check, CRC). Алгоритмы расчета CRC широко используются в сетевых адаптерах, дисковых контроллерах и в других устройствах для проверки идентичности входной и выходной информации. Этот механизм применяется в коммуникационных программах и архиваторах с целью проверки целостности передаваемой и сохраняемой информации.

Механизм CRC основан на полиномиальной арифметике. Делимое, делитель, частное и остаток рассматриваются не как целые числа, а как полиномы с двоичными коэффициентами. Число записывается в виде двоичной строки, биты которой служат коэффициентами полинома.

Например, числу 23 (10111b) соответствует полином $1x^4 + 0x^3 + 1x^2 + 1x^1 + 1x^0$, то есть $x^4 + x^2 + x^1 + x^0$. Арифметические операции осуществляются как операции над полиномами. Например, произведение 13 (1101b) на 11 (1011b) полиномиально представляется как

$$(x^3+x^2+x^0)(x^3+x^1+x^0) = x^6+x^4+x^3+x^5+x^3+x^2+x^3+x^1+x^0 = \\ = x^6+x^5+x^4+3x^3+x^2+x^1+x^0.$$

В контрольном суммировании важна полиномиальная арифметика по модулю 2, где все коэффициенты вычисляются по модулю 2. В качестве x берется 2 и коэффициенты приводятся к двоичным с учетом переноса единицы в старшие разряды: например, $x^7+x^3+x^2+x^1+x^0$, что соответствует числу 143 (10001111b). Это обычная арифметика, в которой основание системы счисления записано явно. Суть в том, что если x неизвестно, то невозможно производить переносы. Если $x=2$, то перенос возможен: $3x^3=x^4+x^3$. В полиномиальной арифметике отношения между коэффициентами не определены, и коэффициенты при разных степенях полностью изолированы.

С отменой переноса исчезают различия между сложением и вычитанием, что эквивалентно поразрядной логической операции XOR, которая обратна сама себе. Следствием является слабое отношение порядка: число X больше числа Y , если номер позиции старшей единицы числа X больше номера позиции старшей единицы числа Y . Этот номер,

отсчитываемый с нуля, называется длиной числа. Например, $1010b$ больше, чем $10b$; но нет причин полагать, что $1010b$ больше, чем $1001b$.

Умножение и деление выполняются с учетом слабого отношения порядка и правил XOR.

Например, умножение

$$\begin{array}{r} 1101 \\ \underline{1011} \\ 1101 \\ 1101. \\ 0000.. \\ \underline{1101...} \\ 1111111 \end{array}$$

Число A кратно числу B (A делится на B), если A можно получить, складывая согласно правилам XOR результаты различных сдвигов влево числа B .

Например, $111010110b$ кратно $11b$:

$$\begin{array}{r} 111010110 \\ = \dots .11. \\ + \dots 11.... \\ + \dots 11..... \\ 11..... \end{array}$$

Однако $111010111b$ уже нельзя составить из сдвигов $11b$, поэтому $111010111b$ не делится на $11b$.

Вычисление CRC интерпретируется как взятие остатка от деления. Для этого, во-первых, выбирают делитель. Его длина должна превышать длину итогового CRC на один разряд, то есть для получения CRC16 необходим 17-битный делитель. Это связано с тем, что остаток должен быть меньше делителя, иначе можно выполнить деление еще один раз. Не все делители дают хорошие результаты, поэтому обычно выбирают так называемые стандартные делители (табл. 11).

Таблица 11

Стандартные делители для расчета CRC

Обозначение	Размер, бит	Степени полинома
CRC32 (Ethernet)	32	32, 26, 23, 22, 16, 12, 11, 10, 8, 7, 5, 4, 2, 1, 0
CRC-CCITT (V.41, X.25)	16	16, 12, 5, 0
CRC16	16	16, 15, 2, 0
CRC12	12	12, 11, 3, 2, 1, 0
LRC8	8	8, 0
VRC (Parity)	1	1, 0

Перед выполнением деления сообщение дополняют W нулями, где W – длина делителя. Сам источник сообщения, например, текстовый файл, при этом не модифицируют. Длина делителя определяется степенью полинома, его представляющего. Например, исходное сообщение 1101011011, делитель – 10011, степень полинома – 4, тогда дополненное сообщение будет равно 11010110110000. Поиск остатка выполняется делением в столбик с применением операции XOR:

$$\begin{array}{r}
 11010110110000 : 10011 = 1100001010 \\
 \underline{10011} \\
 10011 \\
 \underline{10011} \\
 10110 \\
 \underline{10011} \\
 10100 \\
 \underline{10011} \\
 \mathbf{1110} - \text{остаток от деления (длина CRC - 4 бита)}
 \end{array}$$

В общем виде расчет CRC выполняется следующим образом:

1. Выбор длины W и делителя G длиной W .
2. Добавление W нулей к исходному сообщению M . Получается сообщение M' .
3. Деление M' на G с помощью XOR-арифметики. Остаток от деления – искомый код CRC.

В программных реализациях расчета CRC обычно используют сдвиговые бинарные операции.

Код CRC либо дописывается в конец исходного сообщения вместо добавленных нулей (в этом случае исходное сообщение разрастается из-за дозаписи CRC), либо передается независимо от сообщения (в этом случае отдельно передаются собственно исходное сообщение и CRC). Приемник может либо отделить CRC от полученного сообщения, добавить нули в конец сообщения, заново рассчитать CRC, и сравнить с исходным, либо подсчитать CRC для всей переданной цепочки, не добавляя нули, и проверить, получится ли нулевой результат.

Передаваемое сообщение T кратно делителю. Последние W бит сообщения T – остаток от деления сообщения M' на делитель, то есть $T = M' + \text{CRC}$ ("+" означает сложение, а не добавление в конец), а поскольку сложение одновременно является вычитанием, то оно "уменьшает" M' до ближайшего кратного G . Предположим, сообщение пришло с ошибкой. Его можно записать как $T + E$, где E – вектор ошибки. Приемник делит $T + E$ на G . Поскольку $T \bmod G = 0$, то $(T+E) \bmod G = E \bmod G$. Поэтому способность метода отлавливать специфические ошибки будет определяться количеством E , кратных G , поскольку такие помехи не могут быть обнаружены.

Некоторые типы ошибок, которые встречаются при передаче:

ошибка в одном бите. $E = 100\dots000$. Такие ошибки можно отловить, если в G не менее двух битов установлено в 1. Как бы мы ни складывали сдвиги таких G , всегда получим строку, в которой, по крайней мере, два единичных бита, значит, E не может быть кратно G ;

ошибка в двух битах. Надо подобрать G , для которых не являются кратными числа типа 11, 101, 1001, и т.д.;

ошибка в нечетном количестве бит. Такие ошибки можно отловить, взяв G , в котором количество бит четно. Это следует из того, что CRC-умножение является результатом XOR константы в аккумулятор с разными сдвигами. XOR – поразрядная операция, инвертирующая те биты аккумулятора, напротив которых в прибавляемой строке стоит 1. Если сделать XOR кода с четным числом единиц, то в аккумуляторе инвертируется четное количество единиц, в результате чего четность количества единиц аккумулятора сохранится;

ошибка пакета. $E = 000\dots000111\dots111000\dots000$. Ошибка локализована в непрерывном пакете внутри сообщения. E можно представить в виде

произведения: $E = (1000....00)(11111111)$, где первый множитель содержит Z нулей, а второй – N единиц. Если в G младший бит – 1, то левый множитель не может быть делителем G , тогда, если G длиннее правого множителя, то ошибка будет отловлена.

Порядок выполнения работы

Перед выполнением лабораторной работы необходимо для одного произвольного бинарного числа размером 8 разрядов вручную рассчитать

LRC8 с помощью полиномиального деления;

LRC8 с помощью сдвиговых операций;

CRC для полинома (8, 7, 1, 0) с помощью полиномиального деления.

1. Разработать программу, реализующую расчет восьмиразрядной контрольной суммы по формуле $C = T \% (M + 1)$ для произвольного текстового файла.

2. Создать тестовый файл, записать в него "секретную" информацию (100 символов). С помощью созданной программы подсчитать эталонное значение контрольной суммы для данных из тестового файла.

3. Изменить один из символов входного потока. Произвести расчет контрольной суммы. Сравнить полученное значение с эталоном.

4. Добавить в конец исходной строки ее копию, то есть продублировать строку. Рассчитать контрольную сумму. Сравнить полученные результаты с предыдущими.

5. Поменять местами два символа в тестовом файле. Рассчитать контрольную сумму, сравнить ее с ранее полученными значениями.

6. Получить у преподавателя формулу полинома-делителя CRC и разработать программу вычисления CRC с помощью правил полиномиального деления. Для удобства расчетов следует использовать свойства операции XOR, выполняя полиномиальное деление в блоках текста с последующей сборкой результатов при помощи операции XOR. Программа должна выдавать значение CRC в символьном виде, а также в десятичной, двоичной и шестнадцатеричной системах счисления.

7. С помощью созданной программы подсчитать эталонное значение CRC для тестового файла.

8. Повторить выполнение пп. 3...5. Сравнить все полученные значения CRC с эталонным значением, полученным в п. 7.

9. Используя HEX-редактор (например, HVIEW, HexWorkshop) добавить в начало "секретных" данных двоичные нули. Рассчитать значение CRC, объяснить полученный результат.

10. К концу контрольного текста в режиме HEX дописать полученное значение CRC. Рассчитать новый код CRC, прокомментировать результат.

11. К концу модифицированного в п. 10 текста в режиме HEX повторно добавить CRC-код, полученный в п. 9. Рассчитать новый CRC, прокомментировать результат.

12. К концу модифицированного в п. 11 текста в режиме HEX дописать байт 80h. Рассчитать CRC, прокомментировать результат.

13. Выполнить расчет CRC для произвольного бинарного файла объемом более 2 Мбайт (например, файл в формате Word, Adobe). К концу исходного файла в режиме HEX дважды дописать CRC, вычислить значение CRC и сопоставить его с полученным в п. 11, прокомментировать результат.

