Министерство науки и высшего образования РФ
ФГБОУ ВО «Ульяновский государственный университет»
Факультет математики, информационных и авиационных технологий

Сутыркина Е.А.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ ПО ДИСЦИПЛИНЕ

«Вредоносные программы в компьютерных сетях»

Ульяновск, 2021

Методические указания к лабораторным работам по дисциплине «Вредоносные программы в компьютерных сетях» / составитель: Е.А.Сутыркина. - Ульяновск: УлГУ, 2021. Настоящие методические указания предназначены для студентов специалитета по специальностям 10.05.01 и 10.05.03 очной формы обучения. В работе приведены литература по дисциплине, методические указания для самостоятельной работы студентов. Они будут полезны при подготовке к лабораторным работам и к экзамену по данной дисциплине.

Методические указания рекомендованы к введению в образовательный процесс решением Ученого Совета ФМИиАТ УлГУ (протокол 4/21 от 18 мая 2021г.)

Оглавление

1. ЛИТЕРАТУРА ДЛЯ ИЗУЧЕНИЯ ДИСЦИПЛИНЫ

1. Щербаков А.Ю., А.Ю. Щербаков. Современная компьютерная безопасность. Теоретические основы. Практические аспекты. Учебное пособие. - М.: Книжный мир, 2009. - 352 с. - ISBN 978-5-8041-0378-2 - Режим доступа: http://www.studentlibrary.ru/book/ISBN9785804103782.html
2. Внуков, А. А. Защита информации : учебное пособие для бакалавриата и магистратуры / А. А. Внуков. — 2-е изд., испр. и доп. — Москва : Издательство Юрайт, 2019. — 240 с. — (Высшее образование). — ISBN 978-5-534-01678-9. — Текст : электронный // ЭБС Юрайт [сайт]. — URL: https://www.biblio-online.ru/bcode/444046
3. Защита информации : учеб. пособие для студентов вузов по направлению подготовки "Инфокоммуникационные технологии и системы связи" / А. П. Жук [и др.]. - 3-е изд. - Москва : РИОР : Инфра-М, 2018.
4. Ботуз С.П., Управление удаленным доступом / Защита интеллектуальной собственности в сети Internet [Электронный ресурс] / Ботуз С.П. - М. : СОЛОН-ПРЕСС, 2008. - 256 с. - ISBN 5-98003-289-4 - Режим доступа: http://www.studentlibrary.ru/book/ISBN5980032894.html
5. Борисов А.Б., Комментарий к гражданскому кодексу российской федерации части четвертой (постатейный). Правовое регулирование отношений в сфере интеллектуальной собственности. С постатейными материалами и практическими разъяснениями. Автор комментариев и составитель - А.Б. Борисов - м.: книжный мир, 2007. - 288 с. - isbn 978-5-8041-0286-0 - режим доступа: http://www.studentlibrary.ru/book/isbn9785804102860.html
6. Аверченков, В. И. Защита персональных данных в организации : монография / В. И. Аверченков, М. Ю. Рытов, Т. Р. Гайнулин. — Брянск : Брянский государственный технический университет, 2012. — 124 с. — ISBN 5-89838-382-4. — Текст : электронный // Электронно-библиотечная система IPR BOOKS : [сайт]. — URL: http://www.iprbookshop.ru/6993.html
7. Аминаров А. В. Лабораторный практикум по математическим методам защиты информации : учеб.-метод. указания для спец. "Компьютерная безопасность, "Математическое обеспечение и администрирование информационных систем" / А. В. Аминаров, А. М. Иванцов, С. М. Рацеев. Ульяновск : УлГУ, 2016. 55 с. - **URL:** ftp://10.2.96.134/Text/Amiranov_2016.pdf

2. МЕТОДИЧЕСКИЕ УКАЗАНИЯ

Ниже приведены методические указания по самостоятельному выполнению лабораторных работ.

Лабораторная работа №1. Повторение. Основы работы с ассемблером

Рассмотрим пример создания простой программы на языке ассемблера для процессоров семейства x86, с разбора которой можно начать свой путь в покорение низин уровней абстракции.

