Key management will be an important aspect of the new electronic protected health information (e-PHI). Key management is often considered the most difficult part of designing a cryptosystem.

Choose a fictitious or an actual organization. The idea is to provide an overview of the current state of enterprise key management for Superior Health Care.

Review these authentication resources to learn about [authentication](https://leocontent.umgc.edu/content/scor/uncurated/cst/2215-cst620/learning-topic-list/authentication.html?ou=622351) and the characteristics of key management.

Provide a high-level, top-layer network view (diagram) of the systems in Superior Health Care. The diagram can be a bubble chart or Visio drawing of a simple network diagram with servers. Conduct independent research to identify a suitable network diagram.

Read these resources on [data at rest](https://leocontent.umgc.edu/content/scor/uncurated/cst/2215-cst620/learning-topic-list/data-at-rest.html?ou=622351), data in use, and [data in motion](https://leocontent.umgc.edu/content/scor/uncurated/cst/2215-cst620/learning-topic-list/data-in-motion.html?ou=622351).

Identify data at rest, data in use, and data in motion as it could apply to your organization. Start by focusing on where data are stored and how data are accessed.

Review these resources on [insecure handling](https://leocontent.umgc.edu/content/scor/uncurated/cst/2215-cst620/learning-topic-list/insecure-handling.html?ou=622351)and identify areas where insecure handling may be a concern for your organization.

Incorporate this information in your key management plan. User Datagram Protocol (UDP)

[Print](https://leocontent.umgc.edu/content/scor/uncurated/cst/2215-cst620/learning-topic-list/user-datagram-protocol--udp-.html)

User datagram protocol (UDP) is a connectionless transport layer protocol that requires no handshaking process. Unlike transmission control protocol (TCP), UDP transmits data without setting up a dedicated connection or verifying the transmission with the receiver. Consequently, there is no guarantee that data packets are delivered in the right order, or delivered at all.

However, UDP has low latency and is suitable for time-critical transmission where speed is more important than reliability. Common applications of UDP include Domain Name System (DNS) and Simple Network Management Protocol (SNMP).

Learning Resource

[Print](https://leocontent.umgc.edu/content/scor/uncurated/cst/2215-cst620/learning-resource-list/how-to-authenticate-users-with-api-keys.html)

How to Authenticate Users With API Keys

Nowadays, it's quite usual to authenticate the user via an API key (when developing a web service, for instance). The API key is provided for every request and is passed as a query string parameter or via an HTTP header.

The API Key Authenticator

Authenticating a user based on the request information should be done via a pre authentication mechanism. The SimplePreAuthenticatorInterface allows you to implement such a scheme really easily.

Your exact situation may differ, but in this example, a token is read from an apikey query parameter, the proper username is loaded from that value, and then a user object is created:

// src/AppBundle/Security/ApiKeyAuthenticator.php

namespace AppBundle\Security;

use Symfony\Component\HttpFoundation\Request;

use Symfony\Component\Security\Core\Authentication\Token\PreAuthenticatedToken;

use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;

use Symfony\Component\Security\Core\Exception\AuthenticationException;

use Symfony\Component\Security\Core\Exception\CustomUserMessageAuthenticationException;

use Symfony\Component\Security\Core\Exception\BadCredentialsException;

use Symfony\Component\Security\Core\User\UserProviderInterface;

use Symfony\Component\Security\Http\Authentication\SimplePreAuthenticatorInterface;

class ApiKeyAuthenticator implements SimplePreAuthenticatorInterface

{

public function createToken(Request $request, $providerKey)

{

// look for an apikey query parameter

$apiKey = $request->query->get('apikey');

// or if you want to use an "apikey" header, then do something like this:

// $apiKey = $request->headers->get('apikey');

if (!$apiKey) {

throw new BadCredentialsException();

// or to just skip api key authentication

// return null;

}

return new PreAuthenticatedToken(

'anon.',

$apiKey,

$providerKey

);

}

public function supportsToken(TokenInterface $token, $providerKey)

{

return $token instanceof PreAuthenticatedToken && $token->getProviderKey() === $providerKey;

}

Once you've configured everything, you'll be able to authenticate by adding an apikey parameter to the query string, like http://example.com/api/foo?apikey=37b51d194a7513e45b56f6524f2d51f2.

