Delegated Authentication

Overview

Opmantek Applications released after 22 Feb 2017 support a new authentication method called `token`, which offers delegated authentication: an external party can pre-authenticate a user, who can access the Opmantek applications without having to log in with username and password.

How does it work?

The core idea is broadly similar to Kerberos: the Opmantek application and the external party share a cryptographic key, and thus a trust relationship is established. When the external party is satisfied that a user should be (pre-)authenticated for Opmantek apps, then it uses that shared key to create a 'token' or 'ticket', which the user can present to the Opmantek application in lieu of logging in interactively; if the ticket verifies as valid, the Opmantek application accepts the user as authenticated and logged in.

Configuration

To enable token authentication, a few configuration settings must be added to `/usr/local/omk/conf/opCommon.nmis` for legacy modules or `/usr/local/omk/conf/opCommon.json` for current:

1. One or more shared keys must be set up in `auth_token_key`,
2. optionally, the maximum validity for tokens may be specified in `auth_token_maxage`,
3. and finally, the authentication method `token` must be added as one of the three supported authentication methods.

Once these changes are made, the Opmantek daemon must be restarted (`sudo service omkd restart`) to activate them.

Here is an example `opCommon.nmis` snippet, showing just the relevant items:

```
'authentication' => {
# skipped other stuff...
    'auth_method_1' => 'token',
    'auth_method_2' => 'htpasswd',
    'auth_method_3' => '',
    'auth_token_key' => [ 'whateverSuitsU!', 'ForAnotherTrustedTP', ],
    'auth_token_maxage' => 100,
```

The `auth_token_key` list specifies which shared keys should be accepted and tried in sequence. Setting just one key is ok. If you have multiple external parties that you want to delegate authentication duties to, then it is recommended that you give each their own key.

The `auth_token_maxage` setting must be a positive number, and defines how long a token remains valid after creation (in seconds). If not present, the default of 300 seconds is used.

The `token` authentication method is active if and only if one of the `auth_method_1`, 2 or 3 entries is set to `token`. Please note that it is not relevant which of the three is set to token; the `token` method is ignored when the normal username and password login form is used, and conversely the other methods are ignored if the token access URL is visited (see next section).

Generating a Token

Opmantek applications released after May 2017 ship with a small token generator helper in `/usr/local/omk/bin/generate_auth_token.pl` (and also compiled into a standalone `.exe` program, in the same location).

This token generator must be passed the shared key to use, and the username to generate a token for. The encrypted token is created with and for that username and contains the current time (in UTC). Finally, the generator prints the resulting token, as in the following example:

```
$ /usr/local/omk/bin/generate_auth_token.pl 'whateverSuitsU!' operator
53616c7465645f5fd95eadb039692ea599441f8089daf1d7f04ab9ccf479e37fb3afda85b3044f4cde5b15844e9be616
```

Token length varies depending on the username, and each execution does create a different token. Please note that if your shared key contains shell metacharacters (like "!" in the example above) you will have to quote them with single quotes when passing them to the token generator.

Logging in with a Token

To use a token with an Opmantek application, the token generating party should direct the user to a URL in the following format: `http://<yourserver>/omk/<applicationkey>/login/<token>`.

As a concrete example, to access opCharts with the token from before we'd use
`https://testsystem1.opmantek.com/omk/opCharts/login/53616c7465645f5fd95eadb039692ea599441f8089daf1d7f04ab9ccf479e37fb3afda85b3044f4cde5b15844e9be616`

If your system is configured for secure HTTP then it's fine to use `https://`. Token authentication works for all commercial Opmantek applications (e.g. application keys `opEvents`, `opConfig` and so on).

When the user accesses this URL using their browser, the authentication subsystem detects the presence of a token and attempts to verify it. If a suitable shared key was configured on the receiving system, and if the token could be decrypted and is not too old, then authentication succeeds, suitable cookies are created and returned, and the user is redirected to the main page for the given application.
If the token is invalid, the user is shown the classic login form, with a suitable error message.