Вызов подпрограмм

Потребность вызывать подпрограммы влечет за собой несколько тем для изучения: организация подпрограмм, передача аргументов, создание стекового кадра, работа с локальными переменными.

Подпрограммы представляют собой метку, по которой располагается код. Заканчивается подпрограмма инструкцией ret. К примеру, вот такая подпрограмма в DOS выводит в консоль строку «Hello, world»:

```
print_hello:
    mov ah, 0x9
    mov dx, hello
    int 0x21
    ret
```

Для ее вызова нужно было бы использовать инструкцию call:

```
call print_hello
```

Будем передавать аргументы подпрограммам через регистры и указывать в комментариях, в каких регистрах какие аргументы должны быть, но в языках высокого уровня аргументы передаются через стек. К примеру, вот так вызывается функция printf из библиотеки C:

```
push hello
call _printf
add esp, 4
```

Аргументы передаются справа налево, обязанность по очистке стека лежит на вызывающей стороне. При входе в подпрограмму необходимо создать новый стековый кадр. Делается это следующим образом:

```
print_hello:
    push ebp ;сохраняем указатель начала стекового кадра на стеке
    mov ebp, esp ;теперь началом кадра является вершина предыдущего
```

Соответственно, перед выходом нужно восстановить прежнее состояние стека:

```
mov esp, ebp
pop ebp
ret
```

Для локальных переменных используется стек, на котором после создания нового кадра выделяется нужное количество байт:

```
print_hello:
    push ebp
    mov ebp, esp
    sub esp, 8 ;опускаем указатель вершины стека на 8 байт, чтобы выделить память
```

Архитектура x86 предоставляет специальные инструкции, с помощью которых можно более лаконично реализовать эти действия:

```
print_hello:
    enter 8, 0 ;создать новый кадр, выделить 8 байт для локальных переменных

    leave ;восстановить стек
    ret
```

Второй параметр инструкции enter – уровень вложенности подпрограммы. Он нужен для линковки с языками высокого уровня, поддерживающими такую методику организации подпрограмм. В нашем случае это значение можно оставить нулевым.

Лабораторная работа №2. Анализ программных реализаций

Тестирование черного ящика

определяется как методика тестирования, при которой функциональность тестируемого приложения (AUT) тестируется без учета внутренней структуры кода, деталей реализации и знания внутренних путей программного обеспечения. Этот тип тестирования полностью основан на требованиях и спецификациях программного обеспечения. В BlackBox Testing мы просто фокусируемся на входах и выходах программной системы, не заботясь о внутренних знаниях программ.

Общие шаги, которые необходимо выполнить для проведения любого типа тестирования черного ящика:

- Сначала рассматриваются требования и спецификации системы.
- Тестер выбирает допустимые входные данные (положительный сценарий тестирования), чтобы проверить, правильно ли их обрабатывает SUT. Кроме того, некоторые недействительные входные данные (сценарий отрицательного тестирования) выбираются для проверки того, что SUT может их обнаружить.
- Тестер определяет ожидаемые результаты для всех этих входов.
- Тестировщик программного обеспечения создает тестовые наборы с выбранными входами.
- Тестовые случаи выполнены.
- Тестер программного обеспечения сравнивает фактические результаты с ожидаемыми результатами.

- Дефекты, если таковые имеются, исправлены и перепроверены.

Пусть необходимо выполнить тестирование программы, определяющей точку пересечения двух прямых на плоскости. Попутно, она должна определять параллельность прямой одной их осей координат.

В основе программы лежит решение системы линейных уравнений:

$$Ax + By = C \text{ и } Dx + Ey = F.$$

А) Используя **метод эквивалентных разбиений,**

получаем для всех коэффициентов один правильный класс эквивалентности (коэффициент - вещественное число) и один неправильный (коэффициент - не вещественное число). Откуда можно предложить 7 тестов:

1) все коэффициенты - вещественные числа;
2)- 7) поочередно каждый из коэффициентов - не вещественное число.