14. Модифицировать программу расчета CRC, задав другой полином-делитель той же степени. Путем тестовых прогонов программы, найти два полинома, которые вычисляют значения CRC для одного и тог же рабочего файла, но при добавлении их в конец файла для проверки цикличности кодов не дают нулевой CRC (не показывают достоверность информации). Продолжить тестовые прогоны программы и найти два полинома, которые обладают свойством стандартного полинома CRC. Прокомментировать результаты исследования.

15. Модифицировать программу расчета CRC, использовав снова стандартный полином, но задействовав вместо полиномиального деления бинарные сдвиговые операции. Рассчитать CRC для исходного контрольного текста. Сравнить полученное значение CRC с эталонным CRC, вычисленным в п. 7. Прокомментировать полученный результат.

Содержание отчета

1. Пример расчета LRC8 с помощью полиномиального деления.
2. Пример расчета CRC для полинома (8, 7, 1, 0) с помощью полиномиального деления.

3. Пример расчета LRC8 с помощью бинарных сдвиговых операций.
4. Исходные тексты разработанных программ.
5. Перечень используемых полиномов.
6. Все модификации "секретного" сообщения и соответствующие значения контрольных сумм и CRC в процессе выполнения работы.
7. Результаты анализа свойств используемых способов расчета контрольных сумм и CRC по отношению к воздействиям на данные.
8. Результаты расчета CRC для бинарного файла.
9. Результаты расчета CRC для программы, реализующей сдвиговые операции.
10. Ответы на контрольные вопросы.
11. Выводы по работе и предложения, какие изменения необходимо внести в разработанные программы, чтобы улучшить их быстродействие.

Контрольные вопросы

1. В каких случаях с помощью контрольной суммы можно обнаружить подмену или искажение информации?
2. Почему при подсчете CRC применяется полиномиальная арифметика по модулю 2?
3. Какие ошибки CRC могут возникнуть при передаче пакета?
4. Почему для расчета CRC применяют определенные полиномы?
5. Можно ли, зная CRC, исправить ошибку при передаче текста?

Ответ обосновать.

Лабораторная работа 10

МЕТОДЫ НАДЕЖНОГО ХРАНЕНИЯ И ПЕРЕДАЧИ ИНФОРМАЦИИ

Цель работы – ознакомление с методом Хемминга помехоустойчивого кодирования, позволяющим обнаруживать и автоматически исправлять ошибки, возникающие при хранении и передаче информации в ненадежных средах.

Теоретические сведения

Среда, в которой передается, обрабатывается и хранится информация, не может быть абсолютно надежной. В беспроводных системах связи и телекоммуникационных системах уровень помех бывает очень высоким. Наиболее простой способ проверки целостности передаваемых данных – использование контрольных сумм и CRC. Однако такой подход позволяет только обнаруживать влияние среды на передаваемую информацию. В том случае, если необходимо избежать повторной передачи данных и исправить ошибки в уже полученных данных, применяется помехоустойчивое кодирование.

Теорема Шеннона утверждает, что при любой производительности источника данных, меньшей, чем пропускная способность канала, существует такой способ кодирования, который обеспечивает передачу всей информации со сколь угодно малой вероятностью ошибки.

Методы помехоустойчивого кодирования характеризуются использованием избыточной информации и усреднением шума. Закодированные помехоустойчивые сообщения всегда содержат дополнительные (избыточные) символы, поэтому помехоустойчивое кодирование также называют избыточным. Дополнительные символы применяют для того, чтобы уменьшить вероятность потери сообщением своей индивидуальности из-за возможных искажений. Эффект усреднения шума достигается за счет того, что избыточные символы зависят от нескольких значимых информационных символов.

В результате влияния помех в сообщении возникают однобитные или групповые ошибки. У групповых ошибок есть свои позитивные и негативные свойства. Положительное свойство групповой ошибки заключается в следующем. Пусть данные передаются блоками по 1000 бит, вероятность ошибки – 0,001 на бит. Если ошибки изолированы и независимы, то 63% блоков ($0,63 \approx 1 - 0,999^{1000}$) содержат ошибки. Если они возникают группами по 100, то ошибки содержат 1% блоков ($0,01 \approx 1 - 0,999^{10}$). Зато, если ошибки не группируются, то в каждом кадре они невелики, и есть возможность их исправить.

Негативное свойство групповых ошибок – безвозвратный урон, наносимый сообщению, в результате чего требуется повторная пересылка, что в некоторых системах невозможно (например, в телефонных сетях).

В помехоустойчивом кодировании вводится понятие расстояния Хемминга – минимального числа разрядов с неодинаковыми значениями кодовых слов. Например, последовательность двоичных кодов $\{0_{(10)}=00_{(2)}, 1_{(10)}=01_{(2)}, 2_{(10)}=10_{(2)}, 3_{(10)}=11_{(2)}\}$ имеет расстояние Хемминга, равное единице, поскольку два любых числа отличаются, по меньшей мере, на один бит. В таком коде ошибки невозможно обнаружить. Последовательность кодов $\{0_{(10)}=00_{(2)}, 1_{(10)}=11_{(2)}\}$ имеет расстояние Хемминга, равное двум, и позволяет обнаруживать одиночные ошибки. Комбинации $\{01_{(2)}, 10_{(2)}\}$ называются запрещенными. Если они встречаются декодером, то это служит признаком ошибки в переданном кодовом слове. Таким образом, за счет сокращения информационной емкости кодов становится возможным обнаруживать нарушения целостности передаваемой информации. Для исправления одного сбойного бита требуется увеличить расстояние Хемминга до трех. При этом информационная емкость таких кодов существенно падает, например, четырехразрядный код с расстоянием Хемминга, равным трем, способен вместить только два информационных символа.

Обнаружив ошибку, декодер последовательно сравнивает искаженный символ со всеми разрешенными символами, стремясь найти символ, наиболее схожий с искаженным (то есть символ с наименьшим числом различий, а именно не более, чем в $d-1$ позициях, где d – расстояние Хемминга). Например, 10-разрядный код $\{0000000000, 0000011111, 1111100000, 1111111111\}$ имеет расстояние Хемминга, равное пяти ($d = 5$). Он позволяет обнаруживать четырехбитные ошибки ($d-1 = 4$) и гарантированно исправлять двубитные ($(d-1)/2 = 2$). Если вместо символа 0000011111 получен символ 0001010111, то ближайший символ – 0000011111. Если вместо двух бит искажение затрагивает максимально допустимое для восстановления количество разрядов – четыре, то восстановление декодером не гарантируется. Например, если бы вместо 0000011111 декодер получил код 0111111111, то он восстановил бы символ 1111111111 по наименьшему расстоянию.

В общем случае, если количество сбойных бит меньше расстояния Хемминга хотя бы наполовину, то декодер может гарантированно восстановить исходный символ.

Условие обнаружения ошибок: $d \geq r$, где r – количество ошибок. Условие успешного восстановления информации: $d > 2r$. Поэтому простейшая реализации кодирования Хемминга дополнительно к информационным разрядам вводит $L = \log_2 K$ избыточных контролирующих разрядов, где K – число информационных разрядов. Число L округляется до ближайшего большего целого значения. L -разрядный контролирующий код есть инвертированный результат поразрядного сложения номеров тех информационных разрядов, значения которых равны 1. Например, имеется сообщение 100110. Число информационных разрядов $K = 6$. Следовательно, число избыточных битов $L = 3$. Номера единичных битов: 2, 3 и 6. Дополнительный код равен $2_{(10)} + 3_{(10)} + 6_{(10)} = 010_{(2)} + 011_{(2)} + 110_{(2)} = 111_{(2)}$, где "+" обозначает поразрядное сложение. После инвертирования бинарный код принимает вид 000. Декодер рассчитывает избыточный код и сравнивает с переданным значением. Затем он фиксирует код сравнения (поразрядная операция отрицания равнозначности), и, если результат отличен от нуля, то его значение – это номер ошибочно принятого разряда основного кода. Например, если принят код 100010, то рассчитанный декодером дополнительный код равен инверсии суммы $010_{(2)} + 110_{(2)} = 100_{(2)}$, то есть $011_{(2)}$, что означает ошибку в третьем разряде.

Теоретически допустимое количество исправляемых ошибок не ограничено, однако, практически информационная емкость кодовых слов уменьшается с ростом расстояния Хемминга. Поэтому приведенная схема кодирования усовершенствуется за счет того, что каждый контрольный бит отвечает за четность суммы некоторой сопоставленной ему группы битов, причем один бит может относиться к разным группам. Таким образом, один информационный бит влияет на несколько контрольных, и информационная емкость слова значительно возрастает.

Согласно методу Хемминга, для определения номеров позиций битов, которые контролируют информационный бит, стоящий в позиции k , необходимо разложить k по степеням двойки (табл. 12). С ростом разрядности кодового слова контрольные биты располагаются реже, поэтому с увеличением разрядности обрабатываемого слова эффективность кодов Хемминга значительно возрастает.

