JSON Web Token Tools

OA4MP contains an interpreter that allows for the creation, printing, validation, signing etc. of JWTs (JSON Web Tokens). There are also facilities for running individual commands (so you can embed them in shell scripts) as well as a built in batch file system so you can write much larger and more complex sets of commands. There is even support for environment variables.

Abbreviations
JWT = Java Web Token conforms to RFC 7519
JWK = Java Web Key conforms to RFC 7517

JWTs have gotten to be very important within the OAuth community because they allow for a verifiably secure way to get information from a trusted source.

Introduction to JWTs

Here is the quick and dirty introduction to them. The format is extremely simple:

Header.Payload.Signature

All three fields are actually base 64 encoded (which makes sending them over the web very easy). The header and payload are JSON objects. The Signature is a binary string created from these and base 64 encoded. So a real-life token looks like

eyJ0eXAiOiJKV1QiLCJraWQiOiIyQkY5NTVDMjA0QjU1NTgzQjRCNzU3REI5QjY0RDE2OSIsImFsZyI6IlJTMjU2In0.eyJpc3MiOiJodHRwczovL215LmJvZ3VzLmlzc3VlciIsImF1ZCI6Im15LWF1ZGllbmNlIiwiaWF0IjoxNTY4OTA2NTgwLCJuYmYiOjE1Njg5MDY1ODAsImV4cCI6MTU2ODkxNjU4MH0.YEeAFQPdQEupKiUWmrfY9NEl6eoRpWQ4bzC8W4w4pnDjgeJOBazlMpUB5BMZMuH_vv04CaxzyXYdugF39jKvpTRE5ydwRcTezwkIea6OZJUS2VCX_F-YSajll4ddAkUC9oB0Qk4QtW5c72Bo1iUXSQ4EGWithnuQXp0qp4y25Kegrel2iRxgpa-lUQENA7o9fZrqJnY45MkfJ-9nvygJaD2b2QSAkh4cocGLL4_xF3hjON_IEsBRcdjq079TjVA-3-pUUzVmu_irFsrmgYDNQE_vQDNLByENDdoj3p9GeBtx1odebYWUW86s0s63JtOOXgQ17fpvi0c4cGz2asQyGg

Yes there are 2 periods buried in there some place. Decoding the header and payload yields:

header

```
{
 "typ": "JWT",
 "kid": "2BF955C204B55583B4B757DB9B64D169",
 "alg": "RS256"
}
```

payload

```
{
 "iss": "https://my.bogus.issuer",
 "aud": "my-audience",
 "iat": 1568906580,
 "nbf": 1568906580,
 "exp": 1568916580
}
```

So why do this? Because the signature is created (this requires the exact payload and cannot be forged) on a server using its private key. To validate the signature, you must use some cryptography and the server's corresponding public key. This means that while, yes, anyone who gets one can look at the header and payload and for that matter verify the signature, you as the recipient know that the signature could only have possibly come from the server. In practice, if you get one, you can trust it is authentic if the signature checks.

Where to get the binaries

You can find these in the file jwt.tar in the most recent OA4MP release on github. Download it, unpack it and look at the readme.

The Interpreter

At the very least, you can simply fire up the command line interpreter once everything is unpacked by issuing

```
java -jar jwt.jar

jwt>
```

This is the prompt. To see what is available, type in --help and hit return. A list of all commands will follow. To see the specific help for a command, e.g., echo, you would enter that plus --help:

```
jwt>echo --help
echo arg0 arg1...
 Simply echos the arg(s) to the console . This is extremely useful in scripts.
See also set_output_on
```

Note that in this case, there is also a reference to another command that may be useful. There is a file called set-env.sh in the distribution that will let you set the location of the JWT distribution if you want to run it from other locations.

To exit the interpreter, the command is exit:

```
jwt>exit
```

Stem to Stern example

In this example, we have a JSON file called **`my_json.json`** – any will do, actually – and we are going to start `jwt`, create some keys and generate a JWT. First the contents of the JSON file:

```
{
 "claim0":"my claim",
 "isMemberOf":["bgsu_all","bgsu_physics"],
 "iss":"https://bgsu.edu/oidc/"
}
```