Б) По **методу граничных условий**:

можно считать, что для исходных данных граничные условия отсутствуют (коэффициенты - «любые» вещественные числа);

для результатов - получаем, что возможны варианты: единственное решение, прямые сливаются (множество решений), прямые параллельны (отсутствие решений). Следовательно, можно предложить тесты,

с результатами внутри области:

1) единственное решение
2) множество решений
3) отсутствие решений

и с результатами на границе:

1) $\delta = 0{,}01$;
2) $\delta = -0{,}01$;
3) $\delta = \delta_x = 0{,}01$, $\delta_y = 0$;
4) $\delta = 0$, δ_y -0,01, $\delta_x = 0$.

В) **По методу анализа причинно-следственных связей**:

Определяем множество условий.

для определения типа прямой:

1) $\begin{vmatrix} a = 0 \\ b = 0 \\ c = 0 \\ d = 0 \\ e = 0 \end{vmatrix}$ - для определения типа и существования первой прямой;
2) $f = 0$ - для определения типа и существования второй прямой;

для определения точки пересечения:

1) $\delta = 0$
2) $\delta_x = 0$
3) $\delta_y = 0$

Выделяем три группы причинно-следственных связей (определение типа и существования первой линии, определение типа и существования второй линии, определение точки пересечения) и строим таблицы истинности.

A=0	B=0	C=0	Результат
0	0	X	прямая общего положения
0	1	0	прямая, параллельная оси OX
0	1	1	ось OX
1	0	0	прямая, параллельная оси OУ
1	0	1	ось OУ
1	1	X	множество точек плоскости

Такая же таблица строится для второй прямой.

$\delta = 0$	$\delta_x = 0$	$\delta_y = 0$	**Ед. реш.**	**Мн.реш.**	**Реш. нет**
0	X	X	1	0	0
1	0	X	0	0	1
1	X	0	0	0	1
1	1	1	0	1	0

Каждая строка этих таблиц преобразуется в тест. При возможности (с учетом независимости групп) берутся данные, соответствующие строкам сразу двух или всех трех таблиц.

В результате к уже имеющимся тестам добавляются: проверки всех случаев расположения обеих прямых - 6 тестов по первой прямой вкладываются в 6 тестов по второй прямой так, чтобы варианты не совпадали, - 6 тестов; выполняется отдельная проверка несовпадения условия $_x\delta = 0$ или $_y = 0$ (в зависимости от того, какой тест был выбран по методу граничных условий) - тест также можно совместить с предыдущими 6 тестами. По методу предположения об ошибке добавим тест: все коэффициенты - нули.

Всего получили 20 тестов по всем четырем методикам.

Тестирование белого ящика

Рассмотрим следующий фрагмент кода

```
Printme (int a, int b) {                    ------------  Printme is a function
    int result = a+ b;
    If (result> 0)
        Print ("Positive", result)
    Else
        Print ("Negative", result)
    }                                      -----------   End of the source code
```

Целью тестирования WhiteBox является проверка всех ветвей решений, циклов, операторов в коде.

Чтобы выполнить утверждения в приведенном выше коде, тестовые случаи WhiteBox будут:

- A = 1, B = 1
- A = -1, B = -3

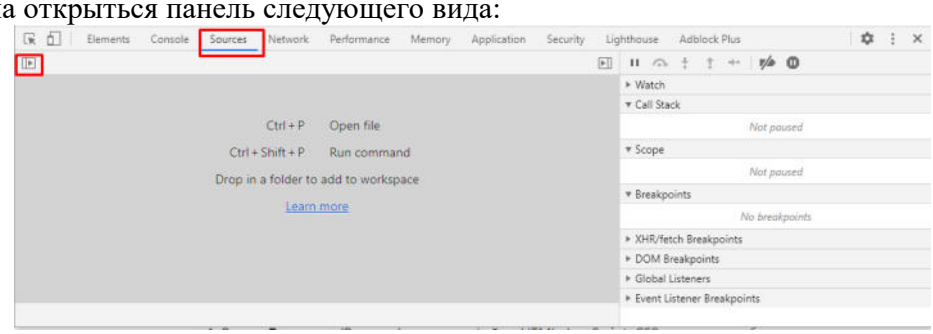
Пример динамического анализа кода в Chrome

Рассмотрим возможность отладки кода в браузере Chrome. Для этого:

