Contents

Arquitetura

UAP

A UAP tem:

- uma **interface gráfica**, para comunicação utilizador - app
- uma **API**, para comunicação app - browser
- um módulo de **pedidos https**, para comunicação app - servidor de autenticação
- o módulo de autenticação

O ficheiro da base de dados **não deve residir na mesma máquina que a UAP**. Assim, aumenta-se o número de dispositivos a comprometer para que haja alguma falha útil a um atacante.

As passes para desencriptar ficheiro e seus conteúdos devem obviamente ser diferentes. No entanto, para simplicidade de avaliação e testes, a que damos tem a mesma passe para ambas as tarefas.

Tudo é cifrado recorrendo a AES.

GUI

A interface gráfica é uma GUI simples em Swing.

Quando a UAP é aberta, começa por **desencriptar a base de dados**. Para tal, pede ao utilizador:

- localização do ficheiro

- passe para desencriptar o ficheiro
- passe para desencriptar os campos

API

A API não tem um certificado, dado que está exposta na rede local e não há transmissão de dados sigilosos.

Expõe apenas 2 métodos:

- /ping, para testar conectividade
- /init_auth, para iniciar o processo de autenticação

O website deve passar o url do servidor de autenticação ao chamar o `/init_auth`.

Módulo de pedidos HTTPS

Neste módulo está o método GET `sendChallenge(Challenge)`. Este método envia um desafio ao servidor e recebe como resposta um desafio dele, até o processo estar concluido.

Autenticador

Há neste módulo 3 autenticadores, dos quais apenas um funciona.

Começámos por implementar o `PBKDF2Authenticator`, no entanto não o conseguimos pôr a funcionar. Assim, criámos o `PythonAuthenticator`, um wrapper para o autenticador em python que foi usado para testar o protocolo.

Para garantir que o script não seria alterado (o que seria uma grave falha de segurança), antes de ser executado o seu hash MD5 é comparado com o hash esperado. Se não for igual não há autenticação.

O `PythonAuthenticator` valida a integridade do certificado da CA usado para assinar o certificado da aplicação web, valida o certificado da aplicação web com o certificado da CA e valida a correta formatação das mensagens do protocolo. Consoante o sucesso ou não do protocolo retorna um exit code diferente e uma mensagem descritiva.

O terceiro módulo, `DoubleRatchetAuthenticator`, foi começado para implementar um algoritmo mais complexo. Este algoritmo é muito complexo e não resolve muitos problemas sérios que os outros dois resolvem (apenas resolveria os valores estáticos na base de dados).

Servidor

No servidor (o site do projeto anterior) foram adicionados 2 novos endpoints:

- `/e-login` serve a página de login do protocolo e processa o login em si.
- `/e-chap` endpoint contactado pela UAP de forma a executar o protocolo. Todas as mensagens do protocolo passam por aqui.

O site também suporta agora HTTPS.

Para geração de valores aleatórios foi usado o módulo `secrets` do Python que alega usar a `melhor fonte de números aleatórios disponibilizada pelo Sistema Operativo`.

Após a criação de uma nova conta (fora de banda, por segurança) é apresentada uma `seed` ao utilizador que deve ser introduzida na UAP. Esta `seed` será usada para determinar a ordem dos bits a serem enviados.

Na implementação do protocolo foram impostas algumas restrições para uma execução com sucesso do protocolo:

- Se o autenticando demorar mais de 15 segundos entre cada mensagem o protocolo falha.
- O utilizador tem 30 segundos após a execução com sucesso do protocolo para efetuar o login na página ou o sessionID é invalidado.
- Uma mensagem mal formatada implica também a falha da execução do protocolo.

Esta implementação suporta vários autenticandos simultaneamente.

Browser

No lado do browser ao clicar no botão E-CHAP o browser vai efetuar um pedido `GET` ao endpoint `/ping` da UAP. Caso não obtenha resposta, ou receba um código de erro, é assumido que a UAP esteja desligada, e é apresentado um alerta ao utilizador informando-o dessa possibilidade. Caso contrário é efetuado um pedido ao endpoint `/init_auth` e o site identifica-se através do seu domínio de forma a iniciar o autenticação.