The authentication process has several steps, and your implementation will probably differ:

**createToken**

Early in the request cycle, Symfony calls createToken(). Your job here is to create a token object that contains all of the information from the request that you need to authenticate the user (e.g., the apikey query parameter). If that information is missing, throwing a BadCredentialsException will cause authentication to fail. You might want to return null instead to just skip the authentication, so Symfony can fall back to another authentication method, if any.

In case you return null from your createToken() method, be sure to enable anonymous in your firewall. This way you'll be able to get an AnonymousToken.

**supportsToken**

After Symfony calls createToken(), it will then call supportsToken() on your class (and any other authentication listeners) to figure out who should handle the token. This is just a way to allow several authentication mechanisms to be used for the same firewall (that way, you can for instance first try to authenticate the user via a certificate or an API key and fall back to a form login).

Mostly, you just need to make sure that this method returns true for a token that has been created by createToken(). Your logic should probably look exactly like this example.

**authenticateToken**

The $userProvider can be any user provider. In this example, the $apiKey is used to somehow find the username for the user. This work is done in a getUsernameForApiKey() method, which is created entirely custom for this use case (i.e., this isn't a method that's used by Symfony's core user provider system).

The $userProvider might look something like this:

1. First, you use the $userProvider to somehow look up the $username that corresponds to the $apiKey;
2. Second, you use the $userProvider again to load or create a User object for the $username;
3. Finally, you create an *authenticated token* (i.e., a token with at least one role) that has the proper roles and the user object attached to it.

The goal is ultimately to use the $apiKey to find or create a User object. *How* you do this (e.g., query a database) and the exact class for your user object may vary. Those differences will be most obvious in your user provider.

The User Provider

The $userProvider can be any user provider. In this example, the $apiKey is used to somehow find the username for the user. This work is done in a getUsernameForApiKey() method, which is created entirely custom for this use case (i.e., this isn't a method that's used by Symfony's core user provider system).

The $userProvider might look something like this:

// src/AppBundle/Security/ApiKeyUserProvider.php

namespace AppBundle\Security;

use Symfony\Component\Security\Core\User\UserProviderInterface;

use Symfony\Component\Security\Core\User\User;

use Symfony\Component\Security\Core\User\UserInterface;

use Symfony\Component\Security\Core\Exception\UnsupportedUserException;

class ApiKeyUserProvider implements UserProviderInterface

{

public function getUsernameForApiKey($apiKey)

{

// Look up the username based on the token in the database, via

// an API call, or do something entirely different

$username = ...;

return $username;

}

public function loadUserByUsername($username)

{

return new User(

$username,

null,

// the roles for the user - you may choose to determine

// these dynamically somehow based on the user

array('ROLE\_API')

);

}

public function refreshUser(UserInterface $user)

{

// this is used for storing authentication in the session

// but in this example, the token is sent in each request,

// so authentication can be stateless. Throwing this exception

// is proper to make things stateless

throw new UnsupportedUserException();

}

public function supportsClass($class)

{

return 'Symfony\Component\Security\Core\User\User' === $class;

}

}

Now register your user provider as a service:

* YAML

# app/config/services.yml

services:

api\_key\_user\_provider:

class: AppBundle\Security\ApiKeyUserProvider

* XML

<!-- app/config/services.xml -->

<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://symfony.com/schema/dic/services

http://symfony.com/schema/dic/services/services-1.0.xsd">

<services>

<!-- ... -->

<service id="api\_key\_user\_provider"

class="AppBundle\Security\ApiKeyUserProvider" />

</services>

</container>

* PHP

// app/config/services.php

// ...

$container

->register('api\_key\_user\_provider', 'AppBundle\Security\ApiKeyUserProvider');

The logic inside getUsernameForApiKey() is up to you. You may somehow transform the API key (e.g., 37b51d) into a username (e.g., jondoe) by looking up some information in a "token" database table.

The same is true for loadUserByUsername(). In this example, Symfony's core User class is simply created. This makes sense if you don't need to store any extra information on your user object (e.g., firstName). But if you do, you may instead have your *own* user class which you create and populate here by querying a database. This would allow you to have custom data on the Userobject.

Finally, just make sure that supportsClass() returns true for user objects with the same class as whatever user you return in loadUserByUsername().

