Autenticação baseada em Zero Conhecimento, com cache de credenciais assimétricas temporárias

Pedro Miguel Nicolau Escaleira
escaleira@ua.pt

06/06/2021

Conteúdo

1 Introdução

Neste relatório, iremos tentar apresentar de forma breve a implementação dum sistema que utiliza um protocolo baseado em *Zero-knowledge Proof* para fazer autenticação das várias entidades em jogo. É importante ter em conta que este sistema é apenas uma prova de conceito, servindo unicamente para estudo da segurança associada a mecanismos de identificação, autenticação e autorização, em ambiente académico.

A nossa solução, tal como demonstrado no diagrama da figura 1, apresenta 3 entidades distintas: um *Identity Provider* (*IdP*), um *Service Provider* (*SP*) e uma *helper application*. O *SP* representa um serviço que permite que utilizadores autenticados acedam a uma página pessoal que, neste caso, permite o *upload* duma fotografia, que fica associada ao respetivo utilizador. De forma a que um dado utilizador possa usar o *SP*, terá de se autenticar perante um *IdP*, que implementa um serviço de *Single Sign-on* (*SSO*) [1] que irá efetuar um protocolo de *Zero-knowledge Proof* com uma *helper application* que será executada localmente na máquina do utilizador. Perante uma autenticação bem sucedida, o utilizador é autorizado a aceder aos seus recursos pessoais no *SP*. Para além do *Zero-knowledge Proof*, a nossa solução prevê também a utilização dum mecanismo de autenticação usando credênciais assimétricas.

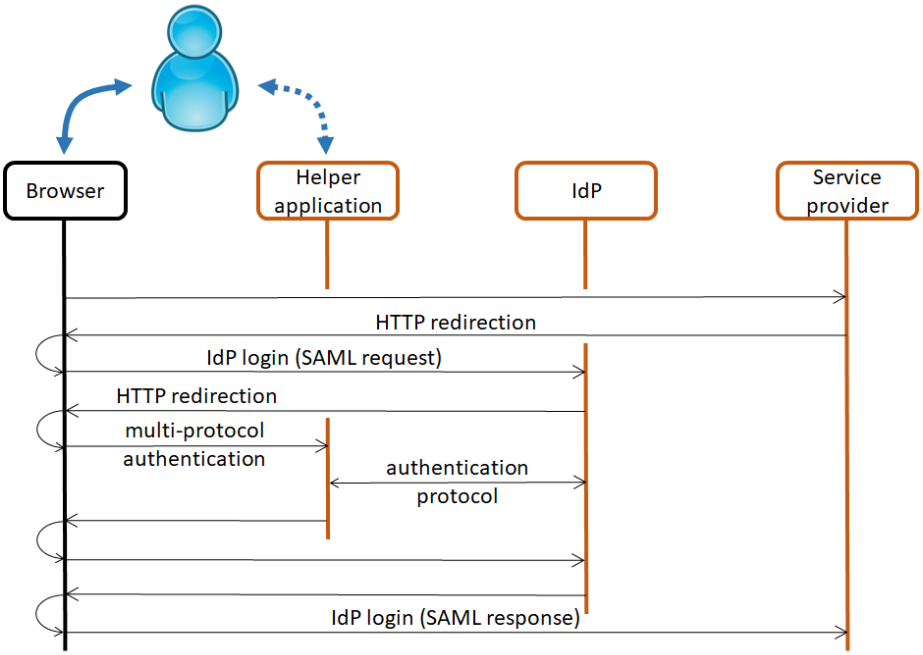

Figura 1: Arquitetura do sistema implementado, providenciada pelo professor da disciplina.

2 Utilização de SAML para identificação do utilizador entre o IdP e o SP

Da figura 1 apresentada na secção 1 é possível perceber que foi utilizado o *standard SAML* (*Security Assertion Markup Language*) para o *IdP* indicar ao *SP* qual utilizador está a tentar aceder à sua conta pessoal na plataforma. Na nossa solução, utilizamos as bibliotecas de *Python PySAML2* [2], na implementação do *IdP*, e *python3-saml* [3], na implementação *SP*, que nos permitiram fazer a criação e verificação facilitada de pedidos e de respostas *SAML* nesta linguagem de programação. Para além disso, usamos como guia a informação disponível no documento [4], principalmente no que toca à composição dos pedidos e respostas *SAML* e do tipo de pedidos que têm de ser realizados em cada situação (capitulo 5.1.2 do referido documento), de forma a seguirmos a norma corretamente.

O funcionamento do protocolo implementado para a autenticação com *SAML* pode ser descrito pelo conjunto sucessivo dos seguintes passos:

1. Primeiro, um utilizador acede ao *SP* pedindo um determinado recurso que, por sua vez, requer que a identidade do utilizador seja comprovada. Desta forma, o *SP* redireciona o utilizador para o *IdP* com um pedido de *SAML*. Um exemplo deste pedido é apresentado na listagem 1. Neste, o *SP* identifica-se propriamente, quer indicando quem ele é, através do campo *Issuer*, quer através da assinatura do pedido, providenciada no parâmetro *Signature* do pedido *GET* feito ao *IdP* (na listagem 2 é possível encontrar um exemplo dos valores dos parâmetros do pedido feito), criada usando a sua chave privada. Outra porção importante do pedido *SAML* é o valor do atributo *AssertionConsumerServiceURL* do campo *AuthnRequest*, que indica o *URL* para o qual o *IdP* terá de enviar a sua resposta ao pedido do *SP*.
2. Quando o *IdP* recebe o pedido de *SAML*, começa por verificar se a assinatura do pedido é válida de acordo com a chave pública do certificado público do *SP* que enviou o pedido. É de notar que o *IdP* possui armazenados localmente os certificados públicos dos SPs perante os quais faz identificação de utilizadores. Caso a assinatura seja válida, contacta a *helper application*, de maneira a que o utilizador se autentique usando as suas credenciais, sendo que os protocolos usados nesta fase serão explicados na secção 3. Caso a assinatura seja inválida, enviará uma resposta de erro com o código *HTTP 401 Unauthorized*. Posteriormente, caso a autenticação do utilizador, feita usando os protocolos de autenticação que mais tarde iremos explorar, tenha terminado com sucesso, o *IdP* irá criar uma resposta de *SAML* ao pedido inicialmente feito pelo *SP*, onde irá enviar informação sobre o utilizador autenticado (neste caso, o nome do utilizador na plataforma). Na listagem 3 é possível observar a resposta *SAML* enviada pelo *IdP* ao *SP*, perante o pedido que este último fez, já apresentado na listagem 1. Nesta resposta, podemos verificar que o nome do utilizador é enviado no campo *AttributeStatement:Attribute:AttributeValue* da asserção. Para além desta informação enviada sobre o utilizador autenticado, é de notar as seguintes características da resposta:
    - Ela é criada especificamente para o pedido de *SAML* inicialmente feito: o valor do atributo *IssueInstant* do campo *Response* é igual ao *ID* do pedido de *SAML*, apresentado na listagem 1.
    - Tal como o *SP*, o *IdP* identifica-se também usando o campo *Issuer* e comprova a sua identidade assinando o pedido, usando para isso a sua chave privada, enviando o correspondente resultado da assinatura como valor do campo *Signature:SignatureValue*.