Protocolo

Embora o nosso protocolo efetue autenticação mútua. Iremo-nos referir ao lado do serviço (neste caso o site) como autenticador e ao cliente (neste caso a UAP) como autenticando.

O protocolo é iniciado pelo autenticando através de uma mensagem de *hello*, onde o utilizador é identificado. `/e-chap?hello=something&username=realusername`. A resposta a mesma mensagem de hello é constituída por 2 elementos:

- O sessionID (SID), identificador único para a sessão atual.
- O Server Challenge (SC), gerado pelo autenticador é um valor aleatório

O autenticando vai agora gerar o Client Challenge (CC), um valor aleatório com o mesmo tamanho do SC. De seguida são calculadas duas sínteses pbkdf2 da password. H1 e H2. O sal de cada uma destas é composto por:

- `H1`: CC + SC + SID
- `H2`: SID + CC + SC

Será também usada uma `seed`, um segredo conhecido por ambas as partes apenas. Esta `seed` será usada para determinar a ordem dos bits de H1 e H2 a serem enviados. Esta ordem deve ser aleatória e dependente apenas da `seed`. O autenticando envia agora SID + CC + H1(n), sendo H1(n) o bit de H1 correspondente à iteração `n` do protocolo.

O autenticador vai agora calcular também H1, H2 e a ordem dos bits a mandar. Vai calcular também H1(n) e comparar com o que recebeu do autenticando. Se estes valores não forem iguais o autenticador apenas vai mandar respostas aleatórias até ao fim do protocolo. Caso contrário vai calcular H2(n), sendo H2(n) o bit de H2 correspondente à iteração `n` do protocolo. O autenticador vai agora responder com SID + H2(n).

O autenticando vai agora calcular também H2(n) e comparar com o que recebeu do autenticador. Da mesma forma que este, se os valores não corresponderem, serão enviadas até ao fim do protocolo respostas aleatórias. Caso contrário o protocolo prossegue para a próxima iteração.

O protocolo sucede quando ambos os lados recebem todos os bits corretos.

Escolha parâmetros:

- Tamanho dos desafios: 64 bytes
- Número de iterações pbkdf2: 1000000
- Tamanho output pbkdf2: 32 bytes
- Número iterações do protocolo: 256

Instruções de execução

Instruções para executar cada aplicação podem ser encontradas no ficheiro `README.md` da sua pasta respetiva.

Referências

Devido a este projeto usar código do projeto anterior os acknowledgements deste projeto serão os mesmos do projeto anterior (que podem ser encontrados na pasta `analysis`) e os seguintes:

- Python secrets
- Python random
- Python cryptography
- Python requests
- Stackoverflow - Gerar ordem a partir de uma seed
- Mozilla - XMLHttpRequest
- Stackoverflow - Detetar timeout XMLHttpRequest

- CHAP
- CHAP
- CHAP

Para a UAP foi feita bastante pesquisa:

- Como fazer pedidos HTTP
- Funções de síntese criptográfica
- Hash MD5
- Criptografia simétrica em Java
- Encriptar/desencriptar Strings
- PBKDF2 em Java
- PBKDF2 em Java
- PBKDF2 em Java
- PBKDF2WithSHA512 em Java snippets from
- Como invocar scripts de python
- Como correr comandos da shell
- Como obter output de um script de python lançado por Java
- Como usar SQLite em java
- Como usar SQLite em java
- CORS no Spring

Para os algoritmos não implementados também se recorreu a recursos externos

- Exemplo de implementação HKDF
- Interface com bases de dados do KeePass

Divisão de tarefas

- João Felisberto (98003): Design do protocolo, comunicação uap -> servidor e uap -> browser, base de dados (uap), relatório
- Rúben Castelhano (97688): Design do protocolo, implementação do protocolo no site e do `PythonAuthenticator`, relatório
- Eduardo Cruz (93088): Design da Interface grafica para a Uap
- Vasco Santos (98391): Authenticator PBKDF2