Curso Programação Backend com Python

Segurança de aplicações backend

Sumário

Segurança

Hoje em dia é um aspecto muito importante de qualquer aplicação, ainda mais com a vigência da **LGPD**.

Veremos aqui aspectos básicos de segurança de aplicações servidoras (**backend**).

Segurança da informação

Os princípios de segurança da informação devem ser garantidos pela aplicação e sua infraestrutura. São eles:

- **Confidencialidade**: Apenas as entidades autorizadas pelo proprietário da informação possuem acesso a ela;
- **Integridade**: Garantia que a informação tenha todas as características originais, incluindo: Conteúdo, formato e até mesmo controle de alterações;
- **Disponibilidade**: Garantia de que a informação esteja sempre disponível para seu uso legítimo;
- **Autenticidade**: Garantia de que a informação provém da fonte original e que não foi adulterada ao longo do caminho;

Aspectos da segurança

Podemos dividir a segurança em três aspectos principais:

1. **Segurança física**: Segurança do datacenter, dos servidores e dos equipamentos de rede;
2. **Segurança em trânsito**: Proteção contra acessos indevidos, proteção à privacidade e autenticidade das informações em trânsito;
3. **Segurança de aplicação**: Proteção contra falhas do software que permitam escalação de privilégios ou ataques à privacidade e confidencialidade através da aplicação;

Segurança física

Inclui tudo o que é externo à aplicação ou à rede utilizada. Talvez, você pense que segurança física seja supérflua nesse contexto, já que estamos abordando desenvolvimento de aplicações. Mas há cuidados e atitudes, muitas vezes negligenciados e que afetam significativamente os princípios da segurança da informação:

- **Segurança das instalações**: Somente tem acesso as pessoas devidamente autorizadas e identificadas;
- **Controle de acesso**: Uso de credenciais únicas e intransferíveis, além de segurança baseada em múltiplos fatores (senha e digital, por exemplo);
- **Segurança em trânsito**: As informações devem transitar por canais seguros, mesmo entre servidores no mesmo datacenter.

Grandes empresas de nuvem impedem visitas físicas aos seus datacenters. Não promovem “tours” nem mesmo de estudantes.

Alguns erros muito comuns acontecem em datacenters e empresas, os quais representam graves violações aos princípios de SI (Segurança da Informação), entre eles:

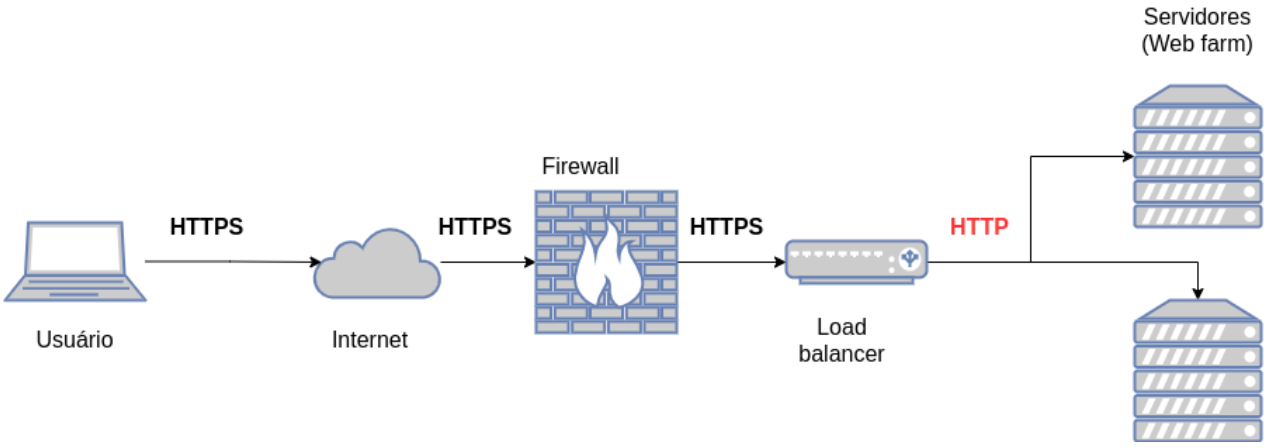
- **Porta da sala de servidores aberta**: Pode não estar aberta, mas é de fácil acesso, sem necessidade de identificação;
- **Uso de senha de “root”:** Compartilhamento e acesso com usuário “root” (ou “sa”, em caso de bancos de dados), sem identificação pessoal;
- **Empréstimo de senha**: Prática em que os funcionários “emprestam” senhas uns para os outros, seja por motivo de férias ou para “quebrar galhos”;
- **Falta de proteção de registros (logs)**: Logs de acesso devem ser protegidos, assim como as informações trafegadas no datacenter. O Marco Civil da Internet já prevê isso;
- **Uso de dados reais para teste**: Sem autorização do proprietário das informações, é prática muito comum os desenvolvedores utilizarem extratos dos dados reais para testar seus programas;
- **Falta de proteção dos dados em repouso**: Dados em repouso, seja em discos rígidos, mídias removíveis (CD, DVD, USB) ou backup (fitas, DVDs) devem ser mantidos preferencialmente encriptados e fora de alcance indevido.