    É de notar também que esta resposta enviada pelo *IdP* para o *SP* é feita através dum formulário *HTML*, que possui um *input* chamado *SAMLResponse*, onde irá a referida resposta. Este formulário é posteriormente submetido automáticamente no *browser* do utilizador no

momento em que é carregado (*POST binding*), enviando o pedido *POST* para o *endpoint* indicado anteriormente pelo *SP*, como referido no ponto 1.

3. Por fim, caso toda a autenticação tenha sido concluída com sucesso entre a *helper application* e o *IdP*, o *SP* recebe a resposta *SAML* com o correspondente nome do utilizador que tentou aceder ao recurso inicial. De forma a que o *IdP* possa confiar nesta resposta, verifica primeiro a assinatura da mesma, usando para isso o certificado publico do *IdP*, que possui armazenado localmente. Outra verificação importante que implementa é analisar se a resposta recebida foi feita ao pedido inicialmente feito, ou seja, se o *ID* do pedido *SAML* feito corresponde ao valor do campo *InResponseTo* da resposta. Caso estas verificações, e outras que são feitas para validar a integridade da resposta, resultem num parecer afirmativo, o *SP* aceita o valor do nome do utilizador enviado na resposta *SAML*, permitindo que este aceda ao recurso inicialmente pedido.

```
<samlp:AuthnRequest xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol" xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
                    ID="ONELOGIN_ecf975e199041d5ccd3fbd03ca14932cfa6f5361" Version="2.0"
                    IssueInstant="2021-06-03T17:50:45Z" Destination="http://127.0.0.1:8082/login"
                    ProtocolBinding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
                    AssertionConsumerServiceURL="http://127.0.0.1:8081/identity">
   <saml:Issuer>http://127.0.0.1:8081</saml:Issuer>
   <samlp:NameIDPolicy Format="urn:oasis:names:tc:SAML:1.1:nameid-format:unspecified"
                       AllowCreate="true" />
   <samlp:RequestedAuthnContext Comparison="exact">
      <saml:AuthnContextClassRef>
         urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport
      </saml:AuthnContextClassRef>
   </samlp:RequestedAuthnContext>
</samlp:AuthnRequest>
```

Listagem 1: Exemplo do pedido de *SAML* feito pelo *SP* ao *IdP*.

```
SAMLRequest=fVPBbuIwEL33K6LcaeKEQLEgEoXuLhILEWT3sJeVa09aS4mdtSdb+ve1U9pSiRIfIo3fe/PeZDK1rKlbOu/
     ↪ wUe3gXwcWr4Lg0NTK0v5qFnZGUc2stFSxBixFTvfzn2uaXMe0NRo113X4iXSZw6wFg1IrT1otZ+F2c7fefl9t/gKvJuMMyGQSD4nIOBdpdS/
     ↪ ilDMynKQJr9ioytIR8cTfYKzTmIVOsheytoOVssgUumKckEE8GsRpScY0i+kw++NRS5dPKoY98xGxpVFEkvF17A6hN/FNEtX6QfbWimO2W6mEVA+
     ↪ XQ92/giz9UZbFoNjuSy8xf4u60Mp2DZg9mP+Sw6/d+nx7EkkBCiU+h7njB8HUz5P24Ux+ljGNTiEfpJZunMvVstC15M993T/
     ↪ ftGkYfh2GOFlfkWJQ9VDaKdsC15UEEb7LzOtaPy0MMIRZiKaDMIg+NT8uE4h+tVx+
     ↪ hAMGC920zEjrxw8HxvEY8yPqKXxRu13ZQZVfXCdOuce5cuFeT9oI/+WAu96lYc68NngcOlnxV9fRBdv51dv16X+SvwA=
RelayState=http://127.0.0.1:8081/account
Signature=II/R/wUcz9i86BB5uzXmY0+9mJDDDDcOh55RArrPeJtivaK+1zkqygOZpinsdugdplhoLmdU7Fi5+mHvBOtz1V7E2KggNURcocR0zmR3/
     ↪ TXnJTakPRwEIFZGclJKZn/Eu/X28ec4nA/b6d3oOEGFYY1o6TMT37UXWCq8tuOiGWly512wav6NR/
     ↪ RbLNUG8q8L6dcAMzPdd0eVD07aCxcisw9YwIsBc4AzrpUhw2wtnkhhCOhoCi6O97uconDQQJth9D2TyYGgS+2
     ↪ ZzeyNKTc0de4EdpbvikuUKebMK4zgNO911pj8HkgPicotkSTubq51ZdiUxF48h6GiOmyGLS7htQ==
SigAlg=http://www.w3.org/2000/09/xmldsig#rsa-sha1
```

Listagem 2: Parâmetros do pedido *GET* correspondente ao pedido *SAML* apresentado na listagem 1.

```
<?xml version="1.0"?>
<ns0:Response xmlns:ns0="urn:oasis:names:tc:SAML:2.0:protocol" xmlns:ns1="urn:oasis:names:tc:SAML:2.0:assertion"
              xmlns:ns2="http://www.w3.org/2000/09/xmldsig#" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              ID="id-Lexd2HVcgbLesq8gk" InResponseTo="ONELOGIN_ecf975e199041d5ccd3fbd03ca14932cfa6f5361"
              Version="2.0" IssueInstant="2021-06-03T17:50:50Z" Destination="http://127.0.0.1:8081/identity">
   <ns1:Issuer Format="urn:oasis:names:tc:SAML:2.0:nameid-format:entity">
      http://127.0.0.1:8082
   </ns1:Issuer>
   <ns2:Signature Id="Signature1">
      <ns2:SignedInfo>
         <ns2:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
         <ns2:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
         <ns2:Reference URI="#id-Lexd2HVcgbLesq8gk">
            <ns2:Transforms>
               <ns2:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
               <ns2:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
               </ns2:Transforms>
               <ns2:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
               <ns2:DigestValue>+Kaf7UsBN3Kh77z7vz0SVoN6k/w=</ns2:DigestValue>
         </ns2:Reference>
      </ns2:SignedInfo>
      <ns2:SignatureValue>
         [Response Signature created with the \textit{IdP} Private Key]
      </ns2:SignatureValue>
      <ns2:KeyInfo>
         <ns2:X509Data>
            <ns2:X509Certificate>
               [\textit{IdP} Public Certificate in Base64]
            </ns2:X509Certificate>
         </ns2:X509Data>
      </ns2:KeyInfo>
   </ns2:Signature>
   <ns0:Status>
      <ns0:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:Success" />
   </ns0:Status>
   <ns1:Assertion Version="2.0" ID="id-Wj00uHeDaGwSA9WI5" IssueInstant="2021-06-03T17:50:50Z">
      <ns1:Issuer Format="urn:oasis:names:tc:SAML:2.0:nameid-format:entity">http://127.0.0.1:8082</ns1:Issuer>
      <ns1:Subject>
         <ns1:NameID NameQualifier="https://localhost:8082/idp.xml"
                     SPNameQualifier="http://127.0.0.1:8081"
                     Format="urn:oasis:names:tc:SAML:1.1:nameid-format:unspecified">
            7b128b9a1fe97f3ec9a379fcf0795ad27d69528abd3c58ecd687186159413685
         </ns1:NameID>
         <ns1:SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
            <ns1:SubjectConfirmationData NotOnOrAfter="2021-06-03T18:05:50Z"
                                         Recipient="http://127.0.0.1:8081/identity"
                                         InResponseTo="ONELOGIN_ecf975e199041d5ccd3fbd03ca14932cfa6f5361" />
         </ns1:SubjectConfirmation>
      </ns1:Subject>
      <ns1:Conditions NotBefore="2021-06-03T17:50:50Z" NotOnOrAfter="2021-06-03T18:05:50Z">
         <ns1:AudienceRestriction>
            <ns1:Audience>http://127.0.0.1:8081</ns1:Audience>
         </ns1:AudienceRestriction>
      </ns1:Conditions>
      <ns1:AuthnStatement AuthnInstant="2021-06-03T17:50:50Z" SessionIndex="id-ffEFa2f4nqVbAoErt">
         <ns1:AuthnContext>
            <ns1:AuthnContextClassRef>
               urn:oasis:names:tc:SAML:2.0:ac:classes:unspecified
            </ns1:AuthnContextClassRef>
         </ns1:AuthnContext>
      </ns1:AuthnStatement>
      <ns1:AttributeStatement>
         <ns1:Attribute Name="username" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
            <ns1:AttributeValue xmlns:xs="http://www.w3.org/2001/XMLSchema" xsi:type="xs:string">
               escaleira
            </ns1:AttributeValue>
         </ns1:Attribute>
      </ns1:AttributeStatement>
      </ns1:Assertion>
</ns0:Response>
```