This is pretty bare-bones, but works fine as an example. Output messages from the tool are sometimes omitted, since it tends to be helpful.

```
java -jar jwt.jar
  (Lots of startup stuff)
jwt> create_keys -default_id AAAAA -out /tmp/my_keys.jwk
jwt>/commands
   (Lists all commands)
jwt>generate_token –help
   (Prints specific help for the generate_token command)
jwt>generate_token -in /tmp/my_json.json -keys /tmp/my_keys.jwk -key_id AAAAA
eyJraWQiOiJBQUFBQSIsInR5cCI6IkpXVCIsImFsZyI6IlJTMjU2In0.eyJuYmYiOjE3MjI1MTQ2NjgsIml
zcyI6Imh0dHBzOi8vYmdzdS5lZHUvb2lkYy8iLCJpc01lbWJlck9mIjpbImJnc3VfYWxsIiwiYmdzdV9waH
lzaWNzIl0sImNsYWltMCI6Im15IGNsYWltIiwiZXhwIjoxNzIyNTE1MjY4LCJpYXQiOjE3MjI1MTQ2Njh9.
ThskKfoGQ6UvOQ4RYmz4oj5ClibotIkrALZE-
6NajBUSu7Nn3mNHoul_cQ6qdM6CzGmmQwbPC63JtinHY7gweV5o8tLH0NMfoYLfmlM9dF5ynKwsAqgwy7Z7
6zvnlYOM_SNLUcabMe9Lpi5prspf6ofN_2I377UEBJsWe90J-CfJczWDv-BL2GUXtzANmNeapxzz1W-
```

```
fFywag83-
AOR_6hMJ57UTnr_oUfWQrq1j08xnIr50hFIICewyQTQQY2w40UuUEU2Api3IkrqbpjyhAulgK8MWltprBdc
55_i36j-XglKeXEdqZpAYg1YvYLqTkeZku6zx_Na_Ab0pMycC3g
```

This has just generated a token. Remember that the token parses as header. payload.signature and since the keys were just generated, the signature of your example cannot possibly match this one. To print this token, use the print_token command on it:

```
jwt> print_token eyJraWQ…
header
{
  "kid": "AAAAA",
  "typ": "JWT",
  "alg": "RS256"
}
payload
{
  "nbf": 1722514668,
  "iss": "https://bgsu.edu/oidc/",
  "isMemberOf":   [
    "bgsu_all",
    "bgsu_physics"
  ],
  "claim0": "my claim",
  "exp": 1722515268,
  "iat": 1722514668
}
```

Of course, you probably want to validate this token to see how it is done. Use the validate_token command:

```
jwk>validate_token -keys /tmp/my_keys.jwk
eyJraWQiOiJBQUFBQSIsInR5cCI6IkpXVCIsImFsZyI6IlJTMjU2In0.eyJuYmYiOjE3MjI1MTQ2NjgsIml
zcyI6Imh0dHBzOi8vYmdzdS5lZHUvb2lkYy8iLCJpc01lbWJlck9mIjpbImJnc3VfYWxsIiwiYmdzdV9waH
lzaWNzIl0sImNsYWltMCI6Im15IGNsYWltIiwiZXhwIjoxNzIyNTE1MjY4LCJpYXQiOjE3MjI1MTQ2Njh9.
ThskKfoGQ6UvOQ4RYmz4oj5ClibotIkrALZE-
6NajBUSu7Nn3mNHoul_cQ6qdM6CzGmmQwbPC63JtinHY7gweV5o8tLH0NMfoYLfmlM9dF5ynKwsAqgwy7Z7
6zvnlYOM_SNLUcabMe9Lpi5prspf6ofN_2I377UEBJsWe90J-CfJczWDv-BL2GUXtzANmNeapxzz1W-
fFywag83-
AOR_6hMJ57UTnr_oUfWQrq1j08xnIr50hFIICewyQTQQY2w40UuUEU2Api3IkrqbpjyhAulgK8MWltprBdc
55_i36j-XglKeXEdqZpAYg1YvYLqTkeZku6zx_Na_Ab0pMycC3g
token valid!
```

Of particular note, the key id is in the header, so you cannot set that. This merely returns that the token is valid.