Segurança em trânsito

Segurança em trânsito é um aspecto extremamente importante nessa nossa sociedade que espera a ubiquidade da informação. Proteger a informação em trânsito é mais do que usar um Certificado Digital.

Um exemplo clássico é: A comunicação entre servidores dentro do datacenter está encriptada? Eu já atuei como perito judicial e sempre que perguntei isso a resposta era algo como: “Ah, dentro do datacenter não precisa!”

Outro exemplo muito inocente é a questão dos **intermediários** (load balancers, proxies reversos etc), como neste exemplo:



Em algumas arquiteturas físicas, a criptografia não segue de “ponta a ponta”, sendo desfeita em equipamentos intermediários, como nessa figura.

As informações podem ser encriptadas e assinadas digitalmente independentemente do mecanismo de proteção de rede. Considere isso em seu projeto de arquitetura de software para aplicações backend.

Segurança de aplicação

É tudo o que você pode fazer para garantir os princípios da SI (Segurança da Informação) no seu papel de desenvolvedor de software. E não se engane: Há muita coisa a ser feita!

A primeira coisa que devemos pensar é: Vou usar SSL! Mas isso é apenas segurança de trânsito. O que acontece quando a informação chega e sai do seu backend? Como ela é tratada? Alguns princípios podem ser seguidos:

1. **Usar apenas as informações estritamente necessárias para produzir os resultados**. Evite coletar informações desnecessárias, evite acessar dados desnecessários do Banco de Dados. Peça ao DBA para criar uma View contendo só o que você precisa;
2. **Cuidado com "vazamentos" de dados**. Mensagens em log, arquivos secundários e "prints" na console, geralmente utilizados durante o desenvolvimento, podem revelar informações sensíveis;
3. **Dados que trafegaram encriptados, devem permanecer assim**. Não adianta usar SSL se você armazena ou re-transmite os dados em formato clear text;
4. **Cuidados com ataques às aplicações**. SQL Injection e Cross-site scripting ainda fazem muitas vítimas por aí, tome cuidado ao construir suas aplicações, especialmente as suas consultas SQL;
5. **Não armazene senhas dentro dos programas**. E nem em arquivos de configurações. Use variáveis de ambiente, em containers imutáveis;

Proteção em trânsito

Uma boa medida é adotar SSL - Secure Sockets Layer na comunicação do seu **backend**. Para isto, você necessitará de um Certificado Digital, contendo chave pública assinada. Você pode adquirir um certificado em empresas conhecidas, como a Certisign (https://loja.certisign.com.br/).

Para desenvolvimento, você pode cortar custos e utilizar um certificado livre, como o Let's Encrypt (muita gente usa em produção):

https://letsencrypt.org/pt-br/

Ou você pode criar um certificado auto-assinado, que serve para propósito de desenvolvimento, mas não para produção. Por que? Bom, todos os navegadores possuem as chaves públicas da maioria das Autoridades Certificadoras, podendo validar a assinatura da Chave pública do seu Certificado. Certificados auto-assinados não são reconhecidos pelos navegadores e o usuário receberá uma mensagem de alerta.

Como exemplo, veremos o uso de um certificado auto-assinado para implementarmos comunicação via SSL com o protocolo HTTPS.