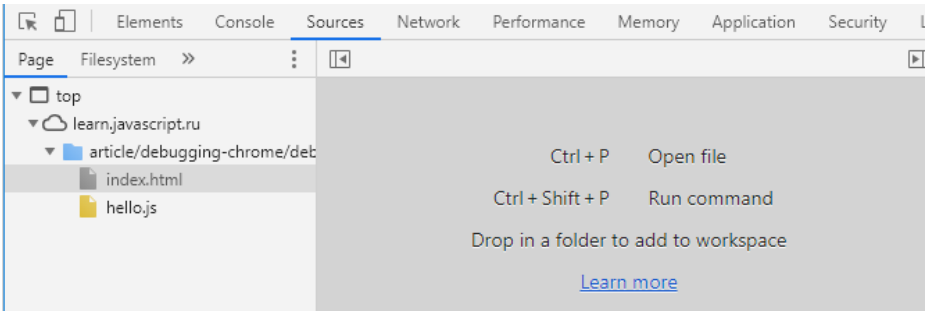
Включите инструменты разработчика, нажав F12;

Щёлкните по панели **sources** («исходный код»);

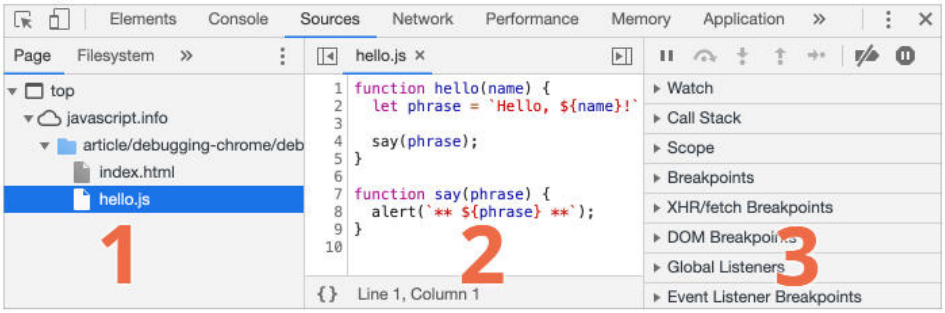
В итоге должна открыться панель следующего вида:



Кнопка-переключатель  откроет вкладку со списком файлов:



Кликните на кнопку-переключатель и выберите hello.js. Панель разделится на 3 зоны:



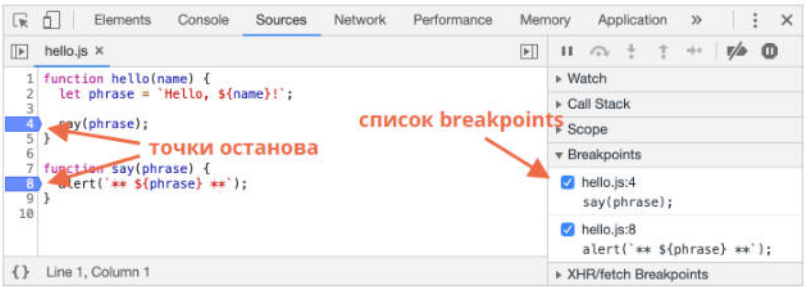
Перед Вами интерфейс отладчика, состоящий из трёх частей:

1) В зоне **Resources** (Ресурсы) показаны файлы HTML, JavaScript, CSS, включая изображения, используемые на странице. Здесь также могут быть файлы различных расширений Chrome.
2) Зона **Source** показывает исходный код.
3) Третья зона **Information and control** (Сведения и контроль) отведена для отладки.

Давайте разберёмся, как работает код нашей тестовой страницы.

Точки останова

В файле hello.js щёлкните по строчке номер 4, именно по самой цифре, не по коду, чтобы поставить первую точку останова. Далее щёлкните по цифре 8 на восьмой линии. Номер строки будет окрашен в синий цвет.



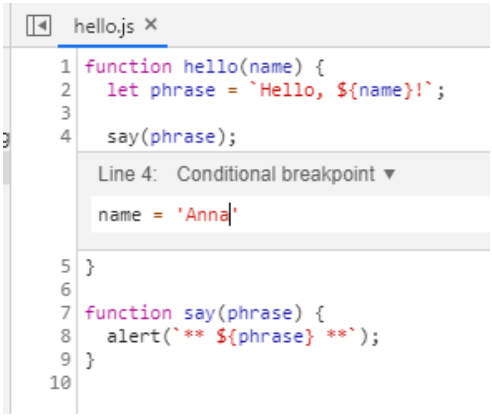
В правой части графического интерфейса представлен список точек останова, с помощью которого ими можно эффективно управлять (перейти к нужной, удалить, деактивировать и т.д.).