Таблица 12

Схема размещения информационных и контрольных бит

Номер бита	Разложение номера бита по степеням	Тип бита
1	2^0	Контрольный (контролирует биты 3, 5 и 7)
2	2^1	Контрольный (контролирует биты 3, 6 и 7)
3	$2^0+2^1 = 1+2$	Информационный (контролируется битами 1 и 2)
4	2^2	Контрольный (контролирует биты 5, 6 и 7)
5	$2^0+2^2 = 1+4$	Информационный (контролируется битами 1 и 4)
6	$2^1+2^2 = 2+4$	Информационный (контролируется битами 2 и 4)
7	$2^0+2^1+2^2 = 1+2+4$	Информационный (контролируется битами 1, 2 и 4)
8	2^3	Крайний бит в слове, не контролирует другие биты

Биты, номера которых являются степенью 2, становятся контрольными, остальные биты – информационными. Каждый контрольный бит отвечает за четность суммы связанный с ним группы бит, расположенных на определенных позициях справа. Если сумма соответствующих информационных битов четная, то контрольный бит равен нулю, и наоборот. Например, в слове 0100 отметим символом * позиции контрольных битов **0*100*. Бит 1, контролирующий биты 3, 5 и 7, равен единице, так как сумма соответствующих информационных битов нечетная (0+1+0). Бит 2, контролирующий биты 3, 6 и 7, равен нулю, так как сумма информационных битов четная (0+0+0). Бит 4, контролирующий биты 5, 6 и 7, равен единице, так как сумма информационных битов нечетная (1+0+0). Битом 8 можно пренебречь, так как он не несет информационной составляющей и ничего не контролирует. Итоговое значение – 1001100. Искажим, например, один бит 1011100. Бит 1 становится равным нулю, бит 2 – единице, бит 4 сохраняется. Контрольные биты 1 и 2 не совпадают с расчетными значениями. Сумма позиций этих битов дает позицию бита ошибки (1+2=3). Исправление ошибки достигается путем инверсии бита 3 в полученном коде.

Коды Хемминга способны исправлять лишь одиночные ошибки. С ростом размера обрабатываемых слов увеличивается и вероятность ошибок. Выбор оптимального размера кодируемых слов является нетривиальной задачей, требующей знания характера и частоты возникновения ошибок. Для исправления групповых ошибок применяются более сложные методы, например, кодирование Рида-Соломона.

Порядок выполнения работы

Перед выполнением лабораторной работы необходимо вручную рассчитать код Хемминга для произвольного 16-разрядного числа.

1. Создать текстовый файл размером 100 символов, сохранить его как эталон.
2. Реализовать программу, выполняющую для произвольного текстового файла функции кодирования и декодирования Хемминга. Размер текстового блока, обрабатываемого в рамках одного цикла кодирования, должен задаваться в виде аргумента командной строки.
3. Закодировать текстовый файл. Зафиксировать и сравнить размеры исходного и закодированного файлов для различных размеров кодируемых блоков: 8, 12, 16, 24, 32, 48, 64 разряда.
4. С помощью HEX-редактора (например, HIEW) в произвольных местах закодированного файла внести одиночные изменения и зафиксировать позицию в файле и исходное значение измененного бита, чтобы впоследствии можно было проверить данные при декодировании.
5. Запустить разработанную программу на декодирование модифицированного закодированного файла. Сопоставить содержимое декодированного файла с эталоном, отметить совпадения и расхождения. Проверить восстановление измененных битов при декодировании.
6. С помощью HEX-редактора в произвольных местах закодированного файла внести групповые изменения. Данные изменения моделируют массовые ошибки при передаче или хранении информации. Для внесенных изменений необходимо зафиксировать позицию в файле и новое значение измененных битов, чтобы впоследствии можно было проверить факт восстановления данных при декодировании файла. Повторить выполнение п. 5.

Содержание отчета

1. Пример расчета кода Хемминга для 16-разрядного числа.
2. Блок-схема алгоритма вычисления помехоустойчивого кода.
3. Описание однобитных и групповых ошибок, внесенных в файл.
4. Результаты восстановления информации.
5. График зависимости изменения объема закодированной информации от размера кодируемых блоков.
6. Листинг разработанной программы.
7. Ответы на контрольные вопросы.
8. Выводы по работе.

Контрольные вопросы

1. Что такое расстояние Хемминга?
2. Опишите метод кодирования Рида-Соломона.
3. Как контролировать ошибку в бите 8 (табл. 12)?
4. На что влияет размер кодируемого блока в методе Хемминга?
5. Постройте схему размещения информационных и контрольных бит в блоке размером 16 разрядов.

Лабораторная работа 11

МЕХАНИЗМ АУТЕНТИФИКАЦИИ ПОЛЬЗОВАТЕЛЕЙ

Цель работы – получение навыков защиты информационных систем путем проверки подлинности пользователей, исследование проблемы подмены механизма аутентификации в операционной системе Windows.

Теоретические сведения

В защищенной системе каждый пользователь должен быть известен и представлен в виде зарегистрированной системной сущности. В начале рабочего сеанса пользователь должен подтвердить свою идентичность системному представлению путем ввода своих учетных данных (логина и пароля). В этой связи проверка пользователя включает два этапа:

идентификацию – присвоение субъектам и объектам системы уникальных идентификаторов;

аутентификацию – проверку заявленного соответствия, то есть того, что пользователь является именно тем, за кого он себя выдает.

В такой системе возникают проблемы безопасности, связанные с тем, что в случае, когда некто посторонний получает доступ к логину и паролю, он получает доступ к информации, владеть которой не должен. Кроме того, если пользователь отходит от компьютера, не закончив сеанс работы, то другой пользователь может воспользоваться этим и совершить вредоносные действия, используя для этого чужую учетную запись. Данная угроза устраняется путем периодической аутентификации через определенные интервалы времени.

Основные проблемы безопасности при аутентификации:

1. Угадывание пароля.

Методы угадывания:

полный перебор – наиболее эффективный метод. От полного перебора не защищен ни один пароль, однако, для нахождения правильного пароля требуется значительное время;

интеллектуальный поиск – метод, позволяющий оптимизировать полный перебор за счет использования начальной информации о пользователе.

Способы противодействия:

изменение пароля, предлагаемого по умолчанию;

регулярное изменение пароля;

увеличение длины и сложности пароля. В пароле следует использовать как можно больше различных цифр, букв разных алфавитов и регистров, специальных символов;

использование утилит проверки и генерации паролей. Такие утилиты оценивают стойкость пароля к словарной атаке (перебору слов);

ограничение числа попыток ввода пароля. В этом случае программы перебора паролей становятся неэффективными.

2. Компрометация файла паролей.

Введенный пароль обычно сопоставляется с данными, которые хранятся в системном файле на диске. Если злоумышленнику удастся

получить доступ к этому файлу, то он сможет узнать пароли всех пользователей системы.

Способы противодействия:

использование хэш-функций – вычисление по паролю некоторого значения, которое является результатом вычисления необратимой функцией от пароля, и сравнение со значением, хранящимся в файле. При этом пароль нельзя получить, вычислив обратную функцию. Недостаток этого метода заключается в том, что можно подобрать такой пароль, что значение, вычисленное для него, будет совпадать с исходным;

контроль доступа – запрет на запись в файл паролей;

использование криптоставки – например, добавление некоторой константы к хэш-функции. Не зная ее, злоумышленник не сможет подобрать пароль методом перебора.

2. Подмена подсистемы идентификации/аутентификации.

Когда пользователь входит в систему, он вводит имя и пароль и попадает в так называемый *logon*-процесс. У пользователя нет возможностей определить, в какой программный модуль или процесс на самом деле передаются вводимые учетные данные. Между пользователем и *logon*-процессом злоумышленник может вставить "прослойку" – программу ложного логина (*fake login*). Эта программа должна активироваться при наступлении какого-либо события, например, входа в систему. Учетные данные, вводимые пользователем, попадают в программу ложного логина, которая их сохраняет, после чего, например, сообщает, что ввод неправильный и вызывает настоящий *logon*-процесс.

Способы противодействия:

система сообщает пользователю количество оставшихся попыток ввода пароля. Пользователь может определить, что введенные данные до системы не дошли, и предпринять защитные меры;

взаимная аутентификация – при выполнении каких-либо действий система должна называть себя, показывая, что пароль вводится верно.

Системный процесс *WinLogon*, отвечающий в операционной системе Windows за аутентификацию пользователей, имеет свой собственный рабочий стол – рабочий стол аутентификации. Никакой другой процесс, в том числе имитирующий его, не имеет доступа к рабочему столу аутентификации и не может расположить на нем свое окно. После запуска

операционной системы на экран компьютера выводится начальное окно рабочего стола аутентификации, содержащее указание нажать на клавиатуре комбинацию клавиш Ctrl+Alt+Del (механизм "Доверенного пути", *trusted path*, – пример взаимной аутентификации системы и пользователя). Сообщение о нажатии этих клавиш передается только системному процессу *WinLogon*, а для остальных процессов, в частности для всех прикладных программ, их нажатие происходит незаметно.

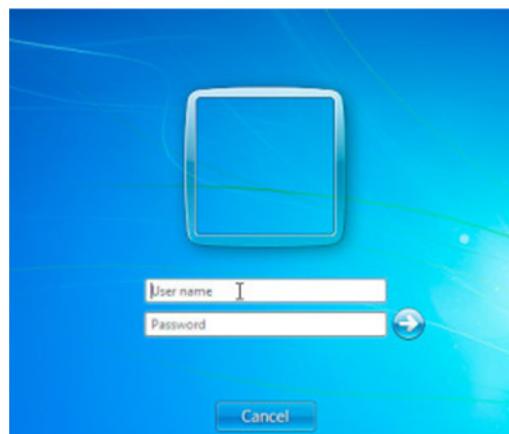


Рис. 10. Пример регистрационного окна процесса *WinLogon* в операционной системе Windows

Далее производится переключение на другое, так называемое регистрационное окно рабочего стола аутентификации (рис. 10). В нем размещается приглашение пользователю ввести свое имя и пароль, которые будут проверены процессом *WinLogon*.

Диалог-имитатор может попытаться воспроизвести не начальное окно рабочего стола аутентификации, а регистрационное окно-приглашение, где вводятся учетные данные пользователя. Однако по прошествии небольшого интервала времени и при отсутствии в системе программ-имитаторов регистрационное окно автоматически заменяется на начальное, если пользователь не предпринимает никаких попыток зарегистрироваться в системе.

Порядок выполнения работы

Во избежание потери информации и нарушения работоспособности используемых систем все действия следует выполнять в виртуальной среде.