If your authentication is stateless like in this example (i.e., you expect the user to send the API key with every request and so you don't save the login to the session), then you can simply throw the UnsupportedUserException exception in refreshUser().

Handling Authentication Failure

In order for your ApiKeyAuthenticator to correctly display a 401 HTTP status when either bad credentials or authentication fails you will need to implement the AuthenticationFailureHandlerInterface on your authenticator. This will provide a method onAuthenticationFailure that you can use to create an error response.

// src/AppBundle/Security/ApiKeyAuthenticator.php

namespace AppBundle\Security;

use Symfony\Component\Security\Core\Exception\AuthenticationException;

use Symfony\Component\Security\Http\Authentication\AuthenticationFailureHandlerInterface;

use Symfony\Component\Security\Http\Authentication\SimplePreAuthenticatorInterface;

use Symfony\Component\HttpFoundation\Response;

use Symfony\Component\HttpFoundation\Request;

class ApiKeyAuthenticator implements SimplePreAuthenticatorInterface, AuthenticationFailureHandlerInterface

{

// ...

public function onAuthenticationFailure(Request $request, AuthenticationException $exception)

{

return new Response(

// this contains information about \*why\* authentication failed

// use it, or return your own message

strtr($exception->getMessageKey(), $exception->getMessageData()),

401

);

}

}

Configuration

Once you have your ApiKeyAuthenticator all set up, you need to register it as a service and use it in your security configuration (e.g., security.yml). First, register it as a service.

* YAML

# app/config/config.yml

services:

# ...

apikey\_authenticator:

class: AppBundle\Security\ApiKeyAuthenticator

public: false

* XML

<!-- app/config/config.xml -->

<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://symfony.com/schema/dic/services

http://symfony.com/schema/dic/services/services-1.0.xsd">

<services>

<!-- ... -->

<service id="apikey\_authenticator"

class="AppBundle\Security\ApiKeyAuthenticator"

public="false" />

</services>

</container>

* PHP

// app/config/config.php

use Symfony\Component\DependencyInjection\Definition;

use Symfony\Component\DependencyInjection\Reference;

// ...

$definition = new Definition('AppBundle\Security\ApiKeyAuthenticator');

$definition->setPublic(false);

$container->setDefinition('apikey\_authenticator', $definition);

Now, activate it and your custom user provider in the firewalls section of your security configuration using the simple\_preauth and provider keys respectively:

* YAML

# app/config/security.yml

security:

# ...

firewalls:

secured\_area:

pattern: ^/api

stateless: true

simple\_preauth:

authenticator: apikey\_authenticator

provider: api\_key\_user\_provider

providers:

api\_key\_user\_provider:

id: api\_key\_user\_provider

* XML

<!-- app/config/security.xml -->

<?xml version="1.0" encoding="UTF-8"?>

<srv:container xmlns="http://symfony.com/schema/dic/security"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:srv="http://symfony.com/schema/dic/services"

xsi:schemaLocation="http://symfony.com/schema/dic/services

http://symfony.com/schema/dic/services/services-1.0.xsd">

<config>

<!-- ... -->

<firewall name="secured\_area"

pattern="^/api"

stateless="true"

provider="api\_key\_user\_provider"

>

<simple-preauth authenticator="apikey\_authenticator" />

</firewall>

<provider name="api\_key\_user\_provider" id="api\_key\_user\_provider" />

</config>

</srv:container>

* PHP

// app/config/security.php

// ..

$container->loadFromExtension('security', array(

'firewalls' => array(

'secured\_area' => array(

'pattern' => '^/api',

'stateless' => true,

'simple\_preauth' => array(

'authenticator' => 'apikey\_authenticator',

),

'provider' => 'api\_key\_user\_provider',

),

),

'providers' => array(

'api\_key\_user\_provider' => array(

'id' => 'api\_key\_user\_provider',

),

),

));

If you have defined access\_control, make sure to add a new entry:

* YAML

# app/config/security.yml

security:

# ...

access\_control:

- { path: ^/api, roles: ROLE\_API }

* XML

<!-- app/config/security.xml -->

<?xml version="1.0" encoding="UTF-8"?>

<srv:container xmlns="http://symfony.com/schema/dic/security"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:srv="http://symfony.com/schema/dic/services"

xsi:schemaLocation="http://symfony.com/schema/dic/services

http://symfony.com/schema/dic/services/services-1.0.xsd">

<rule path="^/api" role="ROLE\_API" />

</srv:container>

* PHP

// app/config/security.php

$container->loadFromExtension('security', array(

'access\_control' => array(

array(

'path' => '^/api',

'role' => 'ROLE\_API',

),

),

));

That's it! Now your ApiKeyAuthenticator should be called at the beginning of each request and your authentication process will take place.