Gerando um certificado auto-assinado

A maneira mais simples é criar um certificado utilizando o OpenSSL:

https://stackoverflow.com/questions/10175812/how-to-generate-a-self-signed-ssl-certificate-using-openssl

Vamos usar o comando seguinte para gerar o certificado:

```
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem
-days 365 -nodes
```

Use a opção “-nodes” para gerar seus arquivos PEM sem passphrase. A biblioteca Requests exige arquivos sem passphrase.

Verifique se você possui OpenSSL instalado:

```
$ openssl version
OpenSSL 1.1.1f  31 Mar 2020
```

Se você não tem o OpenSSL, pode instalar assim:

- MS Windows: https://www.xolphin.com/support/OpenSSL/OpenSSL_-_Installation_under_Windows
- Linux:
  - `apt-get install openssl`
  - ou: `dnf install openssl (CentOS)`
  - ou: `yum install openssl`

O OpenSSL pedirá algumas informações a você, como neste exemplo:

```
$ openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem
-days 365 -nodes
Generating a RSA private key
.....................................................................
.....................................................................
..................++++
.........................................++++
writing new private key to 'key.pem'

You are about to be asked to enter information that will be
incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a
DN.
```

```
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:BR
State or Province Name (full name) [Some-State]:RJ
Locality Name (eg, city) []:RJ
Organization Name (eg, company) [Internet Widgits Pty Ltd]:MySelf
Organizational Unit Name (eg, section) []:MS
Common Name (e.g. server FQDN or YOUR name) []:CLEUTON
Email Address []:cleuton@teste.com
```

E vai gerar dois arquivos:

- cert.pem: Sua chave pública auto-assinada;
- key.pem: Sua chave privada.

```
$ cat cert.pem
-----BEGIN CERTIFICATE-----
MIIF0zCCA7ugAwIBAgIUcjY1FYFQm8nqaPgcz1xX5bwQ19owDQYJKoZIhvcNAQEL
BQAweTELMAkGA1UEBhMCQlIxCzAJBgNVBAgMAlJKMQswCQYDVQQHDAJSSjEPMA0G
A1UECgwGTXlTZWxmMQswCQYDVQQLDAJNUzEQMA4GA1UEAwwHQ0xFVVRPTjEgMB4G
CSqGSIb3DQEJARYRY2xldXRvbkB0ZXN0ZS5jb20wHhcNMjEwNjIyMTc0NTE3WhcN
MjIwNjIyMTc0NTE3WjB5MQswCQYDVQQGEwJCUjELMAkGA1UECAwCUkoxCzAJBgNV
BAcMAlJKMQ8wDQYDVQQKDAZNeVNlbGYxCzAJBgNVBAsMAk1TMRAwDgYDVQQDDAdD
...
k9FWKYM0kiQ447UlKweKJaS3DWR35yFG9ony0emWbJDcPKG9OLNBYHZ0LroX5uSk
sSiv5uobGetB1GQqLrRKn800+Pq02uF7l5ylzixzfjVJoeKyF47lIw0s49T+NAZY
QJnosrgipA==
-----END CERTIFICATE-----

$ cat key.pem
-----BEGIN ENCRYPTED PRIVATE KEY-----
```

```
MIIJnDBOBgkqhkiG9w0BBQ0wQTApBgkqhkiG9w0BBQwwHAQItfNwIgB4uqcCAggA
MAwGCCqGSIb3DQIJBQAwFAYIKoZIhvcNAwcECKTZ6bXnQabzBIIJSJUuEYFU9ql0
7hRH8HGdyjArkEVH53qD6qhCUnOdRHHeJvDYa6ylFO0RPJX1uSDTZEso3g02pdww
lJPOvjPqepmlcTp84/nFfcPbP5kb05cwfr/1ddSWfCncPym8ka9rOlLqjPxK2l3r
+X0jcrQW8LDHoVFUJFGFgNfmyydk4fFo7qh0G6pDlkySpQsK438fK4QzWmJAOyG/
pk668mBY7uKGFuIGPk+eJJYJCwpAKHfhMIwYMJIvncqhEQ0zSfP3jxWYVBv/Y1qJ
...
abHa34Qh2Y8mRudVVc4fI/z+hAmCDa+RR/8u7xg8b4l4h3xieaBOOj1DIW29k3AK
upA6vpRmriZ3LgZww8NETFnXDKqkS1NrjOjBT9DT0Kbx0zr6MV6V8OsIQr+zEGev
BcVFcUDxdj/8bi31rqM3uQ==
-----END ENCRYPTED PRIVATE KEY-----
```