1. Включить отображение *logon*-диалога при загрузке операционной системы Windows, для этого необходимо выполнить следующие действия:

в меню "Пуск" в строке поиска набрать и запустить оснастку *secpol.msc*;

в появившейся оснастке выбрать раздел "*Локальные политики/Параметры безопасности*";

в окне справа для параметра "*Интерактивный вход в систему: не отображать последнее имя пользователя*" установить значение "*Включен*";

Таким образом, после перезагрузки система будет выводить классическое окно ввода учетных данных.

2. Разработать программу, имитирующую *logon*-диалог в операционной системе Windows.

Программа-имитатор должна автоматически загружаться при входе пользователя в систему (например, из автозагрузки). Программа должна обновлять и проверять при запуске ключ в реестре, определенный разработчиком программы, так, чтобы запускаться через раз (в ином случае пользователь никогда не сможет зайти в систему). При запуске программа должна скрывать от пользователя интерфейс рабочего стола (например, для этого достаточно программно завершить процесс *explorer.exe*) и выводить сообщение об ошибке. После закрытия пользователем сообщения об ошибке программа должна выводить *logon*-диалог, функционально идентичный *logon*-диалогу Windows: ввод логина и пароля пользователя. Программа должна перехватывать и сохранять в файле, известном разработчику программы, введенные пользователем логин и пароль, выводить сообщение о том, что пользователем указаны неверные данные, вызывать функцию *ExitWindowsEx(EWX_LOGOFF, 0)* для завершения пользовательского сеанса.

3. Запустить разработанную программу-имитатор. Получить учетные данные, сохраненные программой.

4. Включить механизм доверенного пути в системе Windows. Для этого необходимо выполнить следующие действия:

в меню "Пуск" в строке поиска набрать и запустить оснастку *secpol.msc*;

в появившейся оснастке выбрать раздел "Локальные политики/Параметры безопасности";

в окне справа для параметра "Интерактивный вход в систему: не требовать нажатия CTRL+ALT+DEL" установить значение "Отключен".

Таким образом, после перезагрузки система будет выводить рабочий стол с приглашением пользователю нажать комбинацию клавиш *Ctrl+Alt+Del* для запуска механизма доверенного пути.

5. Снова перезагрузить операционную систему и выполнить вход в систему. Дождаться появления рабочего стола аутентификации, нажать комбинацию клавиш *Ctrl+Alt+Del* для запуска механизма доверенного пути и убедиться, что вход выполнен с помощью штатных средств операционной системы (например, удостоверившись в запуске задачи *explorer.exe* в приложении *Диспетчер задач*).

Содержание отчета

1. Листинг разработанной программы-имитатора.
2. Внешний вид окна запроса логина и пароля пользователя программы-имитатора, а также выводимого сообщения об ошибке.
3. Использованный ключ реестра для контроля запусков программы-имитатора.
4. Описание способа загрузки программы-имитатора и последовательности действий, выполняемых при запуске операционной системы для сохранения учетных данных пользователя.
5. Описание последовательности действий, выполняемых, при работе системы с включенным механизмом доверенного пути.
6. Ответы на контрольные вопросы.
7. Выводы по работе.

Контрольные вопросы

1. Что такое аутентификация? Чем она отличается от идентификации?
2. Какие основные угрозы направлены на систему аутентификации в операционной системе Windows?
3. Какие существуют основные методы противодействия угрозам системе аутентификации?

4. Опишите работу механизма доверенного пути в операционной системе Windows? В чем ее отличия от реализации механизма доверенного пути в Unix-системах?

5. Сколько вариантов необходимо перебрать, чтобы гарантированно определить пароль пользователя, состоящий из 6 цифр? Как изменится это число, если дополнительно использовать латинские буквы?

Лабораторная работа 12 **СИСТЕМА КОНТРОЛЯ ДОСТУПА**

Цель работы – освоение средств контроля и управления доступом пользователей к ресурсам операционной системы, приобретение навыков распределения прав на примере файловой системы NTFS в среде Windows.

Теоретические сведения

Подсистема контроля и управления доступом в операционной системе Windows отличается высокой степенью гибкости, которая достигается за счет разнообразия защищаемых субъектов и объектов доступа, а также дифференциации видов доступа.

Контроль доступа выполняется централизованно с помощью компонента операционной системы – *монитора безопасности* (Security Reference Monitor), работающего в привилегированном режиме. Унификация функций контроля доступа повышает эффективность защиты операционной системы.

В системе Windows реализована объектная модель контроля доступа, согласно которой все субъекты и объекты доступа зарегистрированы в системе и обращение к ним регулируется с помощью множества атрибутов безопасности, наиболее важными из которых являются права доступа.

Для подсистемы безопасности операционной системы Windows характерно наличие большого количества различных предопределенных (встроенных) субъектов доступа – учетных записей пользователей и групп. В системе имеются встроенные пользователи, например, *Администратор*

(Administrator), Гость (Guest), System и группы Пользователи (Users), Администраторы (Administrators), Все (Everyone). Встроенные пользователи и группы наделены полномочиями, заданными по умолчанию, что облегчает администрирование системы. Администратор также может создавать новые группы и новых пользователей, устанавливая для них права и принадлежность группам, реализуя тем самым политику информационной безопасности.

Права доступа – множество операций, которые определены для субъектов доступа (например, пользователей и групп) по отношению к объектам доступа (например, файлам, каталогам, принтерам, процессам). Примерами прав доступа являются чтение файла, удаление каталога, печать документа, изменение прав доступа.

Права доступа, установленные группе, автоматически предоставляются всем ее участникам (членам группы), позволяя администратору рассматривать множество пользователей как учетную единицу и минимизировать свои действия по управлению правами.

При входе пользователя в систему для него создается так называемый маркер доступа (access token), включающий идентификатор пользователя и идентификаторы всех групп, в которые он включен. В маркере также хранится список пользовательских привилегий, учитываемых при выполнении системных действий.

Всем идентифицируемым объектам доступа, включая файлы, потоки, процессы, объекты ядра, объекты синхронизации процессов, события и пр., при создании присваивается дескриптор (описатель) безопасности. Дескриптор безопасности содержит список контроля доступа (Access Control List, ACL) (рис. 11).

Список контроля доступа состоит из набора записей контроля доступа (Access Control Entry, ACE), каждая из которых указывает идентификатор субъекта доступа (пользователя или группы), тип записи и права доступа. ACE может быть либо разрешающей, либо запрещающей, определяя тип прав как разрешения или запреты в маске прав доступа.

Владелец объекта – обычно пользователь, который его создал – обладает правом избирательного управления доступом к объекту, всегда может изменять ACL объекта, чтобы разрешить или запретить другим пользователям доступ к объекту.

При запросе процессом доступа к объекту управление передается монитору безопасности, который сравнивает идентификаторы пользователя и групп пользователей из маркера доступа с идентификаторами, хранящимися в записях ACE из ACL объекта. Процесс в течение сеанса работы может осуществлять доступ ко многим объектам, а количество активных процессов и проверяемых ACE в каждый момент времени довольно большое, поэтому монитор безопасности проверяет возможность доступа процесса к объекту только при его открытии (получении идентификатора), а не при каждом обращении.

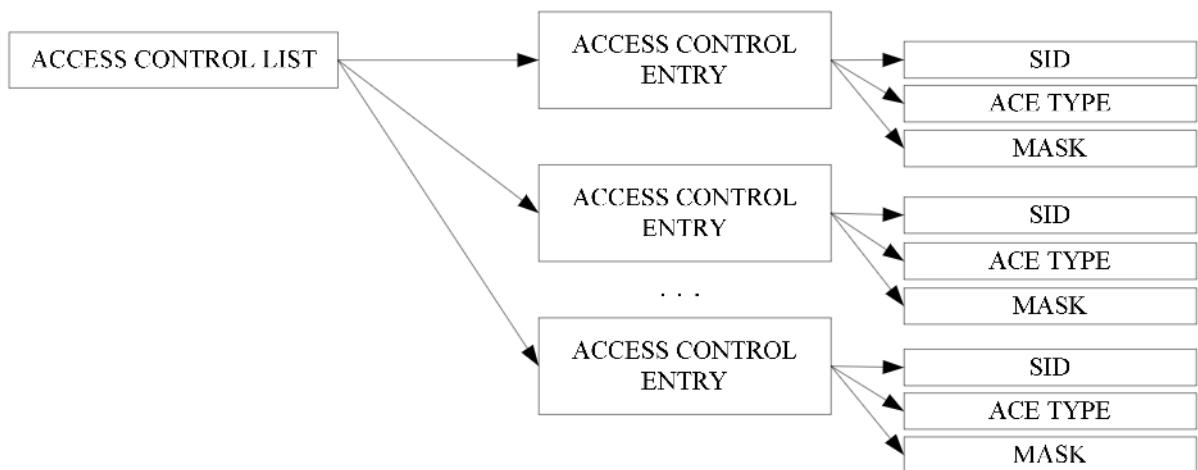


Рис. 11. Структура списка контроля доступа ACL

Управление доступом пользователей к файловым объектам в системе Windows возможно только в рамках логических разделов файловой системы NTFS. Доступ к каталогам и файлам контролируется с помощью прав, устанавливаемых для пользователей и групп в ACL соответствующего файлового объекта.

В рамках одного ACL могут быть заданы несколько ACE для одного пользователя или группы (например, разрешающая пользователю чтение, разрешающая группе запись и чтение и запрещающая пользователю удаление), которые могут быть противоречивы. Монитор безопасности путем свертки прав вычисляет множество действующих разрешений:

1. Из разрешающих ACE данного пользователя (группы) составляется множество разрешений.
2. Из запрещающих ACE данного пользователя (группы) составляется множество запретов.

3. Результатирующее множество разрешений определяется путем вычитания множества запретов из множества разрешений, поскольку запреты обладают большим приоритетом, чем разрешения.

Если для пользователя (группы) не установлено прав, то доступ запрещен. Если ACL не определен, то все виды доступа разрешены.

Для изменения ACL надо либо иметь право доступа "Изменение разрешений" (Change Permissions), либо быть владельцем объекта.