The stateless configuration parameter prevents Symfony from trying to store the authentication information in the session, which isn't necessary since the client will send the apikey on each request. If you *do* need to store authentication in the session, keep reading!

Storing Authentication in the Session

So far, this entry has described a situation where some sort of authentication token is sent on every request. But in some situations (like an OAuth flow), the token may be sent on only *one*request. In this case, you will want to authenticate the user and store that authentication in the session so that the user is automatically logged in for every subsequent request.

To make this work, first remove the stateless key from your firewall configuration or set it to false:

* YAML

# app/config/security.yml

security:

# ...

firewalls:

secured\_area:

pattern: ^/api

stateless: false

simple\_preauth:

authenticator: apikey\_authenticator

provider: api\_key\_user\_provider

providers:

api\_key\_user\_provider:

id: api\_key\_user\_provider

* XML

<!-- app/config/security.xml -->

<?xml version="1.0" encoding="UTF-8"?>

<srv:container xmlns="http://symfony.com/schema/dic/security"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:srv="http://symfony.com/schema/dic/services"

xsi:schemaLocation="http://symfony.com/schema/dic/services

http://symfony.com/schema/dic/services/services-1.0.xsd">

<config>

<!-- ... -->

<firewall name="secured\_area"

pattern="^/api"

stateless="false"

provider="api\_key\_user\_provider"

>

<simple-preauth authenticator="apikey\_authenticator" />

</firewall>

<provider name="api\_key\_user\_provider" id="api\_key\_user\_provider" />

</config>

</srv:container>

* PHP

// app/config/security.php

// ..

$container->loadFromExtension('security', array(

'firewalls' => array(

'secured\_area' => array(

'pattern' => '^/api',

'stateless' => false,

'simple\_preauth' => array(

'authenticator' => 'apikey\_authenticator',

),

'provider' => 'api\_key\_user\_provider',

),

),

'providers' => array(

'api\_key\_user\_provider' => array(

'id' => 'api\_key\_user\_provider',

),

),

));

Even though the token is being stored in the session, the credentials—in this case the API key (i.e., $token->getCredentials())—are not stored in the session for security reasons. To take advantage of the session, update ApiKeyAuthenticator to see if the stored token has a valid user object that can be used:

// src/AppBundle/Security/ApiKeyAuthenticator.php

// ...

class ApiKeyAuthenticator implements SimplePreAuthenticatorInterface

{

// ...

public function authenticateToken(TokenInterface $token, UserProviderInterface $userProvider, $providerKey)

{

if (!$userProvider instanceof ApiKeyUserProvider) {

throw new \InvalidArgumentException(

sprintf(

'The user provider must be an instance of ApiKeyUserProvider (%s was given).',

get\_class($userProvider)

)

);

}

$apiKey = $token->getCredentials();

$username = $userProvider->getUsernameForApiKey($apiKey);

// User is the Entity which represents your user

$user = $token->getUser();

if ($user instanceof User) {

return new PreAuthenticatedToken(

$user,

$apiKey,

$providerKey,

$user->getRoles()

);

}

if (!$username) {

// this message will be returned to the client

throw new CustomUserMessageAuthenticationException(

sprintf('API Key "%s" does not exist.', $apiKey)

);

}

$user = $userProvider->loadUserByUsername($username);

return new PreAuthenticatedToken(

$user,

$apiKey,

$providerKey,

$user->getRoles()

);

}

// ...

}

Storing authentication information in the session works like this:

1. At the end of each request, Symfony serializes the token object (returned from authenticateToken()), which also serializes the user object (since it's set on a property on the token).
2. On the next request the token is deserialized and the deserialized User object is passed to the refreshUser() function of the user provider.