Listagem 3: Exemplo da resposta SAML enviada pelo *IdP* ao *SP* ao pedido de *SAML* feito pelo *SP* ao *IdP*.

3 Autenticação

Na solução criada no âmbito deste trabalho, a autenticação do utilizador é feita usando uma aplicação local, como já apresentado na secção 1, denominada de *helper application*. Esta, tal como o *IdP* e o *SP*, é uma *REST API* que se encontra "a escutar"no *URL* "*zkp_helper_app:1080*"[1] por uma nova chamada feita pelo *IdP*, sendo que nesta circunstância apresenta uma interface gráfica no *browser* do utilizador, permitindo que este se autentique devidamente, como apresentaremos na secção 4. Quando o utilizador tenta aceder ao *SP*, este redireciona-o para o *IdP* com um pedido de *SAML*, como já apresentado na secção 2 e, por sua vez, o *IdP* redireciona-o para esta aplicação. Esta irá, de seguida, permitir ou uma autenticação baseada num segredo partilhado, utilizando um protocolo de *Zero-knowledge Proof* (*ZKP*), como será apresentado na subsecção 3.1 ou, caso seja possível, irá fazer a autenticação do utilizador com base em criptografia assimétrica, como será explicado na subsecção 3.2.

É importante denotar que nesta fase não é feita só a autenticação do utilizador, mas também do *IdP* i.e., a *helper application*, em qualquer um dos tipos de autenticação, faz a validação de se o *IdP* com quem está a comunicar é quem diz ser, de forma a evitar a partilha de informação confidencial com uma possível entidade maliciosa.

3.1 Protocolo de autenticação ZKP

O método de Prova de Conhecimento-zero, ou *Zero-knowledge Proof* (*ZKP*) em inglês, baseia-se na noção de uma entidade A ser capaz de provar a uma entidade B que possui determinada informação, sem que haja revelação da mesma. Em termos práticos, e no contexto deste problema, significa que a *helper application* tem de provar que possui um segredo partilhado entre ela e o *IdP*, que neste caso é a palavra-passe do utilizador, sem que de facto a envie ao *IdP* para que este a verifique. Para além disso, na nossa implementação, o *IdP* tem também de provar que tem conhecimento desse segredo partilhado, sendo necessário o estabelecimento dum elo de confiança entre as duas entidades.

Desta forma, o nosso protocolo prevê a troca de mensagens entre o *IdP* e a *helper application*, onde iterativamente fazem autenticação baseada em desafio-resposta, evitando sempre enviar informação que possa permitir desvendar o segredo partilhado, como iremos verificar de seguida.

Antes de passarmos à explicação do protocolo implementado, é necessário introduzirmos o significado de algumas variáveis usadas:

- i: representa o valor numérico da iteração atual. Quando as duas entidades iniciam o protocolo, o valor desta variável deverá ser igual a zero, sendo incrementado em ambas sempre que calculam um novo desafio.
- N: representa o número máximo de iterações feitas em cada uma das entidades. Desta forma, o protocolo deverá ser executado até $i >= 2N$. O valor de N é decidido entre as duas entidades da seguinte forma:
  1. Primeiro, quando o *IdP* faz o redirecionamento do utilizador para a *helper application*, envia dois parâmetros adicionais no pedido *GET*: *max_iterations* e *min_iterations*. Estes, como o próprio nome indica, definem o número de iterações (ou seja, o valor de N) máxima e mínima que o *IdP* aceita.
  2. Quando a *helper application* recebe este pedido, antes de iniciar o protocolo *ZKP*, verifica se o intervalo de iterações aceite pelo *IdP* coincide com o que ela aceita, fazendo uma

[1] É de ter em conta que é necessário introduzir este *host name* na *DNS* local do utilizador, associada ao *IP* local utilizado pela *helper application*. Em *linux*, isso é feito editando o ficheiro "*/etc/hosts*".

interceção entre o intervalo que ela aceita e o intervalo que o *IdP* aceita. Caso a interceção resulte num intervalo vazio, esta não inicia o protocolo *ZKP* com o *IdP*, apresentando uma mensagem de erro ao utilizador, alertando-o de que o *IdP* contactado poderá não ser confiável, como demonstrado na imagem 2. Por outro lado, caso a interceção resulte num intervalo válido, a aplicação irá selecionar um valor aleatório do mesmo e utilizar esse como valor de N, enviando o seu valor na primeira mensagem do protocolo *ZKP* ao *IdP*.

3. O *IdP*, ao receber a primeira mensagem correspondente ao protocolo *ZKP*, começa por verificar se o valor de N enviado pela *helper application* é válido i.e., se se encontra no intervalo de valores que ele aceita e que tinha previamente enviado à *helper application*. Caso o valor seja inválido, o *IdP* não inicia o protocolo, enviando uma resposta de erro à *helper application* com o código *HTTP 406 Not Acceptable*. Caso contrário, inicia o protocolo *ZKP* com a *helper application*, como iremos apresentar de seguida.