Для управления правами доступа к файловым объектам существует графический интерфейс операционной системы. Выбрав в контекстном меню файла или каталога пункт "Свойства", необходимо перейти на вкладку "Безопасность". На вкладке "Безопасность" в свойствах объекта представлен упрощенный интерфейс управления правами. При нажатии на кнопку "Дополнительно" появляется окно дополнительных параметров безопасности объекта, в котором можно изменить владельца объекта, тонко настроить ACL и наследование прав, рассчитать для конкретного субъекта доступа список действующих разрешений.

Порядок выполнения работы

Во избежание потери информации и нарушения работоспособности используемых систем все действия следует выполнять в виртуальной среде.

1. В разделе файловой системы NTFS создать каталог *test_folder*.
2. Для созданного каталога открыть окно дополнительных параметров безопасности. На вкладке "Разрешения" снять (если установлен) флаг наследования. В появившемся окне выбрать пункт "Удалить". Перейти на вкладку "Владелец" и убедиться, что текущий пользователь является владельцем этого каталога. Если это не так, то установить текущего пользователя владельцем каталога. Закрыть окна изменения параметров безопасности, сохранив внесенные изменения.
3. Проверить действующие разрешения, установленные для активного пользователя на каталог. Для этого необходимо перейти на вкладку "Действующие разрешения" в окне дополнительных параметров безопасности созданного каталога, нажать кнопку "Выбрать" и ввести текущего пользователя. Фактические разрешения, которыми наделен текущий пользователь, отмечены флагами. Зафиксировать данные права.

4. Продемонстрировать, что групповые права распространяются на всех членов группы. По умолчанию все пользователи находятся в группе "Пользователи", поэтому достаточно установить разрешения на группу "Пользователи". Для этого на вкладке "Безопасность" нажать кнопку "Изменить", в открывшемся окне нажать кнопку "Добавить", затем ввести наименование группы в поле ввода "Пользователи" и нажать кнопку "OK". Во вкладке "Безопасность" отметить флагами права чтения и записи в столбце "Разрешить" и нажать кнопку "Применить". Повторить выполнение п. 3.

5. Продемонстрировать приоритет запрещающих прав над разрешающими. Для этого запретить группе "Пользователи" права на запись, а текущему пользователю разрешить полный доступ. Повторить выполнение п. 3.

6. Продемонстрировать суммирование разрешающих прав для двух и более разрешающих ACE. Для этого в окне дополнительных настроек безопасности создать две разрешающие ACE с различными правами для одного пользователя. Повторить выполнение п. 3.

7. Разработать четыре тестовые утилиты, которые выполняют операции создания, чтения, записи и изменения атрибутов безопасности для файла. С помощью утилиты подтвердить на практике возможность доступа к файлу при разрешении и невозможность доступа – при запрете.

Содержание отчета

1. Описание структур ACL для объектов в виде списка элементов, представленных в формате ACE ("субъект – тип ACE – права доступа").
2. Расчет действующих разрешений для объектов на основе индивидуальных и групповых разрешений и запретов.
3. Произведенные модификации в структуре ACL, результаты проверок с помощью разработанных утилит и соответствующие выводы о влиянии изменений ACL на доступ к объекту.
4. Листинги тестовых утилит.
5. Ответы на контрольные вопросы.
6. Выводы по работе.

Контрольные вопросы

1. Что такое множество действующих разрешений?
2. Почему проверка прав доступа к файлу осуществляется только при открытии файла, а не при обращениях к нему?
3. Можно ли запретить администратору системы доступ к какому-либо файлу? Может ли он обойти это ограничение?
4. Как изменить владельца объекта в среде Windows?
5. Имеет ли владелец какие-либо права к файлу, если существует ACE, запрещающая полный доступ к этому файлу на имя владельца?

Лабораторная работа 13

ЗАЩИТА WEB-СЕРВЕРА

ОТ НЕСАНКЦИОНИРОВАННОГО ДОСТУПА

Цель работы – приобретение практических навыков по созданию безопасной рабочей конфигурации web-сервера Apache, исследование функций безопасности web-сервера и механизмов контроля и управления доступом к страницам сайта.

Теоретические сведения

Web-приложение – клиент-серверное приложение, в котором клиентом выступает web-браузер, в котором исполняется активный код или отображается web-контент, а сервером выступает web-сервер, на котором хранится набор программ и данных, передаваемых клиенту. Логика web-приложения реализуется на стороне сервера, клиент лишь формирует запросы и получает ответы. В этой связи безопасность web-сервера становится определяющей для безопасности всей системы в целом и web-приложений в частности.

Пример наиболее распространенной серверной web-платформы – свободно распространяемый, полнофункциональный, расширяемый web-сервер Apache. Web-сервер Apache настраивается путем изменения текстовых конфигурационных файлов. Основные параметры web-сервера

уже заданы по умолчанию. В самом простом случае после установки web-сервер Apache сразу же может использоваться для обработки поступающих web-запросов со стороны клиентских web-приложений к HTML-сайтам. Для обеспечения безопасности сайтов и разграничения доступа web-приложений требуется задействовать функции безопасности web-сервера, например, механизм авторизации запросов.

В операционных системах семейства UNIX обычно файлы создаются с правами `rw- --- ---`, то есть доступны пользователю-владельцу, который создал и запустил web-сервер. Только суперпользователь `root` и пользователь, от имени которого запущен web-сервер, имеют права на чтение этих файлов. Такая система не является действительно безопасной, так как от имени этого же пользователя могут быть выполнены скрипты, доступные по протоколу HTTP.

Если на сервере в скрипте имеется какая-либо уязвимость, позволяющая получать содержимое файлов по запросам web-приложений, то в этом случае атакующий сможет узнать учетные записи любого пользователя. Защититься от этого можно, ограничивая доступ на уровне сайта. Для этого настройки сохраняются в файлах `.htaccess` и `.htpasswd`, при этом используются функциональные модули – `mod_auth` и `mod_access`.

Если на одном сервере с установленной операционной системой семейства Unix и web-сервером Apache заведено несколько пользователей, то каждому из них можно создать отдельную директорию. Точнее, она будет создаваться автоматически вместе с созданием псевдонима пользователя. Это делается с помощью модуля `mod_userdir` и директивы `UserDir`.

Метод basic-аутентификации в web-серверах основывается на спецификации протокола HTTP. Аутентификация пользователей работает следующим образом: если пользователь не ввел пароль или пароль неверен, то сервер отвечает заголовком "401 Unauthorized". Если web-браузер получает от сервера такое сообщение, то браузер должен отобразить пользователю диалоговое окно с требованием ввести имя пользователя и пароль. Как только имя и пароль введены пользователем, web-браузер должен повторить запрос к серверу и направить серверу имя пользователя и пароль. Web-сервер повторно проверяет имя пользователя и пароль и затем либо разрешает доступ, либо вновь отвечает сообщением

"401 Unauthorized". В последнем случае можно полагать, что отправлены неверные сведения или имя. В течение всего сеанса браузер каждый раз передает имя и пароль для доступа ко всем документам, расположенным в том же web-каталоге или в других подкаталогах.

Рассмотрим пример диалога между клиентом и сервером при basic-аутентификации. Предположим, что пользователь обращается к странице сайта <http://www.site.ru/admin/>. Доступ к данной странице разрешен только по имени и паролю, поэтому web-сервер отвечает заголовком.

```
401 Требуется авторизация
Client> GET /admin/ HTTP/1.1 C> Host: www.site.ru
Client > User-Agent: Mozilla/5.0
Client > Accept: /*
Client > Accept-Language: en-us
Client > Accept-Encoding: gzip, deflate
Client > Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Client > Keep-Alive: 3000
Client > Connection: keep-alive
Server> HTTP/1.1 401 Authorization Required
Server > Server: Apache
Server > WWW-Authenticate: Basic realm="admin zone"
Server > Keep-Alive: timeout=15, max=50
Server > Connection: Keep-Alive
Server > Transfer-Encoding: chunked
Server > Content-Type: text/html; charset=iso-8859-1
Server > Извините, требуется авторизация.
Client > GET /admin/ HTTP/1.1
Client > Host: www.site.ru
Client > User-Agent: Mozilla/5.0
Client > Accept: /*
Client > Accept-Language: en-us
Client > Accept-Encoding: gzip, deflate
Client > Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Client > Keep-Alive: 3000
Client > Connection: keep-alive
Client > Authorization: Basic dXNlcjp1c2VycGFz
Server > HTTP/1.1 200 OK
Server > Date: Fri, 05 Nov 2004 13:31:54 GMT
Server > Server: Apache
Server > Keep-Alive: timeout=15, max=50
Server > Connection: Keep-Alive
Server > Transfer-Encoding: chunked
Server > Content-Type: text/html
Server > Спасибо, авторизация пройдена
```

В данном примере показан протокол аутентификации пользователя со стороны клиента. Имя пользователя и пароль зашифрованы алгоритмом base64 и отправляются в заголовке "Authorization" вместе с типом авторизации. Имя пользователя и пароль могут быть расшифрованы любыми средствами, умеющими работать в base64.

Рассмотрим работу системы аутентификации со стороны web-сервера. Реализовать basic-аутентификацию можно встроенными методами Apache, изменяя содержимое файла с именем *.htaccess*. Файл *.htaccess* содержится в защищаемом каталоге и имеет следующее содержимое:

```
order allow,deny  
allow from all  
require valid-user  
AuthName "Private Area"  
AuthType Basic  
AuthUserFile /path/to/.htpasswd
```

В файле */path/to/.htpasswd* находятся имена пользователей и хэши паролей. Для того чтобы создать файл с паролями, необходимо выполнить команду:

```
htpasswd -c /path/to/.htpasswd username
```

После чего дважды ввести пароль пользователя. Файл */path/to/.htpasswd* будет создан автоматически.