Условные точки останова

Можно задать и так называемую условную точку останова.

Для этого щёлкните правой кнопкой мыши по номеру строки в коде и выберите «Add conditional breakpoint».

Если задать выражение, то именно при его истинности выполнение кода будет приостановлено.



Этот метод используется, когда выполнение кода нужно остановить при присвоении определённого выражения какой-либо переменной или при определённых параметрах функции.

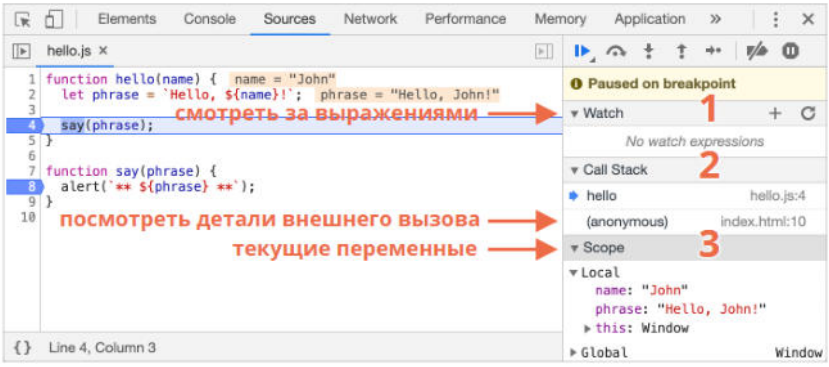
Debugger
Выполнение кода можно также приостановить с помощью команды debugger прямо изнутри самого кода:

```
function hello(name) {
  let phrase = `Hello, ${name}!`;

  debugger;  // <-- здесь выполнение прерывается

  say(phrase);
}
```

Способ удобен тем, что можно продолжить работать в редакторе кода без необходимости переключения в браузер для выставления точки останова.
Далее проведём динамический анализ кода кода.
Перезагрузим страницу, после чего выполнение прервётся н а 4-ой строке:



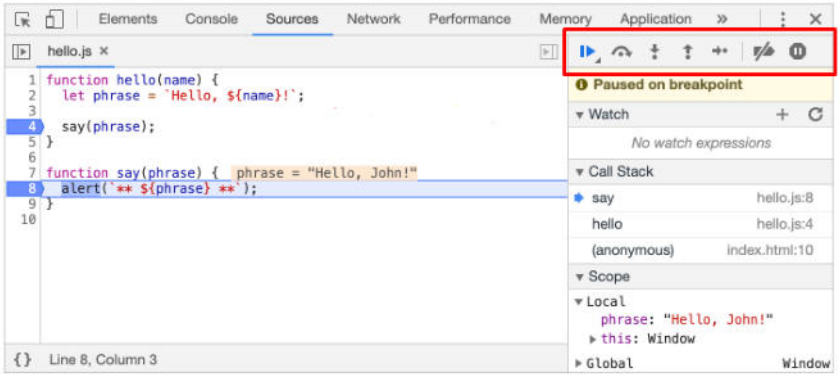
Чтобы понять, что происходит в коде, щёлкните в зоне **Information and control** по стрелочкам:

- **Watch** показывает текущие значения выражений. Нажмите на + и введите выражение. В процессе выполнения отладчик автоматически пересчитывает и выводит его значение.
- **Call Stack** показывает последовательность вызовов функций.В нашем примере отладчик работает с функцией hello(), вызванной скриптом из файла index.html (там нет функции, поэтому вызов «анонимный»).При нажатии на элемент списка (например, на «anonymous») отладчик переходит к соответствующему коду, и нам представляется возможность его проанализировать.
- **Scope** показывает текущие переменные.
- В **Local** отображаются локальные переменные функций, а их значения подсвечены в

исходном коде.