If you need to direct the user to a particular page rather than their Default page/Dashboard you can extend the authentication URL with "?redirect_url=" for example with the token above we can direct someone directly to the topn page as follows:

`https://testsystem1.opmantek.com/omk/opCharts/login/53616c7465645f5fd95eadb039692ea599441f8089daf1d7f04ab9ccf479e37fb3afda85b3044f4cde5b15844e9be616?redirect_url=/omk/opCharts/topn`

Once the client has accessed the first page, they have been issued auth cookies and all standard URLs work without the token string in the URL. You will want to consider how the user is handled to re-authenticate them if the session expires.

Using the token based authentication in the header

We can make API requests against the Opmantek product by passing your generated token within the header of your request.

```
Authorization: Token <data>
```

Token Content and Interoperability Notes

The "payload" data of the token consists of

- the current time (in UNIX seconds since 1970, timezone UTC) as a string (e.g. "1487738312")
- a single space character,
- and the respective username (which must contain printable ascii characters only; e.g. "nmis" or "john doe").

This data is then encrypted symmetrically using the shared key, with the following parameters:

- AES with 128-bit key size in CBC mode,
- block-padded to the normal 128-bit block size using the standard PKCS#5 padding format,
- with OpenSSL-compatible salted header format,
- and using an OpenSSL-compatible password -based key and IV derivation function (PBKDF).
  Please note that the shared `auth_token_key` is used as passphrase to derive the actual key, not as a literal binary 128-bit key.

The resulting encrypted data is binary, and must be encoded for use as a URL component.
The encoding is a trivial hex-encoding: each binary byte is replaced by its representation as two hexadecimal digits.

Limitations

- The token authentication system does not support locking out users after N unsuccessful login attempts.

- As the token contains the current time at the creating system and is valid for a limited time only, reasonably precise time synchronisation is critical for this method to work.
  If a token could be decrypted but was rejected because it was deemed too old, then a suitable log entry is written to the `auth.log`.
- Tokens are not single-use: a token works any number of times as long as it is presented within the configured validity period.

Code Examples for Token Generation

Perl

The following block contains essentially the same code as the token generator shipped as `bin/generate_auth_token.pl`:

```
#!/usr/bin/perl
use strict;
use Crypt::CBC;

my ($key, $username, $tokentime) = @ARGV;
die "Usage: $0 <key> <username> [timestamp]
key: passphrase of arbitrary length.
timestamp: optional, default: now\n"
        if (!$key or !$username or (defined $tokentime && !int($tokentime)));
$tokentime ||= time;

#  what goes into the token? the token time stamp (in unix-seconds, UTC),
# as a plain string, followed by exactly one space and the username.
my $plain = $tokentime." ".$username;

# defaults: RFC2898/pkcs#5 padding, openssl-compatible salted header mode,
# and openssl-compatible key derivation function (PBKDF) -
# see https://www.openssl.org/docs/man1.1.0/crypto/EVP_BytesToKey.html
# but crypt::cbc's default keysize is an incompatible 64 bits
my $engine = Crypt::CBC->new(-key => $key,
                                                        -cipher => "Rijndael",
                                                        -keysize => 128/8);
my $crypted = $engine->encrypt_hex($plain);

print $crypted,"\n";
exit 0;
```

Shell using the OpenSSL CLI

The following small shell script requires the `openssl` command line tool and `hexdump` to perform the token generation (with the key being the first argument, username second):

```
#!/bin/sh
KEY=$1
USER=$2
TEMPFILE=`mktemp /tmp/gentoken.XXXXXX`
NOW=`date +%s`
echo -n "$NOW $USER" > $TEMPFILE
# see man enc: -salt -e are default, could be omitted;
# openssl requires a real file as input, so we need a temp file
# hexdump converts the binary bytes into their hex representation
openssl aes-128-cbc -in $TEMPFILE -salt -e -pass "pass:$KEY" | \
        hexdump -v -e '/1 "%02x"'
echo
rm -f $TEMPFILE
exit 0
```

Python

Python's pycrypto module should contain everything required, except the OpenSSL-specific PBKDF which can be found here.