Для того чтобы добавить нового пользователя в существующий файл или изменить пароль любого существующего в файле пользователя, необходимо выполнить команду:

```
htpasswd /path/to/.htpasswd username
```

Для существующего пользователя будет изменен пароль, а если пользователь не существует, то пользователь и его пароль будут добавлены в файл. Файл */path/to/.htpasswd* не содержит паролей в открытом виде. Этот файл может выглядеть следующим образом:

```
admin:24nN.4cqs18hE
user1:zP0ggTOuNcxwc
user2:HfVn3BhVdnuiA
```

В примере указаны пароли для трех пользователей.

Остальные ключи утилиты *htpasswd* представлены в табл. 13. Дополнительную информацию можно получить, если использовать справочную команду *man htpasswd*.

Таблица 13

Настройки пароля

Ключ	Настройка
<i>-n</i>	<Имя пользователя: хэш пароля> будут выведен на экран
<i>-p</i>	Пароль хранится в текстовом виде без шифрования. Формат поддерживается только в операционных системах Windows, Netware, BEOS.
<i>-d</i>	Хэш пароля вычисляется с использованием стандартной UNIX-функции CRYPT. Алгоритм шифрования по умолчанию. Используется только на UNIX-серверах.
<i>-m</i>	Хэш пароля вычисляется по алгоритму MD5.
<i>-s</i>	Хэш пароля вычисляется по алгоритму SHA1.
<i>-D</i>	Удаление заданного пользователя из файла с паролями.

Защищаемый ресурс прописывается в конфигурационном файле *.htaccess*. В конфигурации необходимо указать тип аутентификации *AuthType*. В рассмотренном примере – тип *Basic*. Далее необходимо определить название для области требующей авторизации *AuthName*. Это название появится в диалоговом окне ввода логина и пароля. Затем задается параметр *AuthUserFile* – местонахождение файла с паролями, и *AuthGroupFile* – местонахождение файла групп. Затем определяются требования, необходимые для доступа к защищенной области *Require*. Таким образом, значение *valid-user* позволяет производить авторизацию для любого пользователя, прописанного в файле *.htpasswd*, например:

```
< Directory "/var/www/html/secret">
AuthType Basic
AuthName "Private Area"
AuthUserFile /var/www/html/site/.htpasswd
```

```
Require valid-user
< /Directory>
```

Преимущества basic-аутентификации:

простота и прозрачность реализации;
паролем защищены каталоги, а не файлы;
на сервере хранятся хэши паролей.

Недостаток basic-аутентификации: если у пользователя есть доступ к файлу с хэшами паролей, то он может попытаться подобрать пароль.

Порядок выполнения работы

Во избежание потери информации и нарушения работоспособности используемых систем все действия следует выполнять в виртуальной среде.

1. Установить на виртуальную машину web-сервер Apache, совместимый с версией операционной системы.
2. Изучить и описать содержание и значения параметров, указанных в файлах *.htaccess* и *.htpasswd* и созданных по умолчанию.
3. Реализовать basic-аутентификацию пользователей web-сервера Apache. Для этого необходимо выполнить действия:
 - 3.1. Создать файл с паролями пользователей *.htpasswd*.
 - 3.2. Создать пользователей *user1*, *user2*, *user3*, *user4*, *user5* с непустыми паролями.
 - 3.3. Создать на web-сервере каталоги *secret*, *not_secret*, *some_secret* и разместить их в каталоге */var/www/html/*.
 - 3.4. Разрешить доступ к каталогу *secret* только пользователю *user1*.
 - 3.5. Разрешить доступ к каталогу *not_secret* всем пользователям.
 - 3.6. Разрешить доступ к каталогу *some_secret* пользователям *user1*, *user3*, *user5*.
4. Настроить групповой доступ. Для этого необходимо выполнить действия:
 - 4.1. Создать файл *.htgroup* для работы с группами.
 - 4.2. В файле *.htgroup* прописать в строку *admins: user1 user2*.
 - 4.3. В файле *.htgroup* прописать в строку *users: user3 user4 user5*.
 - 4.4. Привести файл *.htaccess* к следующему виду:

```
AuthType Basic
AuthName "Private Area"
AuthUserFile /path/to/.htpasswd
AuthGroupFile /path/to/.htgroup
Require group admins
```

4.6. Проверить доступ к папке *secret*.

5. Разработать программу подбора пароля пользователя, зашифрованного способом base64. Требования к паролю следующие. Длина пароля не более четырех символов, разрешены буквы английского алфавита, цифры и знак подчеркивания. Программа должна находить пароль пользователя методом полного перебора.

6. Разработать программу, которая проверяет задаваемые пароли для пользователей web-сервера. Программа должна проверять выполнение следующих условий: длина пароля – не менее 10 символов, одновременное наличие в пароле разных символов в разном регистре, специальных символов и цифр. Программа в случае нарушения условий должна разрешать пользователю повторять ввод пароля. Разработанная программа должна быть встроена в алгоритм работы утилиты *htpasswd* и проверять вводимые пользователем пароли на соблюдение указанных условий. В случае если условия соблюdenы, программа разрешает создание нового пользователя web-сервера.

Содержание отчета

1. Содержимое файлов *.htpasswd* и *.htaccess* до изменения.
2. Список созданных каталогов и подкаталогов.
3. Содержимое файлов *.htpasswd* и *.htaccess* после изменения.
4. Листинг и описание заданных настроек прав доступа.
5. Результаты экспериментов по разрешению и запрету доступов для каждого из созданных пользователей к каждому созданному каталогу. Результаты представить в виде матрицы доступа, в которой перечислены пользователи и каталоги, а в ячейках указаны права доступа, подтвержденные экспериментально. Действия по проверке осуществления доступа должны быть описаны и подтверждены снимками экрана.
6. Листинги программы подбора и проверки пароля пользователя.

7. Изменения в кодах web-сервера Apache, произведенные с целью встраивания программы проверки паролей.

Контрольные вопросы

1. Какую роль web-сервер выполняет при работе с web-приложениями?

2. Перечислите отличия web-приложения от обычного приложения.

3. Какие конфигурационные файлы используются при работе механизма контроля доступа к web-страницам сайта, размещенного на web-сервере Apache?

4. Опишите процесс basic-аутентификации со стороны клиента.

5. Что позволяет нарушить безопасность web-сервера Apache при basic-аутентификации? Предложите и обоснуйте методы защиты от нарушений безопасности web-сервера Apache при basic-аутентификации.

Лабораторная работа 14

ЗАЩИТА ОТ УГРОЗ НАРУШЕНИЯ БЕЗОПАСНОСТИ ТИПА "ОТКАЗ В ОБСЛУЖИВАНИИ"

Цель работы – изучение механизма реализации компьютерной угрозы типа "отказ в обслуживании", ознакомление со способами защиты от такого рода угроз.

Теоретические сведения

Компьютерные угрозы типа "отказ в обслуживании" (*Denial-of-service, DoS*) – класс угроз, направленных на вычислительную систему, целью которых является препятствование возможности легального пользователя получить доступ к работающей системе из-за множества обращений, которые генерируется злоумышленником.

DoS-атака обычно является промежуточным этапом в процессе получения полного контроля над системой или ее вывода из нормального режима функционирования из-за перегрузок.

DoS-атаки возможны по нескольким причинам, связанным с ограниченностью вычислительных и информационных ресурсов: например, с ограниченной пропускной способностью каналов связи, ограниченной производительностью сетевого оборудования и серверов, которые не в состоянии обрабатывать огромное число одновременно поступающих запросов ("шквал запросов").

В настоящее время вычислительные мощности нападающей стороны практически не ограничены (например, используются бот-сети, высокопроизводительные вычислительные системы), поэтому мощность DoS-атак также значительно возросла: они стали распределенными и интеллектуальными.

Рассмотрим обобщенный сценарий реализации DoS-атаки.

Информация в сети передается пакетами. Структура пакета включает в себя адрес отправителя, адрес получателя, порт отправителя и порт получателя, а также другие служебные данные. Пакеты могут быть фрагментированы, то есть один пакет может быть разбит на несколько частей. Информация о фрагментации добавляется к служебной, чтобы компьютер получателя мог собрать фрагменты в пакет.

Наиболее распространены три типа сетевых пакетов:

TCP пакет – тип пакета, надежного при передаче, поскольку компьютер получателя отправляет уведомление о получении пакета. Если уведомление не получено, пакет посыпается повторно. В данном случае потеря информации исключается. Недостаток использования TCP-пакетов – замедление передачи по причине пересылки уведомлений;

UDP-пакет – тип пакета, не предусматривающего уведомлений о его получении, следовательно, допускающий потери информации;

ICMP-пакет – тип пакета, предназначенного для служебных задач (например, для диагностики сети, выполнения *ping*) и не используемого для передачи информации.

DoS-атаки основаны на идее флуда (*flood*), то есть "затопления" жертвы огромным количеством пакетов.

Рассмотрим пример флуда с помощью TCP-пакета. Перед началом обмена данными протокол TCP устанавливает соединение методом трехэтапного подтверждения. Флаг SYN обозначает начало запроса на установку соединения. Флаг ACK обозначает подтверждение о получении

пакета. Флаг FIN предназначен для разрыва соединения после завершения передачи. Метод трехэтапного подтверждения установки соединения заключается в простом обмене пакетами (рис. 12).

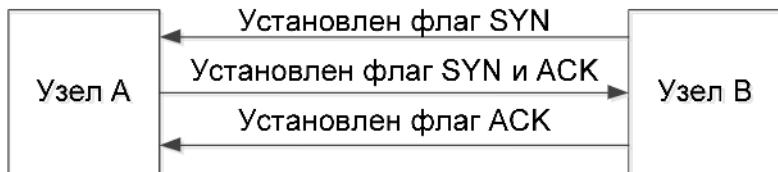


Рис. 12. Установка сетевого соединения по протоколу TCP

Рассмотрим последовательность событий:

1. Узел А посыпает пакет узлу В, установив в нем значение бита SYN равным 1. Поле *Sequence Number* содержит начальное значение.