- c_i: este valor irá corresponder ao desafio associado à iteração i. O desafio, quer na *helper application*, quer no *IdP*, é criado de forma aleatória. Para isso, é sempre gerado um novo *UUID* (*Universally Unique Identifier*), na sua versão 4, ou seja, um *UUID* que possui 122 *bits* completamente aleatórios:
$$c_i = UUID4() \tag{1}$$

- C_i: esta variável corresponde ao conjunto de todos os valores de c criados até à iteração i, inclusivé:
$$C_i = c_0 +\!\!+ c_1 ... +\!\!+ c_i \tag{2}$$

- R_i: corresponde à resposta ao desafio correspondente à iteração i. É calculado da seguinte forma:
$$R_i = f(C_i, P) = HMAC(C_i, P) \tag{3}$$
Onde P é o segredo partilhado entre o *IdP* e a *helper application*, ou seja, a palavra-passe do utilizador. É também de notar que foi usada a função *HMAC* (*Hash based Message Authentication Code*) com a função de *hash SHA-256*. Desta forma, obtemos um valor de R_i que depende unicamente do valor de C_i e de P e que não permite obter estes dois de forma direta (isto é, este valor não é reversível, não permitindo obter o valor de C_i ou de P a partir do valor de R_i).

- r_i: este valor corresponde a um *bit* de R_i obtido a partir duma posição pré-definida (de forma a que as duas entidades possam obter o mesmo valor de r_i para o mesmo R_i, qualquer que seja i) da seguinte forma:
$$r_i = R_i[i \mod length(R_i)] \tag{4}$$
Onde $length(R_i)$ representa o tamanho, em *bits*, de R_i, e $R_i[a]$ representa o *bit* na posição a de R_i.

Error:

The range of allowed iterations received from the IdP is incompatible with the range allowed by the local app. A possible cause for this is the IdP we are contacting is not a trusted one!

Figura 2: Mensagem de erro apresentada pela *helper application* quando o intervalo de iterações aceites pelo *IdP* intercetado com o seu resulta num intervalo vazio.

Posto este conjunto de noções, o protocolo *ZKP* implementado pode ser caracterizado pelo conjunto dos seguintes passos iterativos:

1. Assim que o *IdP* contacta a *helper application*, através dum redirecionamento quando recebe o pedido *SAML* do *SP*, como apresentado na secção 2, esta começa por verificar, como já visto acima, os valores máximo e mínimo das iterações que o *IdP* aceita, sendo que se estes forem válidos, inicia o protocolo. Neste, começa por iniciar o valor da variável i a zero e calcula o primeiro desafio, c_0, enviando-o ao *IdP*.
2. De seguida, o *IdP* recebe a primeira mensagem do protocolo enviada pela *helper application*. Começa então, como já explicado, por verificar se o valor de N recebido é válido de acordo com as suas especificações e, em caso afirmativo, procede para a inicialização do protocolo localmente:
   (a) Inicia o valor de i a zero.
   (b) Calcula a resposta, R_0, ao desafio recebido.
   (c) Obtém o valor de r_0 a partir de R_0.
   (d) Incrementa o valor de i.
   (e) Calcula um novo desafio, c_i.
   (f) Envia à *helper application* o valor de r_0 e c_i.
3. A *helper application* recebe a resposta do *IdP*, podendo ter um dos dois comportamentos seguintes:
   - Caso a *helper application* confie no outro interveniente i.e., no *IdP*, executa o conjunto de passos descritos na enumeração 1.
   - Caso a *helper application* já não confie no outro interveniente i.e., no *IdP*, executa o conjunto de passos descritos na enumeração 2.
4. O *IdP* recebe a resposta da *helper application*, podendo ter um dos dois comportamentos seguintes:
   - Caso o *IdP* confie no outro interveniente i.e., na *helper application*, executa o conjunto de passos descritos na enumeração 1.
   - Caso o *IdP* já não confie no seu interveniente i.e., na *helper application*, executa o conjunto de passos descritos na enumeração 2.
5. O conjunto de passos a partir do ponto 3, inclusivé, é repetido até que $i >= 2N$. Quando esta verificação é tida como verdadeira, o protocolo implementado prossegue para os passos seguintes.
6. O *IdP* cria uma resposta *SAML* ao pedido de *SAML* recebido pelo *SP*, como explicado na secção 2, guardando-a em *cache*, associando-a ao *ID* do pedido de *SAML* inicialmente feito pelo *SP*.
7. A *helper application* redireciona o utilizador para contactar o *IdP*, com um pedido que irá requisitar que a identidade do utilizador seja validada ao *SP*. O *IdP*, ao receber este pedido, verifica se possui em *cache* uma resposta *SAML* para o *ID* indicado no pedido (que deverá corresponder ao *ID* do pedido de *SAML* inicialmente feito pelo *SP*) e, em caso afirmativo, responde ao *browser* do utilizador com um formulário *HTML* que se irá submeter automáticamente, fazendo um pedido *POST* com a resposta de *SAML* ao *SP*, como apresentado na secção 2.

Enumeração 1: Conjunto de paços efetuados pela *helper application* e pelo *IdP* caso ainda confiem no outro interveniente no protocolo.

1. Verificação se a resposta r_i recebida é válida, ou seja, verificação se este valor é igual ao r'_i armazenado localmente. Caso seja igual, prossegue para o passo seguinte, caso contrário, deixa de confiar na outra entidade (no outro interveniente).
2. Incrementação do valor de i.
3. Calculo da resposta, R_i, ao desafio recebido.
4. Obtenção do valor de r_i a partir de R_i.
5. Incrementação do valor de i.
6. Calculo dum novo desafio, c_i. Com base neste, calculo do correspondente R_i e $r_i^{'}$, para posterior comparação com a resposta a receber do outro interveniente no protocolo.
7. Envio ao outro interveniente do valor de r_{i-1} e de c_i calculados nos passos anteriores.

Enumeração 2: Conjunto de paços efetuados pela *helper application* e pelo *IdP* caso já não confiem no outro interveniente no protocolo.

1. Incrementação do valor de i.
2. Calculo dum *bit* de forma aleatória, usando a função "*os.urandom*"do *Python*.
3. Incrementação o valor de i.
4. Calculo dum novo desafio, c_i.
5. Envio ao outro interveniente do *bit* calculado aleatóriamente e do valor de c_i.

Como podemos entender do conjunto de passos apresentados anteriormente, o protocolo foi implementado de tal maneira que é passado o mínimo de informação entre os dois intervenientes sobre o segredo partilhado, de forma a que este continue secreto havendo, para isso, sempre a transmissão dum único *bit*, obtido duma computação duma função de *hash* sobre desafios aleatórios e a correspondente palavra-passe (segredo partilhado). Desta forma, foi possível fazer juz ao nome do método usado: *Zero-knowledge Proof*.

Outra das características que foi alcançada com o protocolo implementado foi que, para além de não ser dada qualquer informação sobre o segredo, não é dada mais informação do que a necessária caso um dos intervenientes detete que o outro não é quem diz ser, ou seja, que não possui o segredo partilhado. Por um lado, quando uma das entidades deteta que a outra não é fidedigna, ao invés de retornar simplesmente uma mensagem de erro e terminar o processo de autenticação, continua o mesmo, de forma a iludir a entidade maliciosa. Por outro lado, quando um dos intervenientes deixa de confiar no outro, passa a enviar respostas puramente aleatórias i.e., que não foram obtidas a partir do segredo partilhado de qualquer forma que seja, mas que "aos olhos"da entidade maliciosa, parecem respostas normais. Através destes dois atributos, é possível assegurar que não é dada qualquer informação adicional a uma possível entidade maliciosa que esteja a tentar fazer-se passar ou pela *helper application*, ou pelo *IdP*.