- В **Global** перечисляются глобальные переменные (т.е. объявленные за пределами функций).Трассировка скрипта.

Для пошагового выполнения скрипта обратимся опять к зоне **Information and control** и рассмотрим имеющиеся кнопки:



– продолжить выполнение (клавиша – F8).
Возобновляет выполнение кода. Если больше нет точек останова, отладчик прекращает работу и позволяет приложению работать дальше.

– сделать шаг (выполнить следующую команду), не заходя в функцию (клавиша – F10).
Если мы нажмём на неё – будет вызван *alert*. Важно: на месте *alert* может быть любая другая функция, выполнение просто перешагнёт через неё, полностью игнорируя её содержимое.

– сделать шаг (клавиша – F11).
В отличие от предыдущего примера, здесь мы «заходим» во вложенные функции и шаг за шагом проходим по скрипту.

– продолжить выполнение до завершения текущей функции (клавиша – Shift+F11).
Выполнение кода остановится на самой последней строчке текущей функции. Этот метод применяется, когда мы случайно нажали и зашли в функцию, но нам она неинтересна и мы как можно скорее хотим из неё выбраться.

– активировать/деактивировать все точки останова.
Эта кнопка не влияет на выполнение кода, она лишь позволяет массово включить/отключить т очки останова.

– разрешить/запретить остановку выполнения в случае возникновения ошибки.
Если опция включена и инструменты разработчика открыты, любая ошибка в скрипте приостанавливает выполнение кода, что позволяет его проанализировать. Поэтому если скрипт завершается с ошибкой, открываем отладчик, включаем эту опцию, перезагружаем страницу и локализуем проблему.

Итак, используя отладчик мы можем проанализировать переменные и пошагово пройти по процессу в поисках интересующей нас информации.

Лабораторная работа №3. Защита программных реализаций от исследования

Средства обфускации JavaScript

Функции btoa и atob

В простых случаях для защиты от парсеров не нужны громоздкие инструменты обфускации и достаточно просто «спрятать» некоторые строки.

Допустим, некоторый сайт sample.site стали проксировать на другом домене.

То есть выводится содержимое сайта и в нём заменены все ссылки на чужой домен, чтобы при переходе по ссылкам пользователь оставался на этом постороннем сайте.

Поскольку сайт копировался полностью, вместе со всеми скриптами, то достаточно добавить подобный код:

```
<script>
        if (/(www\.)?sample\.site/.test(window.location.hostname)) {

        }
        else {
         window.location = 'https://sample.site';
        }
</script>
```

Этот код проверяет, на каком домене открыта страница, и если эта страница не на домене sample.site, то делается переход на https:// sample.site.

Проблема в том, что при проксировании все упоминания sample.site заменяются на ЧУЖОЙ_ДОМЕН.ru, в результате и код превращался в:

```
<script>
        if (/(www\.)?sample\.site/.test(window.location.hostname)) {

        }
        else {
         window.location = 'https://ЧУЖОЙ_ДОМЕН.ru';
        }
</script>
```

Чтобы усложнить жизнь кодокопателю, код нужно обфусцировать, причём, простой минимизации кода будет недостаточно.

Самый простой вариант - спрятать строку «sample.site» с помощью функций btoa и atob.

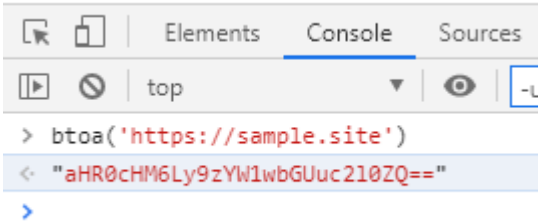
Функции btoa и atob являются встроенными функциями JavaScript и всегда доступны.

Функция btoa переводит указанную строку в набор символов (работает наподобие Base64), а функция atob выполняет обратную операцию.

Применим функции btoa и atob к тестовой строке:

```
btoa('Some text'); // U29tZSB0ZXh0
atob('U29tZSB0ZXh0'); // Some text
```

Далее, посмотрим, во что превратится строка https://sample.site . Это можно сделать прямо в консоли Chrome.