2. Узел В создает структуры данных для установки соединения. Узел В отвечает на запрос соединения, направляя узлу А пакет с флагом ACK, равным 1, и тем самым подтверждая прием пакета от узла А. Флаг SYN в этом втором пакете также равен 1, таким образом узел А знает, что поле *Sequence Number* в пакете содержит начальное значение для узла В.

3. После получения подтверждения от узла В узел А подтверждает прием начального значения от узла В. При этом пересыпается уже содержательный пакет с установленным флагом ACK.

В нормальных условиях соединение успешно устанавливается, и узлы могут продолжать обмен данными между собой.

Предположим, что в первом пакете с запросом на установку соединения, пришедшим от узла А, указан неверный IP-адрес. В таком случае подтверждение от узла В не вернется к узлу А. Узел В, не дождавшись ответа в заданный временной промежуток, должен освободить память, выделяемую для поддержания соединения. Указанная особенность используется для DoS-атаки *SYN-flood*.

При проведении *SYN-flood*, атакующий компьютер непрерывно посыпает сообщение с запросами на установку соединения. В пакете указывается несуществующий IP-адрес. Атакуемый узел создает новые динамические структуры данных и запускает таймер для каждой новой попытки соединения до тех пор, пока не исчерпает свои ресурсы. После этого атакуемый компьютер перестает отвечать на попытки подключения.

Проведение *SYN-flood* выявляется, путем подсчета числа "полуоткрытых" TCP-соединений, например:

```
# netstat -na | grep ":80 " | grep SYN_RECV
```

При нормальном состоянии системы "полуоткрытых" соединений быть не должно, либо их число не должно превышать 1-3.

Принцип атаки *ICMP-flood* (или *Ping Flood*) схож с *SYN-flood*, но атакуемый компьютер "забрасывается" пакетами типа ICMP. Поскольку система должна ответить на такой пакет, это приводит к созданию большого количества ответных пакетов, и, как следствие, к снижению пропускной способности канала.

С помощью протокола ICMP злоумышленник в состоянии изменить настройки сети, например, таблицы маршрутизации. В этом случае возникает эффект недоступности узла, хотя на самом деле не так. Поскольку соответствующая запись в таблице маршрутизации является неверной, сетевой трафик до узла не дойдет.

Атака *HTTP-flood* организована по аналогии с рассмотренными атаками. Целью атаки является web-сервер. При подозрении на *HTTP-flood* прежде всего следует подсчитать количество процессов web-сервера (например, для web-сервера Apache) и количество соединений на порт 80:

```
# ps aux | grep httpd | wc -l  
# netstat -na | grep ":80 " | wc -l
```

Если значения в несколько раз превышают среднестатистические, то есть основания подозревать *HTTP-flood*. Далее следует просмотреть список IP-адресов, с которых идут запросы на подключение:

```
# netstat -na | grep ":80 " | sort |  
        uniq -c | sort -nr | less
```

Дополнительным подтверждением атаки может стать результат анализа пакетов с помощью утилиты *tcpdump*:

```
# tcpdump -n -i eth0 -s 0 -w output.txt  
        dst port 80 and host
```

Показателем также может служить большой поток однообразных пакетов с различных IP-адресов, направленных на один порт или сервис (например, на определенный исполняемый скрипт).

Проведение *UDP-flood* организовано следующим образом. На порт 7 атакуемого компьютера по широковещательному запросу направляются *echo*-команды. Затем IP-адрес злоумышленника подменяется на IP-адрес атакуемого. В результате этого атакуемый компьютер получает множество ответных сообщений. Число таких сообщений зависит от числа узлов в сети. *UDP-flood* приводит к исчерпанию полосы пропускания и затем к полному отказу в обслуживании атакуемого компьютера. Если служба *echo* отключена, то будут сгенерированы ICMP-пакеты, что также вызывает отказ в обслуживании.

Рассмотрим некоторые способы защиты от рассмотренных примеров DoS-атак. Защита от *SYN-flood* строится на отключении очереди "полуоткрытых" TCP-соединений:

```
# sysctl -w net.ipv4.tcp_max_syn_backlog=1024
```

Ограничение максимального числа "полуоткрытых" соединений с одного IP-адреса к конкретному порту:

```
# iptables -I INPUT -p tcp --syn --dport
           80 -m iplimit --iplimit-above 10 -j DROP
```

Для того, чтобы защититься от *ICMP-flood* необходимо отключать ответы на запросы ICMP *echo*:

```
# sysctl net.ipv4.icmp_echo_ignore_all=1
```

С помощью правил межсетевого экрана это можно реализовать следующим образом:

```
# iptables -A INPUT -p icmp -j DROP --icmp-type 8
```

Для защиты от *HTTP-flood* следует увеличить одновременное количество максимальных подключений к базе данных сервера, установив

перед web-сервером Apache более производительный web-сервер *Nginx* для кэширования запросов.

Защита от *UDP-flood* заключается в том, чтобы отключить UDP-сервисы и установить ограничение на число соединений с DNS-сервером:

```
# iptables -I INPUT -p udp --dport 53  
          -j DROP -m iplimit --iplimit-above 1
```

Кроме того, следует руководствоваться следующими общими рекомендациями по защите от DoS-атак:

все серверы, имеющие доступ во внешнюю сеть, должны быть подготовлены к удаленной аварийной перезагрузке. Более того, желательно наличие второго сетевого интерфейса, через который по SSH-соединению можно получить доступ к серверу;

программное обеспечение сервера всегда должно быть обновленным;

организовать фильтрацию и блокирование трафика, исходящего от атакующих машин. Фильтрация может быть двух видов: использование межсетевых экранов и списков контроля доступа ACL. Использование межсетевых экранов позволяет блокировать конкретный поток трафика, но не позволяют отделить полезный трафик от флуда. ACL-списки фильтруют второстепенные протоколы и не затрагивают протокол TCP.

Порядок выполнения работы

Во избежание потери информации и нарушения работоспособности используемых систем все действия следует выполнять в виртуальной среде.

1. Изучить интерфейс и основные функции генераторов пакетов *Nemesis*, *Colasoft Packet Builder*, *PACKETH*, *ostinato*. Сравнить и описать возможности данных программных продуктов.

2. Смоделировать *SYN-flood* на виртуальный сервер. Описать механизм смоделированной атаки и оценить количество запросов, время между запросами, скорость поступления запросов.

3. Смоделировать атаку *ICMP-flood* на виртуальный сервер. Описать механизм смоделированной атаки и оценить количество запросов, время между запросами, скорость поступления запросов.

4. Организовать элементы защиты от *SYN-flood* и *ICMP-flood*:
отключить очередь "полуоткрытых" TCP-соединений командой

```
#sysctl -w net.ipv4.tcp_max_syn_backlog=1024
```

отключить ответы на запросы ICMP *echo* командой

```
#sysctl net.ipv4.icmp_echo_ignore_all=1
```

повторить выполнение пп. 2 и 3 и оценить новые результаты моделирования атак, сравнив их с ранее полученными результатами.

Содержание отчета

1. Описание возможностей исследованных генераторов пакетов.
2. Описание механизмов атак *SYN-flood* и *ICMP-flood* на виртуальный сервер.
3. Пакеты, генерируемые каждым генератором пакетов.
4. Сравнение результатов проведенных атак с помощью каждой программы.
5. Сопоставление результатов экспериментов, полученных без и с использованием элементов защиты от *SYN-flood* и *ICMP-flood*.
6. Ответы на контрольные вопросы.

Контрольные вопросы

1. В чем заключается общий принцип проведения DoS-атаки?
2. Какие способы защиты от DoS-атак существуют?
3. В чем заключается работа генератора пакетов? Какие настройки можно установить при генерации пакетов?
4. Какова структура UDP-пакета? Что делает возможным проведение DoS-атаки на уровне UDP-трафика?
5. За счет чего отключение ответов на запросы ICMP позволяет решить проблему *ICMP-flood*?

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. **Аблязов Р. З.** Программирование на Ассемблере на платформе x86-64 / Р. З. Аблязов . – М.: ДМК-Пресс, 2011.
2. **Ватолин Д.** Методы сжатия данных. Устройство архиваторов, сжатие изображений и видео / Д. Ватолин, А. Ратушняк, М. Смирнов, В. Юкин . – М. : Диалог-МИФИ, 2003.
3. **Зегжда П. Д.** Основы информационной безопасности : Учеб. пособие / П. Д. Зегжда, Е. А. Рудина . – СПб. : Изд-во Политехн. ун-та, 2008.
4. **Зубков С. В.** Assembler. Для DOS, Windows и Unix / С.В. Зубков . – М.: ДМК-Пресс, 2017.
5. **Касперски, К.** Компьютерные вирусы изнутри и снаружи / К. Касперски . – М. [и др.] : Питер, 2007.
6. **Касперски, К.** Искусство дизассемблирования : наиболее полное руководство / К. Касперски, Е. Рокко . – СПб. : БХВ-Петербург, 2008.
7. **Ломов А. Ю.** Apache, Perl, MySQL: практика создания динамических сайтов : самоучитель / А. Ломов . – СПб. : БХВ-Петербург, 2007.
8. **Миано Дж.** Форматы и алгоритмы сжатия изображений в действии. Учеб. пособие / Дж. Миано – М. : Изд-во Триумф, 2003.
9. **Никифоров С. Н.** Методы защиты информации. Шифрование данных / С.Н. Никифоров . – СПб. : Лань, 2018.
10. **Новиков Ф. А.** Дискретная математика : учебник для бакалавров и магистров. Стандарт третьего поколения / Ф. А. Новиков . – СПб : Питер, 2017.
11. **Побегайло А. П.** Системное программирование в Windows / А.П. Побегайло . – СПб. : БХВ-Петербург, 2006.
12. **Романовский И. В.** Дискретный анализ / И. В. Романовский . – СПб. : БХВ-Петербург, 2016.
13. **Руссинович М.** Внутренне устройство Windows / М. Руссинович, Д. Соломон, А. Ионеску, П. Йосифович . – СПб. : Питер, 2018.
14. **Сергеенко В.С.** Сжатие данных, речи, звука и изображений в телекоммуникационных системах / В.С. Сергеенко, В. В. Баринов . – М. : РадиоСофт, 2011.
15. **Фостер Дж.** Защита от взлома: сокеты, эксплойты, shell-код / Джеймс С. Фостер . – М. : ДМК-Пресс, 2006.