Uma das considerações importantes neste protocolo é qual o valor de N que deve ser usado. Por um lado, se este for demasiado grande, o protocolo irá garantir mais segurança, uma vez que é necessário um maior conjunto de desafios e respostas para autenticar ambos os intervenientes, com o custo de demorar mais tempo, uma vez que isto tem como efeito o aumento do número de mensagens a serem trocadas em rede. Com um N menor, o contrário passa-se, ou seja, o protocolo torna-se menos seguro quanto menor for este valor, mas será mais rápido a ser efetuado. Desta forma, na nossa solução, o *IdP* aceita um intervalo entre 300 e 1000 e a *helper application* entre 200 e 500, para o valor de N, uma vez que, apesar de ser ligeiramente lento tendo em conta os padrões normais de espera do utilizador aquando da autenticação num *website*, parece-nos uma espera razoável, que garante alguma segurança.

Uma vez que este protocolo pode ser, em certas situações, como quando a largura de banda é reduzida, ou o valor de N é elevado, demorado, implementou-se um segundo protocolo que permite que a autenticação dos dois intervenientes seja feita de forma segura e eficiente. Neste, é utilizado um par de chaves assimétricas partilhadas após uma autenticação bem sucedida entre o *IdP* e a *helper application* usando o protocolo *ZKP* apresentado, e irá ser explicado em mais detalhe na secção 3.2. Assim sendo, quando a autenticação explicada nesta secção termina, a *helper application* tem a possibilidade de decidir se pretende enviar uma chave pública associada ao utilizador para o *IdP*, de forma a que, durante um dado intervalo de tempo, a autenticação possa ser feita de forma mais eficiente[2]. Desta forma, quando os dois intervenientes chegam ao ponto onde $i >= 2N$, o seguinte conjunto de passos poderá acontecer, de forma a definir as credenciais assimétricas do utilizador:

1. A *helper application* começa por criar uma nova chave privada, de 2048 *bits*, para o utilizador que acabou de autenticar, enviando a correspondente chave pública para o *IdP*.
2. O *IdP* recebe este pedido, começando por verificar se possui em *cache* uma resposta *SAML* para este utilizador (como já demonstrado acima, quando o protocolo termina com sucesso, o *IdP* cria e armazena automáticamente a resposta de *SAML* do utilizador, de forma a envia-la ao *SP* quando a *helper application* o requisitar). Em caso afirmativo, cria um identificador de forma aleatória (um *UUID* versão 4), armazena esta chave pública na sua base de dados local, associada ao utilizador autenticado e ao o identificador criado, e com um determinado tempo de vida. Caso tudo tenha ocorrido sem erros até este ponto, o *IdP* responde à *helper application* com o tempo de vida que o par de chaves terá i.e., durante quanto tempo a *helper application* poderá usar este par de chaves para autenticar este utilizador neste *IdP* em específico, usando o protocolo que será explicado na secção 3.2. Para além disso, envia também o identificador, que permitirá à *helper application* que autentique este utilizador usando este par de chaves.
3. Por ultimo, a *helper application* receberá ou uma mensagem de erro, caso o *IdP* de alguma forma não tenha aceitado ou conseguido armazenar a chave pública do utilizador, ou uma mensagem de sucesso com o tempo de vida das credenciais. Neste ultimo caso, irá armazenar localmente a chave privada do utilizador (como será explicado mais tarde na secção 4) associada ao identificador recebido, e a data em que deixará de ser válida.
4. De seguida, a *helper application* termina o protocolo *ZKP*, redirecionando o utilizador para o *IdP*, com o pedido para a identidade deste ser validada ao *SP*, como já explicado anteriormente.

É também de salientar que nesta troca de mensagens para definição das credenciais assimétricas, quer a mensagem de envio da chave pública, quer a mensagem de resposta do *IdP*, são cifradas, usando-se o algoritmo de cifragem simétrica *AES*, com o modo *CBC*. A chave usada é obtida de forma determinística com base nas respostas aos desafios criados durante o protocolo, de acordo com o pseudocodigo apresentado na listagem 4, onde R é obtido da seguinte forma:

$$R = R_0 +\!\!+ R_1 ... +\!\!+ R_i \tag{5}$$

[2]Na nossa implementação, a *helper application* envia sempre a chave pública para o *IdP*. Quando referimos que ela pode escolher, significa que o *IdP* não "obriga"a que ela envie a chave, sendo apenas uma *feature* que ele oferece de forma a oferecer um serviço mais *user friendly* e, simultâneamente, seguro.

```
def asymmetric_upload_derivation_key(R, i, key_size=32):
 result = b''
 for i in range(key_size):
  result += R[(variable*i) % length(R)]

 return result
```

Listagem 4: Pseudocodigo correspondente à geração da chave a ser usada na cifragem das mensagens de definição das credênciais assimétricas.

3.2 Protocolo de autenticação baseado em chaves assimétricas

Como já referido anteriormente, o protocolo *ZKP* não é o mais eficiente, sendo demasiado lento em algumas situações. Desta forma, usá-mos um segundo protocolo para autenticação do utilizador baseado em criptografia assimétrica. Como apresentado na secção 3.1, após uma autenticação bem sucedida com o protocolo *ZKP*, a *helper application* pode escolher criar uma chave privada para o utilizador que acabou de autenticar, enviando a correspondente chave pública para o *IdP* através dum canal seguro, que utiliza uma chave obtida a partir das várias respostas aos desafios durante a execução do protocolo. Tendo em conta este pormenor de como foram criadas e distribuídas as chaves, podemos fazer, desde já, duas assunções:

- Sendo que a *helper application* criou a chave privada do utilizador de forma aleatória e a armazenou localmente, protegida por um segredo (mais sobre o armazenamento das chaves na secção 4), enviando a correspondente chave pública por um canal seguro para o *IdP* (evitando a possibilidade de haver qualquer tipo de *eavesdropping*), podemos afirmar que o *IdP* pode autenticar este utilizador posteriormente, usando a correspondente chave pública que agora possui para verificar a assinatura dum desafio, que só pode ser criada pela utilização da correspondente chave privada que, lógicamente, só a *helper application* tem acesso.
- Sendo que a chave pública foi enviada por um canal seguro para o *IdP*, podemos considerar que não poderia haver qualquer tipo de *eavesdropping*. Assim sendo, futuramente a *helper application* pode também autenticar este *IdP*, bastando para isso usar a sua chave privada para verificar se uma dada mensagem (um desafio) foi cifrada usando a chave pública enviada previamente que, devido à forma como foi enviada, só o *IdP* é que possui. Ou seja, pelo facto de apenas o *IdP* (e claro, a *helper application*) ser a única entidade que tem acesso à chave pública do utilizador, basta numa nova autenticação verificar se este possui a chave pública (através, por exemplo, do envio dum desafio para este cifrar usando a chave), para termos uma prova fidedigna de que o outro interveniente no protocolo de autenticação é, de facto, o *IdP*.

Sumariando estas constatações, após a *helper application* enviar a chave pública do utilizador para o *IdP*, é possível a autenticação futura quer do *IdP*, quer do utilizador a quem pertence o par de chaves, de forma completamente segura.