Теперь в исходном коде делаем с помощью функции atob обратное преобразование этой строки, получаем:

```
if (/(www\.)?sample\.site/.test(window.location.hostname)) {

}
else {
    window.location = atob('aHR0cHM6Ly9zYW1wbGUuc2l0ZQ==');
}
```

Данный код делает именно то, что нужно — проверяет на каком домене сайт был открыт и в случае если это не sample.site, то делает редирект на sample.site.

При такой замене всего одной строки, парсеры сайта не найдут строку sample.site и не сделают никаких изменений в этом фрагменте кода, а знчит, перенаправление на ЧУЖОЙ_ДОМЕН.ru не произойдёт.

3. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие сетевые атаки могут быть реализованы в рамках модели «искажение»?
2. Что такое информационный поток?
3. Как в рамках субъектно-ориентированной модели описывается операция порождения нового субъекта доступа?
4. Какие модели взаимодействия программной закладки с атакуемой системой вы знаете?
5. Как формально определяется модель «наблюдатель»?
6. Является ли задача выявления компьютерного вируса алгоритмически разрешимой в общем случае?
7. Как устроены перехватчики паролей второго рода?
8. Какие средства динамического изменения полномочий поддерживаются операционными системами семейства UNIX?
9. Когда появились первые компьютерные вирусы?
10. Почему написать вирус для Windows сложнее, чем для MS-DOS?
11. Почему первые макровирусы так широко распространились?
12. Как формально определяется модель «уборка мусора»?
13. Каким требованиям должен удовлетворять эффективно размножающийся компьютерный вирус?
14. Что означает требование универсальности, предъявляемое к компьютерным вирусам?
15. Для программы с переполнением буфера напишите эксплойт, не приводящий к досрочному завершению атакованной программы.
16. Как переполнение буфера в стеке программы позволяет нарушителю передать управление на произвольный адрес в текущем адресном пространстве?
17. Как отлаживать в Microsoft Visual Studio консольную программу, запущенную в режиме перенаправления стандартного ввода?
18. Скомпилируйте программу с переполнением буфера с опцией компилятора /GS. Убедитесь, что переполнение буфера невозможно поэксплуатировать.
19. К какому классу компьютерных вирусов относится вирус Морриса?
20. Сколько времени обычно требуется для заражения незащищенного компьютера, подключенного к Интернету?
21. Почему прогнозы аналитиков о грядущем «вирусном апокалипсисе» не оправдались?
22. Сколько времени прошло от опубликования уязвимости RPC DCOM Exploit до начала эпидемии вируса MSBlast?
23. Какие вредоносные воздействия на зараженные системы осуществлял вирус MSBlast?
24. Чем отличаются онлайн-вирусы от почтовых вирусов?
25. Каковы основные этапы жизненного цикла онлайн-вируса?
26. Как почтовые вирусы чаще всего прикрепляют свое тело к зараженному письму?
27. В чем заключается метод прикрепления тела почтового вируса к зараженному письму, основанный на встроенных кодах HTML?
28. Как почтовые вирусы используют в ходе распространения уязвимости программного обеспечения атакуемых систем?

29. На какие две группы делятся методы защиты от программных закладок?
30. Что такое принцип минимизации программного обеспечения?
31. Что такое принцип минимизации полномочий?
32. Что такое изолированная программная среда?
33. Какие требования предъявляются к программно-аппаратным средствам антивирусной защиты?

4. ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. Постановка задачи анализа программных реализаций.
2. Метод экспериментов с “черным ящиком”.
3. Статический метод.
4. Динамический метод.
5. Особенности анализа некоторых видов программ.
6. Постановка задачи защиты программных реализаций от изучения.
7. Динамическое изменение кода программы.
8. Искусственное усложнение структуры программы.
9. Нестандартные обращения к функциям операционной системы.
10. Искусственное усложнение алгоритмов обработки данных.
11. Выявление факта выполнения программы под отладчиком.
12. Программные закладки и формальные модели их взаимодействия с атакуемой системой.
13. Формальная модель “наблюдатель”.
14. Формальная модель “перехват”.
15. Формальная модель “искажение”.
16. Методы внедрения программных закладок.
17. Компьютерные вирусы.
18. Средства и методы защиты от программных закладок.
19. Организационные и административные меры антивирусной защиты.