ПРИЛОЖЕНИЕ

Описание генераторов компьютерных вирусов

Во избежание потери информации и нарушения работоспособности систем все действия следует выполнять в виртуальной среде.

Генератор Raptor Virus Generator

Вкладка "Оформление" (рис. 13) открывает меню выбора иконки EXE-файла вируса, который будет сгенерирован.

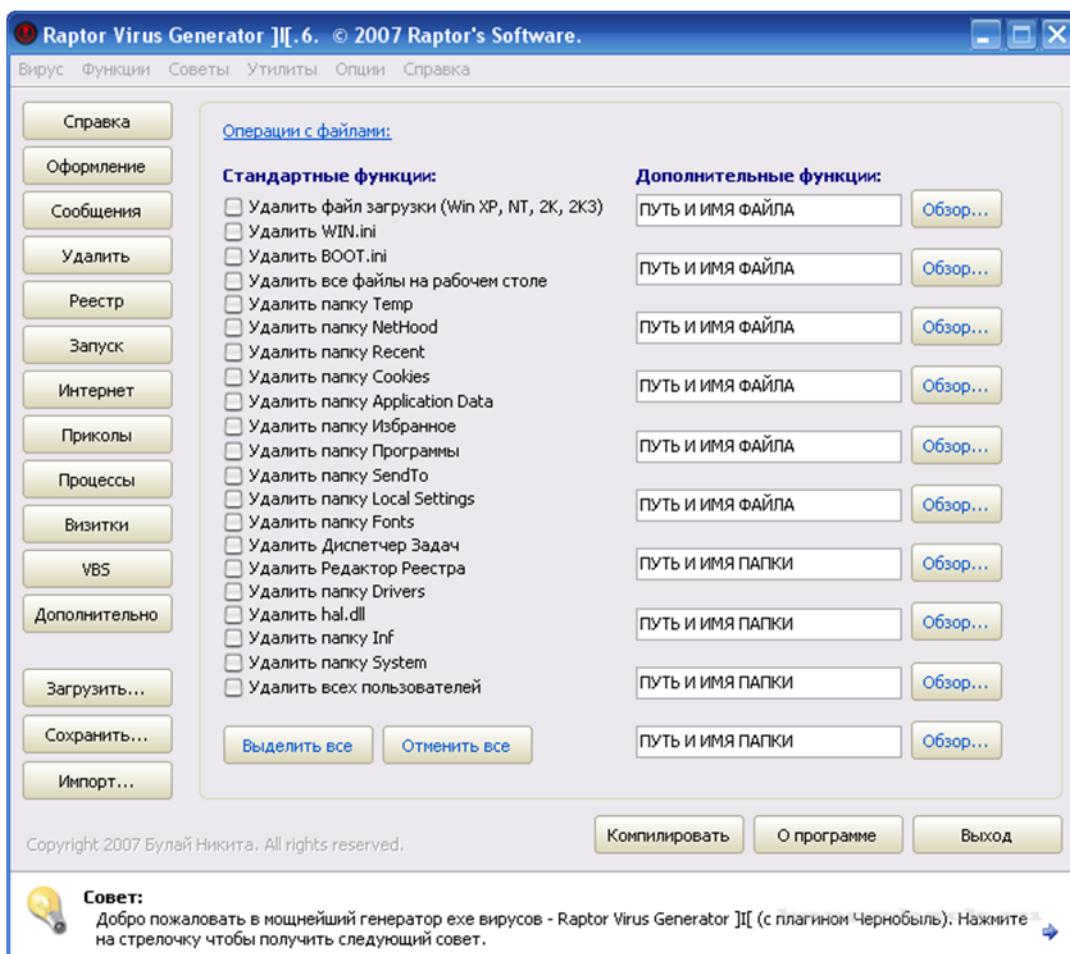


Рис. 13. Вид главного окна генератора Raptor Virus Generator

Вкладка "Сообщение" позволяет настроить функции отображения на экран пользователя сообщения при запуске зараженного файла или при завершении работы зараженного файла.

Вкладка "Удалить" позволяет настроить функции, выполняемые вирусом с файлами – удаление, копирование, перемещение.

Вкладка "Реестр" позволяет настроить функции, выполняемые вирусом с реестром на зараженной машине – создание ключа, удаление ключа, изменение значений стандартных ключей.

Вкладка "Запуск" позволяет добавить вирусу функцию запуска исполняемых файлов на зараженной машине.

Вкладка "Интернет" позволяет добавить вирусу функцию скачивания файлов и, открытию страниц в браузере

Дополнительные деструктивные действия вирусу можно добавить на вкладке "Дополнительно".

Кнопка "Компилировать" запустит процесс создания вируса.

Генератор GVDG

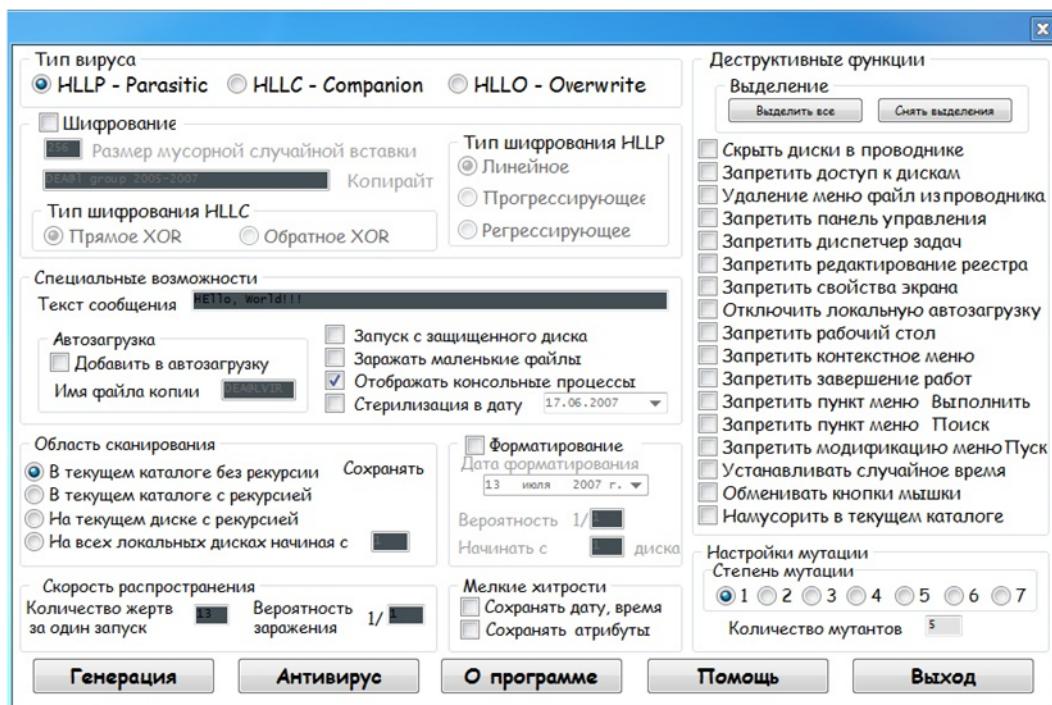


Рис. 14. Вид главного окна генератора GVDG

Область окна "Тип вируса" (рис. 14) позволяет выбрать способ заражения вирусом. При выборе типа вируса происходит изменение доступности по редактированию всех остальных элементов управления. Например, для типа *HLLP* опция "Шифрование" доступна, а для остальных – нет. Таким образом, работу с генератором в каждом новом цикле создания вируса следует начинать именно с этой опции.

Тип *HLLP* – вирус-паразит (тело вируса будет дописано к исполняемому файлу, без повреждения содержимого).

Тип *HLLC* – вирус-компаньон (вирус при заражении переименует исходный файл и запишет себя с его исходным именем).

Тип *HLLO* – перезаписывающий вирус (вирус при заражении затрет содержимое заражаемого файла).

Область окна "*Шифрование*" позволяет включить автоматическое шифрование тела вируса при заражении, что позволяет защитить вирус от обнаружения антивирусными средствами.

Область окна "*Специальные возможности*" позволяет добавить в вирусы функции по работе с файловой системой зараженной машины, а также автоматизировать запуск вируса путем добавления его в автозагрузку.

Область окна "*Область сканирования*" определяет, в каких каталогах файловой системы вирус будет искать исполняемые файлы для заражения.

Область окна "*Деструктивные действия*" позволяет добавить вирусу дополнительные функции.

Кнопка "*Генерация*" запускает процесс создания вируса.

Кнопка "*Антивирус*" запускает встроенный в программу антивирус.

*Зегжда Петр Дмитриевич
Калинин Максим Олегович*

**ОСНОВЫ
ИНФОРМАЦИОННОЙ БЕЗОПАСНОСТИ
ВВЕДЕНИЕ В ПРОФЕССИОНАЛЬНУЮ
ДЕЯТЕЛЬНОСТЬ
ЛАБОРАТОРНЫЙ ПРАКТИКУМ**

Учебное пособие

Налоговая льгота – Общероссийский классификатор продукции
ОК 005-93, т. 2; 95 3005 – учебная литература

Подписано в печать 23.01.2019. Формат 60×84/16. Печать офсетная.

Усл. печ. л. 7,0. Тираж 200. Заказ 12.

Отпечатано с готового оригинал-макета, предоставленного авторами,
в Издательско-полиграфическом центре Политехнического университета.
195251, Санкт-Петербург, Политехническая ул., 29.
Тел.: (812) 552-77-17; 550-40-14.