Posto isto, este protocolo segue o conjunto dos seguintes passos iterativos:

1. Este protocolo é iniciado de forma similar ao *ZKP*, ou seja, inicialmente o *IdP* começa por redirecionar o utilizador para a *helper application*, após ter recebido o pedido *SAML* do *SP*. Neste ponto, a *helper application* pode decidir se quer efetuar a autenticação usando o protocolo *ZKP* ou este em específico, caso possua uma chave privada válida i.e., o *IdP* não impõe a utilização preferencial de um dos protocolos, sendo isso uma escolha dada à *helper application*. Contudo, na nossa solução, ela escolhe sempre usar preferencialmente a autenticação com chaves assimétricas, caso estas sejam válidas, uma vez que é mais eficiente. Deste modo, a *helper application* começa por verificar se possui uma chave privada para o utilizador que se quer autenticar no *IdP* que a contactou, e se a mesma é válida, sendo que, em

caso afirmativo, envia um desafio ao *IdP* (correspondente a um *UUID* versão 4), o nome do utilizador e o identificador associado simultâneamente a esse utilizador e ao par de chaves.

2. O *IdP* recebe o pedido de autenticação por parte da *helper application*, começando por verificar se possui alguma chave pública válida para o utilizador e identificador recebido. Desta verificação, podem haver dois resultados:
   - Ou a verificação resulta numa avaliação negativa. Neste caso, o *IdP* pode não possuir a chave pública para o utilizador e identificador recebidos, respondendo à *helper application* com o código de erro *HTTP 424 Failed Dependency*. Por outro lado, caso possua a chave pedida, mas esta já não seja válida, responde com o código de erro *HTTP 410 Gone*.
   - Ou a verificação resulta numa avaliação afirmativa. Nesta situação, o *IdP* começa por cifrar o desafio recebido, usando a chave pública e envia à *helper application* uma resposta com o desafio cifrado e com um novo desafio (também correspondente a um *UUID* versão 4).
3. A *helper application* recebe a resposta do *IdP* e, de acordo com o tipo da mesma, tem dois comportamentos distintos:
   - Caso a resposta seja de erro, prossegue para uma tentativa de autenticação usando o protocolo *ZKP*, como apresentado na secção 3.1.
   - Caso a resposta não seja de erro, começa por verificar se o desafio enviado ao *IdP* foi cifrado com a chave pública do utilizador (utilizando para esta verificação a correspondente chave privada). Em caso afirmativo, ela passa a confiar no *IdP* e, desta forma, assina o desafio recebido deste usando a sua chave privada. Caso contrário, significa que o *IdP* é uma entidade maliciosa, pelo que a *helper application* cessa o protocolo e apresenta um alerta ao utilizador para o decorrido, como apresentado na figura 3.
4. Caso a *helper application* tenha enviado uma resposta ao *IdP*, este irá verificar a assinatura do desafio previamente enviado, usando a sua chave pública. Mais uma vez, de acordo com o resultado desta validação, podem ocorrer dois eventos distintos:
   - Caso a assinatura seja inválida, o *IdP* irá responder à *helper application* com o código de erro *HTTP 401 Unauthorized*.
   - Caso a assinatura seja válida, o utilizador é autenticado com sucesso, o que significa que o *IdP* cria, como no protocolo *ZKP*, a resposta *SAML* ao pedido *SAML* feito inicialmente pelo *SP*.
5. De acordo com a resposta recebida pelo *IdP*, a *helper application* irá:
   - Caso a resposta seja de erro, a *helper application* inicia o protocolo *ZKP* com o *IdP*.
   - Caso a resposta não seja de erro, a *helper application* irá redirecionar o utilizador para o *IdP*, com um pedido para este o identificar propriamente perante o *SP*, de forma similar ao que acontece no final do protocolo *ZKP* já apresentado.
6. O *IdP*, caso receba um pedido para identificar o utilizador perante o *SP*, irá executar o mesmo passo que executa no final do protocolo *ZKP*, começando por analisar se possui uma resposta de *SAML* já criada para este utilizador. Em caso afirmativo, redireciona o utilizador para o *SP*, com um *POST binding* com a resposta de *SAML*.

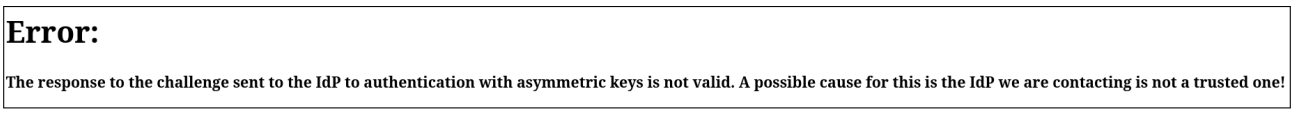

Figura 3: Mensagem de erro apresentada pela *helper application* quando a resposta cifrada com a chave pública do desafio enviado ao *IdP* é inválida.

3.3 Cifragem da comunicação

De forma a evitar que a comunicação, em qualquer um dos protocolos de autenticação, fosse escutada por uma possível entidade maliciosa, ou qualquer outro interveniente que não o *IdP* e a *helper application*, o conteúdo sensível das mensagens enviadas por cada uma das entidades intervenientes é cifrado usando *AES* em modo *CBC*, usando como chave um segredo partilhado entre elas, de 32 *bits*. Este segredo partilhado é criado pelo *IdP* antes de iniciar a autenticação e enviado quando redireciona o utilizador pela primeira vez para a *helper application*, como valor dum dos parâmetros do pedido *GET* (parâmetro *key*). Apesar de parecer que à primeira vista, esta forma de distribuir o segredo partilhado é insegura, uma vez que a chave é enviada em *clear text* no pedido, nós pressupomos que esta comunicação entre o *IdP* e o *browser* do utilizador, num ambiente real, seria protegida com *SSL* i.e., toda a comunicação entre o *browser* do utilizador e o *IdP* seria feita usando *HTTPS*, assegurando que a comunicação da chave era segura.

Desta forma, quer o *IdP*, quer a *helper application* usam uma mesma chave de 32 *bits* secreta para cifrar todos os dados sensíveis enviados. Os dados que consideramos como sendo "não sensíveis"foram o vetor de inicialização, de 16 *bits*, usado para cifrar o conteúdo cifrado de cada mensagem, criado de forma aleatória em cada pedido e em cada resposta feitos, e o *ID* do pedido *SAML* inicialmente feito pelo *SP*, que é usado nas comunicações entre a *helper application* e o *IdP* de forma a se identificarem propriamente durante a autenticação[3]. Todos os outros dados são cifrados usando o método referido. O conteúdo a ser cifrado está contido numa *string* de *JSON*, que é depois cifrada em cada pedido ou em cada resposta, e o texto cifrado resultante é enviado como valor do parâmetro *ciphered* dos pedidos ou das respostas. Depois, quando o outro interveniente recebe o correspondente pedido ou resposta, decifra o valor do parâmetro *ciphered* e obtém a *string* de *JSON* inicial, onde se encontram os dados sensíveis enviados.