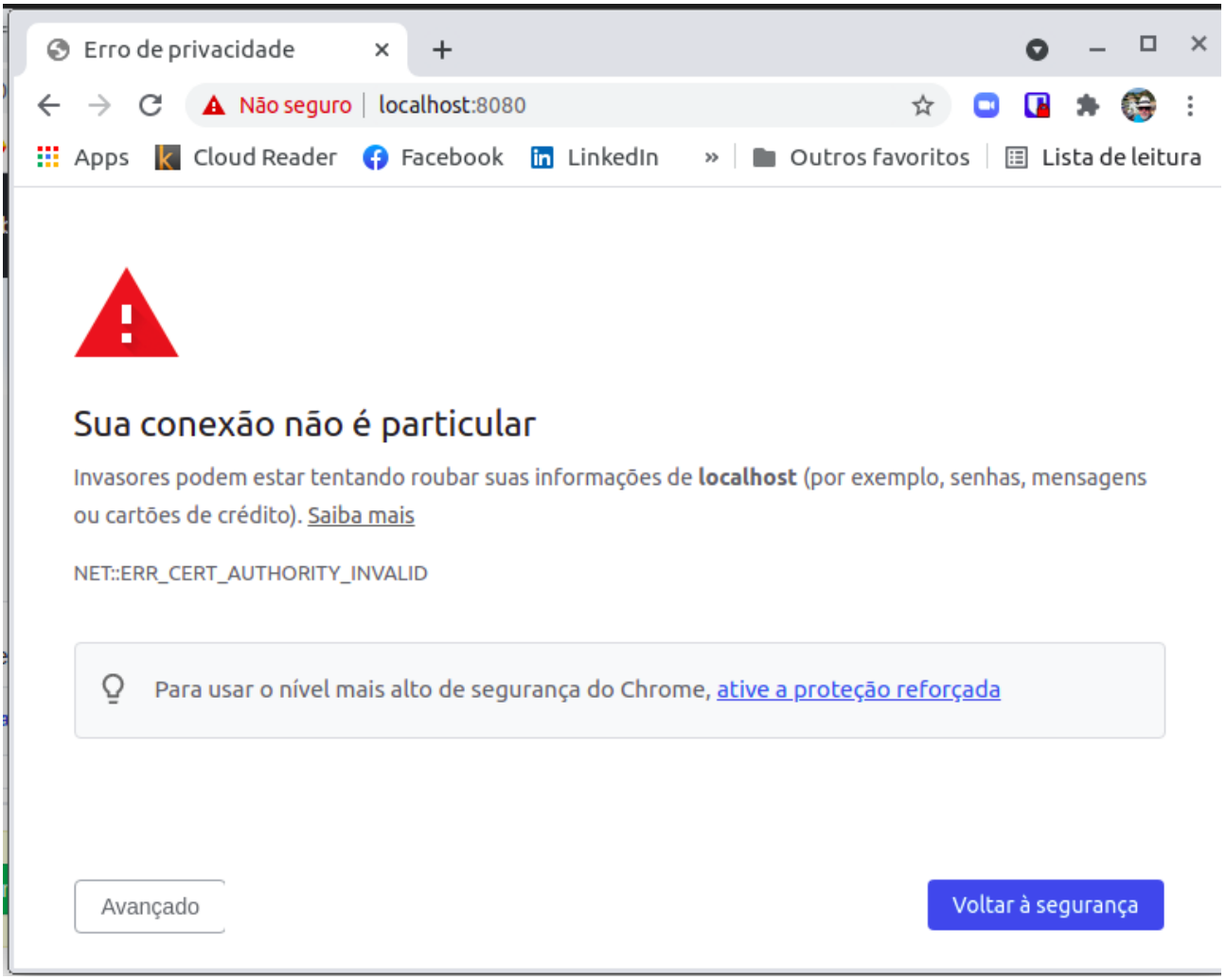
A sua chave primária é necessária para assinar coisas e a chave pública é o certificado que você vai usar. Note que ele pediu uma “passphrase”, que é uma senha para encriptar os arquivos PEM. Guarde o que você digitou.

