

Relatório Técnico Segurança Informática



Grupo 10

Nome	n.º Aluno
Tiago Duarte	42525
David Abreu	43316
Mário Friande	43785

Parte 1

1. Considere o protocolo TLS:

- (a) Porque motivo a propriedade *perfect forward secrecy* não é garantida usando o processo base com RSA para estabelecimento do *master_secret*?
- (b) Identifique dois possíveis ataques ao *record protocol* e explique as técnicas usadas para os prevenir.

(a) No RSA se a chave privada for comprometida, acontece que o pre master secret dos handshakes seguintes e dos anteriores (guardados pelo atacante) podem ser decifrados. Ao contrario se a propriedade perfect forward secrecy for preservada garante que se chave privada for comprometida não é possível ao atacante decifrar master secret anteriores e consequentemente não é possível decifrar mensagens do record protocol.

(b) Dois possíveis ataques são o ataque de repetição e o ataque de reflexão. No caso do ataque de repetição no TLS, este ataque é coibido porque a tag de autenticação da mensagem é computada sobre o seu número de sequência original. Em segurança informática, um nonce é um número arbitrário (muitas vezes aleatório) utilizado uma única vez (nonce vem da junção das palavras number e once). No TLS, o nonce - e, portanto, o número de sequência - é utilizado como parte da mensagem para fins de cálculo da tag de autenticação. Por exemplo se o atacante capturar uma mensagem e tentar reenvia-la quando o receptor fazer a confirmação da tag vai necessariamente perceber que a mensagem falha na autenticidade.

No caso do ataque de reflexão, a solução do TLS para este problema é o uso de material criptográfico diferente para cada sentido da comunicação. Como discutimos na secção anterior, as chaves e IV utilizados no sentido de A para B são diferentes daqueles usados no sentido de B para A.

2. Considere uma aplicação *web* que usa como informação de validação das *passwords* a forma $h_u = H(pwd_u || salt_u)$, sendo H uma função de *hash*, pwd_u a *password* do utilizador u e $salt_u$ um número aleatório gerado no momento do registo do utilizador u ($||$ representa a concatenação de bits).

O uso da técnica conhecida como CAPTCHA (*Completely Automated Public Turing Test to Tell Computers and Humans Apart*) contribui para mitigar ataques de dicionário à informação de validação? Explique.

Os CAPTCHAs são desafios que visam distinguir humanos de bots ou programas automatizados. Geralmente, esses desafios envolvem a identificação de letras distorcidas, seleção de imagens específicas, resolução de quebra-cabeças ou completar tarefas simples que são fáceis para humanos, mas desafiadoras para programas automatizados. Os CAPTCHAs são fáceis para humanos e difíceis para computadores.

3. Considere uma aplicação *web* que mantém estado de autenticação entre o *browser* e a aplicação servidor usando *cookies*. No *cookie* é guardado um JSON web token (JWT) com o identificador do utilizador. Como é que a aplicação servidor pode detetar se o conteúdo do *cookie* foi adulterado no *browser*?

Para garantir a integridade do JWT armazenado em um cookie e detectar se seu conteúdo foi adulterado no navegador, pode-se utilizar uma técnica de assinatura digital. O JWT possui três partes: o cabeçalho (header), o payload (corpo) e a assinatura. A assinatura é gerada usando uma chave secreta conhecida apenas pelo servidor. Essa assinatura é aplicada sobre o cabeçalho codificado em Base64, o payload codificado em Base64 e a chave secreta. Quando o JWT é enviado de volta ao servidor no cookie, o servidor pode recalcular a assinatura usando a chave secreta conhecida e comparar essa assinatura com a presente no JWT recebido. Se as assinaturas não coincidirem, isso indica que o conteúdo do JWT foi alterado.

4. Considere a norma OAuth 2.0 e OpenID Connect no fluxo *authorization code grant*:

(a) Em que situação está previsto o uso da estrutura JWT e com que objetivo?

(b) No OAuth 2.0, após o dono de recursos ter autorizado e consentido o uso de um recurso, descreva as ações da aplicação cliente para conseguir fazer os pedidos ao servidor de recursos.