É também importante referir que a utilização deste mecanismo de cifragem da comunicação não invalida a cifragem das mensagens que é feita aquando da definição das credenciais assimétricas efetuada no final do protocolo *ZKP*. O que acontece neste caso é que os conteúdos a serem enviados são primeiro cifrados usando a técnica referida no final da secção 3.1 e, de seguida, o texto cifrado é cifrado novamente, desta vez com a chave partilhada entre as duas entidades, como explicado nesta secção. Quando o outro interveniente recebe a mensagem correspondente, começa por decifrar o conteúdo da mesma usando a chave partilhada entre os dois e, por ultimo, a chave obtida a partir dos desafios criados durante o protocolo *ZKP*, como explicado no final da secção 3.1.

[3]Na nossa solução, consideramos que o *IdP* poderia receber vários pedidos de autenticação de múltiplos utilizadores que se pretendiam identificar perante o *SP*. Desta forma, de maneira a que o *IdP* conseguisse comunicar simultâneamente com múltiplos utilizadores, armazena localmente o estado de cada protocolo que tem com cada utilizador no momento, sendo que a *helper application* tem de enviar sempre este *ID* de forma a informar o *IdP* que se trata daquele pedido em especifico.

4 Gestor de credenciais da helper application

Uma vez que a *helper application* é uma aplicação que interage diretamente com o utilizador, permitindo a este autenticar-se perante um dado *IdP*, criamos um gestor de palavras-passe e de chaves na propria *helper application*, de forma a que a autenticação possa ocorrer da forma mais transparente possível para o utilizador, apesar de poderem haver múltiplos *IdPs* que contactem a *helper application* para o autenticar num mesmo *SP*.

Sendo assim, a primeira característica deste gestor é que permite armazenar todas as palavras-passes duma mesma conta de utilizador, cada uma para o respetivo *IdP*, possibilitando que o utilizador (humano) não necessite de se lembrar de todas elas. Isto é ideal, uma vez que podem existir múltiplos *IdPs* que um mesmo *SP* pode contactar para fazer a autenticação do utilizador, sendo que em cada um dos *IdPs*, as credenciais de acesso do utilizador podem não ser iguais. A única palavra-passe que o utilizador se terá de lembrar e utilizar na *helper application* é uma **chave mestra**, que servirá para cifrar todas as chaves e palavras-passes do mesmo.

Posto isto, a *helper application* oferece, desde logo, uma página para os utilizadores (humanos) registarem o seu nome de utilizador, associado a um ou mais *IdPs*, e a respetiva chave mestra. Na figura 4 é possível encontrar um exemplo desta página, onde o utilizador (humano) terá de introduzir o seu nome de utilizador, que estará associado a um ou mais *IdPs* e criar uma chave mestra à sua escolha que irá, como referido, proteger as suas credenciais localmente. É importante também referir que caso o utilizador introduza um nome de utilizador já registado, a *helper application* irá mostrar uma mensagem de erro e não atualizará, como seria expectável, a chave mestra tendo em conta a introduzida no momento.

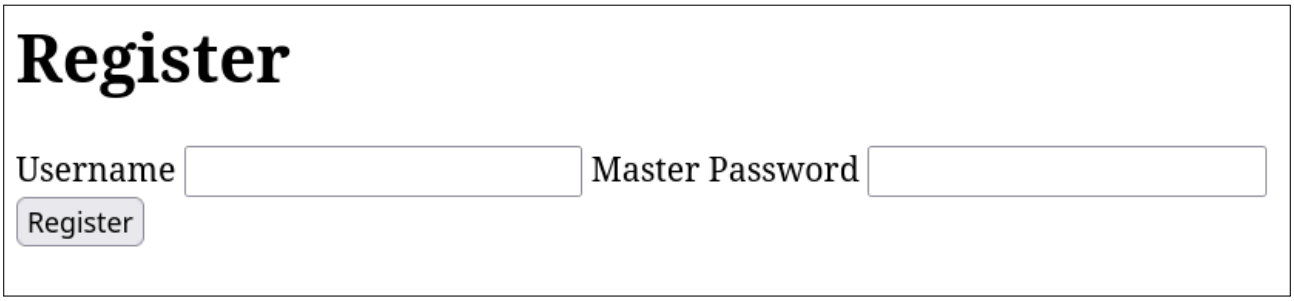

Figura 4: Página da *helper application* que permite registar um novo utilizador, que poderá estar associado a múltiplos *IdPs*, e a correspondente chave mestra.

A *helper application* irá, desta forma, receber o nome do utilizador e a sua chave mestra e armazenar os mesmos num ficheiro local em formato *JSON*. É importante referir que a chave mestra nunca é armazenada em *clear text*, mas sim uma derivação da mesma, usando a função de derivação *scrypt* [5].

Posteriormente, quando o utilizador acede ao *SP* e ocorrem todos os passos explicados nas secções anteriores até ser redirecionado para a *helper application*, esta apresenta-lhe uma página para introduzir o seu nome de utilizador, com o qual se pretende autenticar perante o *IdP* e a correspondente chave mestra, como demonstrado na figura 5. Claro está que, caso o utilizador introduza um nome de utilizador que não tenha sido registado ou uma chave mestra incorreta, a *helper application* irá apresentar uma mensagem de erro e não irá efetuar a autenticação do utilizador.

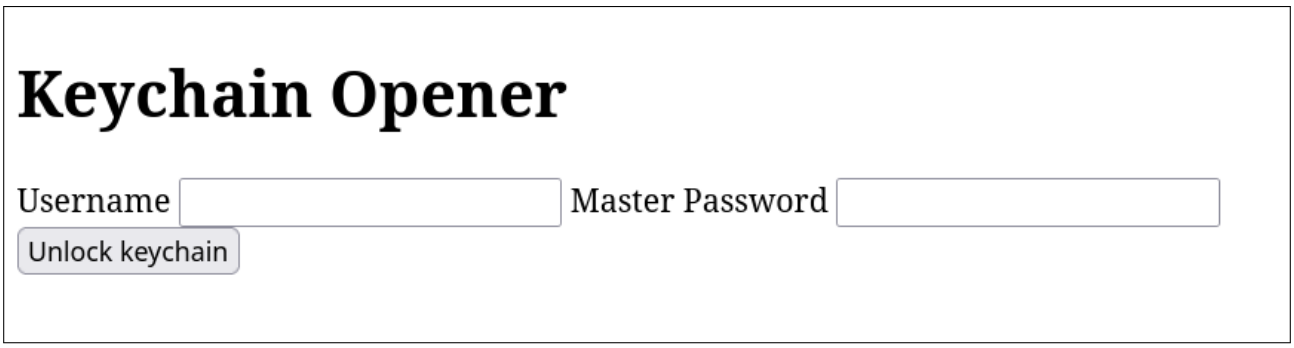

Figura 5: Página da *helper application* que permite ao utilizador autenticar-se localmente, introduzindo o nome do utilizador com o qual pretende ser autenticado perante o *IdP* e a sua chave mestra.