Um servidor HTTPS

Baixe o arquivo “exemplosSeguranca.zip” e verá um script “servidorHTTPS.py”. Você deve criar um ambiente virtual com o VENV e instalar as dependências, que estão no arquivo requirements.txt. Depois, crie o certificado auto-assinado, conforme foi mostrado, e coloque os dois arquivos PEM na mesma pasta do script.

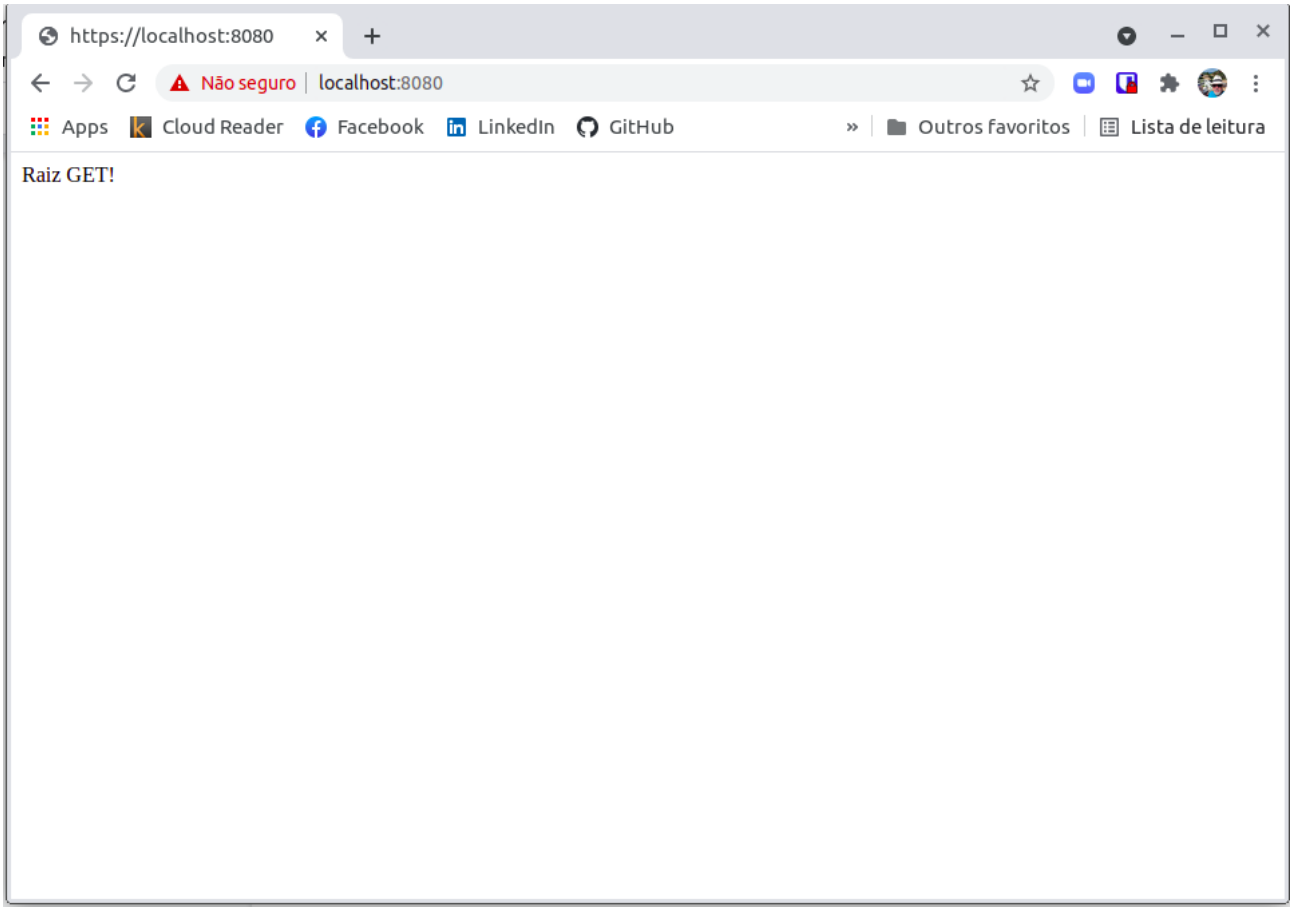
Execute o script e tente navegar para ele:

https://localhost:8080



Esse erro de privacidade acontece porque o navegador não reconheceu a assinatura do certificado. Por esta razão você deve comprar um certificado digital válido antes de colocar seu backend em produção.

Se clicar em “Avançado” conseguira acessar o site:



Pronto! Temos um servidor backend que usa SSL.

São poucas mudanças no código-fonte:

```
...
if __name__ == "__main__":
    context = ('cert.pem', 'key.pem') #certificate and key files
    app.run(host='0.0.0.0', port=8080, debug=True,
ssl_context=context)
```

Cliente HTTPS

Podemos utilizar a biblioteca requests para criar clientes capazes de acessar servidores HTTPS. No mesmo zip tem um script “clienteHTTPS.py” que faz isso.

Eis os resultados:

```
$ python3 clienteHTTPS.py
/home/cleuton/Documentos/projetos/udemy-python/seguranca/https/lib/python3.8/site-packages/urllib3/connectionpool.py:1013: InsecureRequestWarning: Unverified HTTPS request is being made to host 'localhost'. Adding certificate verification is strongly advised. See: https://urllib3.readthedocs.io/en/latest/advanced-usage.html#ssl-warnings
  warnings.warn(
<p>Raiz GET!</p>
/home/cleuton/Documentos/projetos/udemy-python/seguranca/https/lib/python3.8/site-packages/urllib3/connectionpool.py:1013: InsecureRequestWarning: Unverified HTTPS request is being made to host 'localhost'. Adding certificate verification is strongly advised. See: https://urllib3.readthedocs.io/en/latest/advanced-usage.html#ssl-warnings
  warnings.warn(
<p>Usuario = fulano</p>
/home/cleuton/Documentos/projetos/udemy-python/seguranca/https/lib/python3.8/site-packages/urllib3/connectionpool.py:1013: InsecureRequestWarning: Unverified HTTPS request is being made to host 'localhost'. Adding certificate verification is strongly advised. See: https://urllib3.readthedocs.io/en/latest/advanced-usage.html#ssl-warnings
  warnings.warn(
<p>beltrano, xpto</p>
```

Funcionou ok, só que ele gera warnings sobre o fato de não validarmos o certificado. Para usar HTTPS basta informar o protocolo correto na URL e, se for um certificado auto-assinado, use o parâmetro “verify=False”:

```
r = requests.get('https://localhost:8080', verify=False)
print(r.text)
```

Autenticação e autorização HTTP

Podemos proteger os dados em trânsito com SSL, mas como autenticar os usuários e verificar sua autorização de uso? Todos os seus recursos REST são públicos? Todos podem obter, postar e deletar recursos?

Você pode criar algum jeito de passar uma senha no request e utilizá-la para verificar se o usuário é válido e se tem permissão para usar o recurso REST. Mas o HTTP já tem um cabeçalho exclusivo para isto: `Authorization`. Este cabeçalho contém informações de autenticação do usuário que emitiu o request. Seu formato geral é:

```
Authorization: <tipo> <credenciais>
```

Vejamos um exemplo aqui:

```
> POST /token HTTP/1.1
> Host: localhost:8080
> User-Agent: insomnia/2021.2.2
> Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
> Accept: */*
> Content-Length: 0
```

O argumento “**tipo**” identifica o esquema de autenticação a ser utilizado. No exemplo acima é o “**Basic**”, que é o mais comum.

Este esquema **Basic** por si só não é seguro, pois as credenciais são codificadas como strings BASE64 e passadas assim pela rede. BASE64 é totalmente reversível, portanto, é possível saber quais são as credenciais. Se estiver utilizando em conjunto com SSL, há um razoável nível de proteção, desde que a conexão SSL esteja terminando no Servidor final, e não em um proxy.