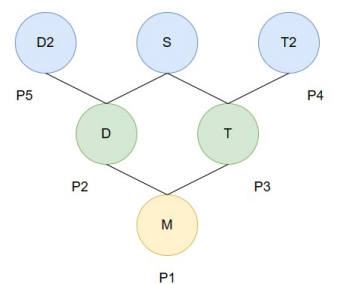
(a) Em resumo, os JWTs são usados no fluxo Authorization Code Grant do OAuth 2.0 e do OpenID Connect para fornecer tokens de acesso e identificação estruturados, compactos e seguros, permitindo a troca de informações entre os diferentes componentes do sistema de maneira confiável.

(b) A primeira ação que a aplicação cliente realiza é enviar uma requisição GET para o servidor de autorização, especificando um escopo determinado. Em seguida, o servidor de autorização redireciona para um URL específico de retorno (Authorization Grant). A aplicação cliente então envia uma requisição POST para um endpoint de token no servidor de autorização, requisitando um Access Token. Com o Access Token recebido, a aplicação cliente pode solicitar recursos ao servidor de autorização, incluindo o Access Token válido para acessar os dados desejados.

5. Considere o modelo de controlo de acessos $RBAC_1$. Para realizar o controlo de acessos aos recursos foi definida a seguinte política $RBAC_1$ que inclui os papéis (*M*)ember, (*D*)eveloper, (*T*)ester e (*S*)upervisor.

- $U = \{u_1, u_2, u_3, u_4\}$
- $RH = \{M \preceq T, M \preceq D, D \preceq S, T \preceq S, T \preceq T_2, D \preceq D_2\}$
- $UA = \{(u_1, M), (u_2, T_2), (u_3, D_2), (u_4, S)\}$
- $PA = \{(M, p_1), (D, p_2), (T, p_3), (D_2, p_5), (T_2, p_4)\}$

Justifique qual o conjunto total de permissões que podem existir numa sessão com o utilizador u_4 .



R: O conjunto total de permissões que podem existir numa sessão com o utilizador $u_4 = \{P1, P2, P3\}$. Como podemos verificar na figura o utilizador $u_4(S)$ no modelo de controlo de acessos $RBAC_1$ herda as permissões de $M(P1)$, $D(P2)$ e $T(P3)$.

desenvolver resposta 5

6. Configure um servidor HTTPS, **sem** e **com** autenticação de cliente. Tenha por base o ficheiro do servidor no repositório github da disciplina (HTTPS-server\http-server-base.js). Considere o certificado e chave privada do servidor `www.secure-server.edu` em anexo, o qual foi emitido pela CA1-int do primeiro trabalho.

⇒ No documento de entrega descreva sucintamente as configurações realizadas.

- (a) Usando um *browser*, ligue-se ao servidor HTTPS sem e com autenticação de cliente Alice_2.
(b) Usando a JCA, realize uma aplicação para se ligar ao servidor HTTPS sem autenticação de cliente.

Tenha em conta as seguintes notas:

- Comece pelo cenário base: servidor HTTPS testado com cliente *browser* sem autenticação. Para executar o servidor tem de ter instalado o ambiente de execução node.js. Após a configuração mínima na secção `options` do ficheiro `http-server-base.js`, o servidor é colocado em execução com o comando:
`node http-server-base.js`
- As chaves e certificados no servidor node.js são configuradas usando o formato PEM. Para converter ficheiros CER (certificado) e PFX (chave privada) para PEM use a ferramenta de linha de comandos *OpenSSL*. Existem vários guias na Internet sobre o assunto, tendo todos por base a documentação oficial (<https://www.openssl.org/docs/manmaster/man1/>). Um exemplo de um desses guias pode ser visto aqui:
<https://www.sslshopper.com/article-most-common-openssl-commands.html>.
O ficheiro `secure-server.pfx` tem a chave privada do servidor e não está protegida por *password*.
- O certificado fornecido para configurar o servidor associa o nome `www.secure-server.edu` a uma chave pública. No entanto, o servidor estará a executar localmente, em *localhost*, ou seja, `127.0.0.1`. Para o *browser* aceitar o certificado do servidor, o nome do domínio que consta no URL introduzido na barra de endereços, <https://www.secure-server.edu:4433>, tem de coincidir com o nome do certificado. Para que o endereço `www.secure-server.edu` seja resolvido para *localhost*, terá de fazer a configuração adequada no ficheiro `hosts`, cuja localização varia entre diferentes sistemas operativos: [https://en.wikipedia.org/wiki/Hosts_\(file\)](https://en.wikipedia.org/wiki/Hosts_(file)).

```
openssl x509 -inform der -in secure-server.cer -out secure-server.pem
openssl pkcs12 -in secure-server.pfx -out private-key.pem -nodes
```

```
openssl x509 -inform der -in certificates-keys/end-entities/Alice_2.cer -out
Alice_2.pem
```

```
openssl x509 -inform der -in certificates-keys/intermediates/CA1-int.cer -out CA1-
int.pem
```

```
cat Alice_2.pem CA1-int.pem > certificate-chain.pem
```