The second step is the important one: Symfony calls refreshUser() and passes you the user object that was serialized in the session. If your users are stored in the database, then you may want to re query for a fresh version of the user to make sure it's not out-of-date. But regardless of your requirements, refreshUser() should now return the user object:

// src/AppBundle/Security/ApiKeyUserProvider.php

// ...

class ApiKeyUserProvider implements UserProviderInterface

{

// ...

public function refreshUser(UserInterface $user)

{

// $user is the User that you set in the token inside authenticateToken()

// after it has been deserialized from the session

// you might use $user to query the database for a fresh user

// $id = $user->getId();

// use $id to make a query

// if you are \*not\* reading from a database and are just creating

// a User object (like in this example), you can just return it

return $user;

}

}

You'll also want to make sure that your User object is being serialized correctly. If your User object has private properties, PHP can't serialize those. In this case, you may get back a user object that has a null value for each property.

Only Authenticating for Certain URLs

This entry has assumed that you want to look for the apikey authentication on *every* request. But in some situations (like an OAuth flow), you only really need to look for authentication information once the user has reached a certain URL (e.g., the redirect URL in OAuth).

Fortunately, handling this situation is easy: just check to see what the current URL is before creating the token in createToken():

// src/AppBundle/Security/ApiKeyAuthenticator.php

// ...

use Symfony\Component\Security\Http\HttpUtils;

use Symfony\Component\HttpFoundation\Request;

class ApiKeyAuthenticator implements SimplePreAuthenticatorInterface

{

protected $httpUtils;

public function \_\_construct(HttpUtils $httpUtils)

{

$this->httpUtils = $httpUtils;

}

public function createToken(Request $request, $providerKey)

{

// set the only URL where we should look for auth information

// and only return the token if we're at that URL

$targetUrl = '/login/check';

if (!$this->httpUtils->checkRequestPath($request, $targetUrl)) {

return;

}

// ...

}

}

This uses the handy HttpUtils class to check if the current URL matches the URL you're looking for. In this case, the URL (/login/check) has been hard-coded in the class, but you could also inject it as the second constructor argument.

Next, just update your service configuration to inject the security.http\_utils service:

* YAML

# app/config/config.yml

services:

# ...

apikey\_authenticator:

class: AppBundle\Security\ApiKeyAuthenticator

arguments: ["@security.http\_utils"]

public: false

* XML

<!-- app/config/config.xml -->

<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://symfony.com/schema/dic/services

http://symfony.com/schema/dic/services/services-1.0.xsd">

<services>

<!-- ... -->

<service id="apikey\_authenticator"

class="AppBundle\Security\ApiKeyAuthenticator"

public="false"

>

<argument type="service" id="security.http\_utils" />

</service>

</services>

</container>

* PHP

// app/config/config.php

use Symfony\Component\DependencyInjection\Definition;

use Symfony\Component\DependencyInjection\Reference;

// ...

$definition = new Definition(

'AppBundle\Security\ApiKeyAuthenticator',

array(

new Reference('security.http\_utils')

)

);

$definition->setPublic(false);

$container->setDefinition('apikey\_authenticator', $definition);

That's it! Have fun!

# app/config/config.yml

services:

# ...

apikey\_authenticator:

class: AppBundle\Security\ApiKeyAuthenticator

arguments: ["@security.http\_utils"]

public: false

<!-- app/config/config.xml -->

<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://symfony.com/schema/dic/services

http://symfony.com/schema/dic/services/services-1.0.xsd">

<services>

<!-- ... -->

<service id="apikey\_authenticator"

class="AppBundle\Security\ApiKeyAuthenticator"

public="false"

>

<argument type="service" id="security.http\_utils" />

</service>

</services>

</container>

// app/config/config.php

use Symfony\Component\DependencyInjection\Definition;

use Symfony\Component\DependencyInjection\Reference;

// ...

$definition = new Definition(

'AppBundle\Security\ApiKeyAuthenticator',

array(

new Reference('security.http\_utils')

)

);

$definition->setPublic(false);

$container->setDefinition('apikey\_authenticator', $definition);

Licenses and Attributions

[How to Authenticate Users with API Keys](http://symfony.com/doc/current/security/api_key_authentication.html) from *Symfony* is available under a [Creative Commons Attribution-ShareAlike 3.0 Unported](https://creativecommons.org/licenses/by-sa/3.0/) license. UMGC has modified this work and it is available under the original license.

© 2022 University of Maryland Global Campus

All links to external sites were verified at the time of publication. UMGC is not responsible for the validity or integrity of information located at external sites.