Um esquema mais seguro seria utilizar o **DIGEST**, que utiliza hashes em vez de passar as credenciais em formato aberto.

Finalmente, temos o esquema mais utilizado, que é o **Bearer**, empregado pelo protocolo OAuth 2.0, utilizado por várias empresas, como: Google, Facebook, Instagram e outros. O esquema Bearer é baseado em um Token de segurança. Aqui está em linhas gerais o funcionamento do OAuth 2.0, conforme especificado na RFC (https://datatracker.ietf.org/doc/html/rfc6750):

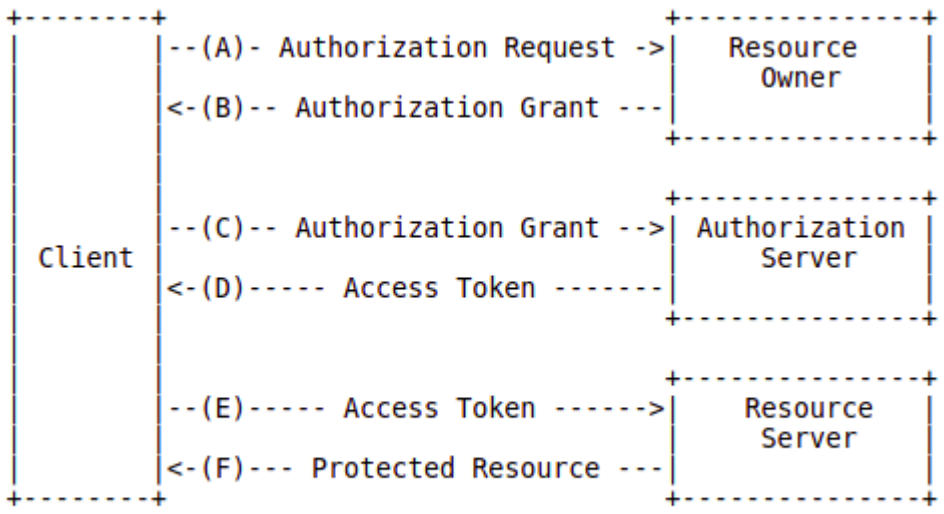


Figure 1: Abstract Protocol Flow

O usuário quer acessar um recurso protegido, então envia um request a um Resource Owner, que responde com um Authorization Grant (uma permissão). Então, ele envia esta permissão a um Authorization Server que emite um Token de acesso. O Token de acesso não contém os dados da credencial e pode ter uma data de expiração. Então, o usuário informa este token ao servidor web onde está o recurso, dentro do cabeçalho Authorization, utilizando o esquema Bearer.

Um dos formatos de token mais conhecidos é o JWT (JSON Web Token): https://jwt.io/introduction. Este token é assinado digitalmente e pode estar encriptado.

Exemplo de controle de acesso

Neste curso vamos ver o uso do controle de acesso com a biblioteca Flask-HTTPAuth, que permite usar vários esquemas de autenticação (Basic, Digest e Bearer).

Como exemplo vou mostrar um servidor que utiliza o esquema Basic e protege recursos REST. Acesse o servidor em:

https://github.com/cleuton/FalandoSobre.Software/blob/master/flatcode/fatcode_sample/servidor.py

Se quiser, pode baixar e executar o projeto todo clonando o repositório no Github:

https://github.com/cleuton/FalandoSobre.Software.git

Eis o arquivo requirements.txt do projeto para você ver as dependências:

```
flask
Flask-HTTPAuth
pyyaml
rsa
```

Começo importando duas bibliotecas envolvidas com isso:

```
from flask_httpauth import HTTPBasicAuth
from werkzeug.security import generate_password_hash,
check_password_hash
```

A biblioteca flast_httpauth me permite validar o cabeçalho Authorization, enviado pelo cliente web, com alguns **decorators**:

```
@auth.verify_password
def verify_password(username, password):
    if username in users and \
```

```
            check_password_hash(users.get(username), password):
        return username
```

A função decorada com @auth.verify_password é invocada a cada request e valida o header Authorization. Se o usuário tiver o header, então ela envia o username e a password (que pode ser um hash da password real). Se o usuário não tiver o token, ambos os parâmetros virão como string vazio (“”).