Caso o utilizador (humano) introduza o nome de utilizador e a chave mestra corretamente, um de dois eventos pode ocorrer:

- Caso a *helper application* não possua armazenada a palavra-passe deste utilizador em relação ao *IdP* que a está a contactar (o que possívelmente significa que é a primeira vez que esta *helper application* está a autenticar este utilizador perante este *IdP*), a *helper application* irá pedir ao utilizador (humano) que introduza a palavra-passe associada ao nome de utilizador e ao *IdP* que está a contactar, de forma a poder prosseguir com a autenticação. Na figura 6 é apresentada a página que a *helper application* envia ao utilizador neste caso. Após o utilizador submeter esta palavra-passe, a *helper application* irá armazenar a mesma num ficheiro local, de forma criptográficamente segura, da seguinte maneira (caso o protocolo *ZKP* feito posteriormente seja bem sucedido):
  1. A *helper application* irá primeiro criar um vetor de inicialização (*iv*), de 16 *bits*, de forma aleatória (usando a biblioteca de *Python* "*os.urandom*").
  2. De seguida, irá também criar um valor de *salt* de 16 *bits* aleatóriamente, da mesma maneira. Posto isto, irá derivar uma chave de *AES* de 32 *bits* usando a função de derivação *PBKDF2*, a partir do valor da chave mestra e deste *salt* computado.
  3. Finalmente, irá cifrar o valor da palavra-passe que o utilizador introduziu para autenticação neste *IdP*, usando o algoritmo *AES* com o modo *CBC* com a chave e o vetor de inicialização calculados nos passos anteriores[4].
  4. Por fim, cria um ficheiro local, com o nome na forma "*<nome do utilizador>_secret_-<nome do IdP em Base64>*", onde irá armazenar os valores do vetor de inicialização, do *salt* e do texto cifrado obtidos nos passos anteriores, em formato binário.
- Caso a *helper application* possua armazenada a palavra-passe deste utilizador em relação ao *IdP* que a está a contactar, irá ler o conteúdo do ficheiro "*<nome do utilizador>_secret_-<nome do IdP em Base64>*"e, usando o vetor de inicialização e o *salt* nele armazenados, irá primeiro produzir uma derivação da chave mestra, usando o valor do *salt*, para obter uma chave de *AES* válida e, usando esta e o vetor de inicialização, irá decifrar o conteúdo cifrado deste ficheiro que, como já explicado no ponto anterior, corresponde à palavra-passe deste utilizador neste *IdP*, préviamente armazenada pela *helper application* de forma segura.

[4] Sendo que a palavra-passe pode não ter um comprimento múltiplo do tamanho mínimo do *block size* aceito pelo *AES*, foi usado o método de *padding*, de forma a que o texto de entrada tivesse um tamanho que fosse de encontro a este requerimento. Como algoritmo de *padding*, usamos o *PKCS7*.

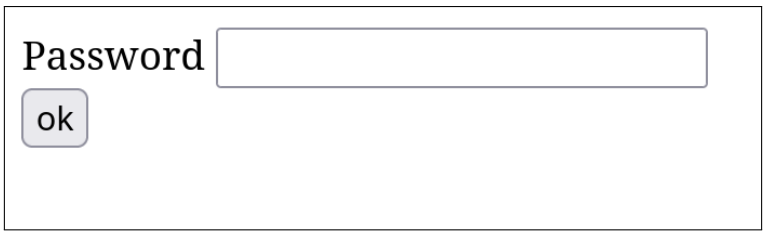

Figura 6: Página da *helper application* enviada ao utilizador quando esta não possui a palavra-passe do mesmo para o autenticar perante o *IdP* que a contactou.

Qualquer que seja o cenário que tenha ocorrido no passo anterior, a *helper application* irá obter uma palavra-passe que irá utilizar para tentar efetuar o protocolo *ZKP*, tal como explicado na secção 3.1.

Como também já apresentado anteriormente, caso o protocolo *ZKP* seja bem sucedido, a *helper application* criada irá tentar criar uma chave privada para este utilizador, enviando a correspondente chave publica ao *IdP*, de forma a permitir que a autenticação possa ser efetuada de forma mais eficiente em futuros acessos. Desta forma, este gestor também tem um mecanismo para gerir as chaves privadas localmente de forma segura. Assim sendo, quando o protocolo *ZKP* termina com sucesso e a *helper application* cria a referida chave privada, irão ocorrer os seguintes passos, que garantem que a mesma seja armazenada de forma segura:

1. Primeiro, a *helper application* irá criar um valor de *salt*, de 16 *bits*, de forma aleatória.
2. De seguida, irá derivar uma chave de *AES* de 32 *bits*, usando a função de derivação *PBKDF2*, a partir da concatenação entre a chave mestra e a palavra-passe utilizada para fazer a autenticação com o *IdP*, usando o valor de *salt* obtido no passo anterior.
3. Usando esta chave, irá serializerar a chave privada com o *encoding PEM*, no formato *Tradition OpenSSL*, cifrada usando a chave obtida no passo anterior.
4. Por fim, cria um ficheiro, cujo nome apresenta o formato "*<nome do utilizador>_<nome do IdP em Base64>.pem*", onde armazena o identificador do utilizador, o tempo de vida da chave[5], o valor do *salt* usado para derivar a chave usada para cifrar a chave privada e o valor da chave privada cifrada.

Desta forma, é assegurada a segurança no armazenamento e acesso ao valor da chave privada. Posteriormente, quando o utilizador pretender autenticar-se novamente usando este nome de utilizador e neste *IdP*, caso a chave ainda seja válida, a *helper application* implementada irá automáticamente obter o valor da chave privada armazenada (sendo que, claro, primeiro irá pedir ao utilizador a sua chave mestra, de seguida irá abrir o ficheiro "*<nome do utilizador>_secret_<nome do IdP em Base64>*"e obter a palavra-passe do mesmo para este *IdP* e, por fim, com estes dois valores, decifrar o valor do texto cifrado que contém a chave privada armazenado, como já apresentado, no ficheiro "*<nome do utilizador>_<nome do IdP em Base64>.pem*"), permitindo fazer a autenticação com credenciais assimétricas como apresentado na secção 3.2. Caso a chave privada que possui já não seja válida, simplesmente irá proceder à autenticação do utilizador através do protocolo *ZKP* e, no final, irá criar uma nova chave privada para este *IdP* e atualizar o conteúdo do correspondente ficheiro local que contém a chave.

[5] A origem destes dois valores é explicada no final da secção 3.1 e na secção 3.2.

5 Referências

[1] “Single sign-on: What is it & how does it work?” [Acedido em 06-06-2021]. [Online]. Available: https://www.onelogin.com/learn/how-single-sign-on-works

[2] IdentityPython, “Pysaml2 - saml2 in python,” [Acedido em 03-06-2021]. [Online]. Available: https://github.com/IdentityPython/pysaml2

[3] Onelogin, “Onelogin’s saml python toolkit (compatible with python3),” [Acedido em 03-06-2021]. [Online]. Available: https://github.com/onelogin/python3-saml

[4] “Security assertion markup language (saml) v2.0 technical overview,” pp. 27–30, 2008, [Acedido em 03-06-2021]. [Online]. Available: https://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0.pdf

[5] “The scrypt key derivation function and encryption utility,” [Acedido em 06-06-2021]. [Online]. Available: https://www.tarsnap.com/scrypt.html