7. Realize uma aplicação *web* para criar tarefas de um utilizador Google através da *Google Tasks API* [1] usando milestones de projetos GitHub [2]. É opcional, mas valorizado, o acesso a projetos GitHub privados.

Requisitos da aplicação:

- A aplicação só aceita pedidos de utilizadores autenticados, exceto na rota de autenticação. Os utilizadores são autenticados através do fornecedor de identidade social Google, usando o protocolo OpenID Connect [3, 4]. Após autenticação do utilizador na aplicação é dado acesso aos serviços, em função do papel atribuído ao utilizador pela política de segurança;
- Política de controlo de acessos: A aplicação usa o modelo RBAC1 para gerir 3 papéis (*roles*): *free*, *premium* e *admin*. No papel *free* os utilizadores apenas podem ver as suas tarefas, no papel *premium* podem ver e adicionar tarefas a partir de milestones GitHub. Por simplificação do exercício, o papel *admin* não tem funcionalidades específicas atribuídas, mas herda todas as outras. Deve ser visível na interface da aplicação o papel ativo.
- A política de controlo de acessos é aplicada usando a biblioteca *Casbin* [5]. A política é carregada no início da aplicação e usada pela biblioteca quando necessário nos *policy enforcement points* (PEP) da aplicação. Pode testar diferentes tipos de políticas usando o editor web do *Casbin*: <https://casbin.org/editor/>
- O estado de autenticação entre o *browser* e a aplicação web é mantido através de *cookies*;
- Os dados guardados para cada utilizador podem estar apenas em memória.

Não pode usar os SDK da Google e GitHub para realizar os pedidos aos serviços. Os mesmos têm de ser feitos através de pedidos HTTP construídos pela aplicação *web*, como demonstrado durante as aulas.

Considere os *endpoints* de registo e autorização de aplicações nos serviços Google e os endpoints equivalentes do GitHub [7]:

- Registo de aplicações: <https://console.developers.google.com/apis/credentials>
- *Authorization endpoint*: <https://accounts.google.com/o/oauth2/v2/auth>
- *Token endpoint*: <https://oauth2.googleapis.com/token>
- *UserInfo endpoint*: <https://openidconnect.googleapis.com/v1/userinfo>

⇒ No documento de entrega do trabalho deve explicar a política de controlo de acessos.

Referências

- [1] Google Tasks API. <https://developers.google.com/tasks/reference/rest>
- [2] Github Milestones API. <https://docs.github.com/en/free-pro-team@latest/rest/reference/issues#milestones>
- [3] Google OpenID Connect. <https://developers.google.com/identity/protocols/oauth2/openid-connect>
- [4] Especificação *core* do OpenID Connect. https://openid.net/specs/openid-connect-core-1_0.html
- [5] Biblioteca de autorização Casbin. <https://casbin.org/docs/en/overview>
- [6] Criação de credenciais na Google para as aplicações cliente.
- [7] Documentação sobre OAuth2 no GitHub <https://docs.github.com/en/apps/oauth-apps/building-oauth-apps/authorizing-oauth-apps#web-application-flow>

npm packages

```
npm install -g nodemon
npm install cookie-parser body-parser express express-handlebars axios path form-
data jsonwebtoken cookie-session config casbin
npm install --save-dev crypto
```

usefull links:

```
https://github.com/mfriande/repo-viewer
https://developers.google.com/tasks/reference/rest/v1/tasks/insert?hl=pt-br
https://developers.google.com/oauthplayground/?
code=4/0AfJohXm9vy13qEUQK6K0a11XLSdb_N9INwXdB12XYtEbv102vyFnD0gk9hKQqMR34y_Eg&sco
pe=https://www.googleapis.com/auth/tasks
```