Neste exemplo, estou usando a função check_password_hash, da biblioteca werkzeug.security, para calcular e validar a senha contra o hashcode que eu criei no dicionário **users**.

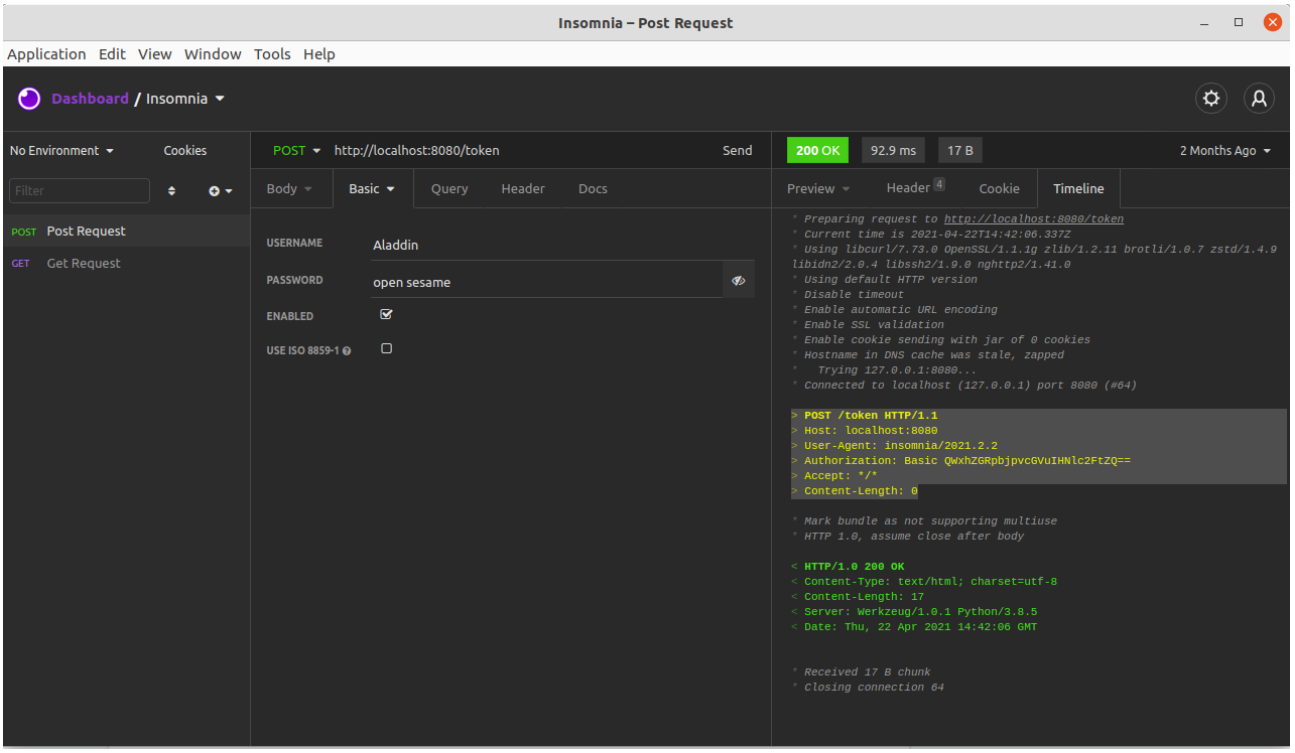
```
@app.route('/token', methods=['POST'])
@auth.login_required
def ask_token():
    result = {}
    result["token"] =
business_layer.start_calc(factory,auth.current_user(),poolx)
    return json.dumps(result)
```

Uma função decorada com @auth.login_required só será invocada se a autenticação foi bem sucedida, de acordo com a função decorada com @auth.verify_password.

Para testar este servidor eu usei o utilitário **insomnia**:

https://insomnia.rest/

Este utilitário me permite escrever um request com cabeçalhos e autenticações. Por exemplo:



Ele monta o request de acordo com os parâmetros que eu forneci, criando o header Authorizadion de acordo.

O mínimo que você deve fazer é utilizar algum destes esquemas de autenticação em seu servidor backend, em conjunção com SSL.