What if you wanted to try this with elliptic curve keys instead? You could generate those with

```
jwk>create_keys -ec -out /tmp/my_ec_keys.jwk -default_id AAAAA
jwk>list_keys -in /tmp/my_keys.jwk
{"keys": [
    {
    "alg": "ES256",
    "kid": "AAAAA",
    "use": "sig",
```

```
    "kty": "EC",
    "x": "AJ-Z9g38va9GoJJN6C9BVGPXmyAmNJEOe3DS_-XgYSv0",
    "y": "ALxLxb46Uyys2BZHrel-2xs0j0aw4QvPAaXoh4wdSYpe",
    "d": "ANon6ULAYkBuDwOj8QUmdRj6Wj2WC3gFRcCHR2K1GXup"
  },
    {
    "alg": "ES512",
    "kid": "99285E66E58C7B61",
    "use": "sig",
    "kty": "EC",
    "x":
"NWHAFDdlnmhXTS9HF_fwTwXLg1DOKeVVU7ZfzGCf_7eH2onThIiZEqyIk49hNWKAs4_zh1lBI2WL5XxanR
Q-4dA",
    "y":
"AR6uGmegXCbU1eIFbugQ4VjgyzYUiiJxTNZyClxvoQZveOG9dgU4ARj5FbsX8eryZBzUY0OOKBMIf_R-
7n_Clbxw",
    "d":
"AbzC1DuZt9i5YuPryXPvHdm7iaiL9ZBac6VWUPyKBc0VkaLqK5wXoCJSzhN4AtlxzEibs6TQxv5CngJfYj
FxY8od"
  },
    {
    "alg": "ES384",
    "kid": "57030495233B0BC4",
    "use": "sig",
    "kty": "EC",
    "x": "XFKePeJnJMmIWmIZK2ftu1k1xLHD2d6crykUcH8oIc3HJY6ajaDAZNqtg0sL8OMj",
    "y": "GZfmrTs1bC1aFC4NHXky0Wc6TqfMdbQLJVMCRocHR077HiNfdCehP0T_ayTQdLmb",
    "d": "RBc-lBzwbXPtT_Xd5eGEr6zUCaPoIdkXaTWjyrlHDvQvJn5aG4RcPdFi8wRwDELW"
  }
]}
```

Note that we specified that default key id. In both RSA and elliptic curve keys sets this is assigned to whatever is currently considered to be the most common key to use. To generate the token:

```
jwk>generate_token -in /tmp/my_json.json -keys /tmp/my_ec_keys.jwk -key_id AAAAA
eyJraWQiOiJBQUFBQSIsInR5cCI6IkpXVCIsImFsZyI6IkVTMjU2In0.eyJuYmYiOjE3MjI1MTYwNDksIml
zcyI6Imh0dHBzOi8vYmdzdS5lZHUvb2lkYy8iLCJpc01lbWJlck9mIjpbImJnc3VfYWxsIiwiYmdzdV9waH
lzaWNzIl0sImNsYWltMCI6Im15IGNsYWltIiwiZXhwIjoxNzIyNTE2NjQ5LCJpYXQiOjE3MjI1MTYwNDl9.
Ffr-
DfzjxhePgzB1EUbxSo_7cBf_Zflbta3oDDgjiuOeeP03V2op9hCn3ctTjfFtHJM0he0ggvqSLnmeL4Yw9g
```

(This highlights one of the major pluses with using elliptic curve keys and that is smaller keys and signatures, though they keys themselves are much more computationally intensive to create.)

Logging

At invocation, you can set a log file. Note that this will put a lot of extra things in to it, such as the output for commands, so it is possible to have sensitive information in it. Just be sure you do not set the log file to be public, like any other log file. You wouldhave

```
java -jar jwt.jar -log /path/to/log
```

If you need to generate extra output, you make issue the set_verbose_on command with an argument of true. This will make JWT much chattier and dump more things in to the log. You may turn off by setting it to false.

Setting shell variables first

Before running the scripts, you should set the environment (assigning values to a couple of shell variables) by running the set-evn.sh script (typically using the source command):

source ./set-env.sh

Basically you just need to point java at the directory where you put the distribution. If you got this as a tarball, then it should all just work from the untarred directory. You only need to set the environment once in your session.

Running Batch (Single) commands

Included are bash scripts that allow you to execute single commands, such as creating keys and so forth. The intent of allowing the interpreter to run single commands is that you can embed them in shell scripts.

So to run a single command with the interpreter, you add the -batch flag. Here is how to print out the help for the create_keys command:

```
java -jar jwt.jar -batch create_keys --help

create_keys [-in set_of_keys -public] | [-private] -out file
 Create a set of RSA JSON Web keys and store them in the given file...
```

(Lots more stuff prints here, I just want you to see how it works.)

So invoking this in the course of a shell script is pretty. There are a few such scripts supplied in the distribution. These invoke a few common single commands. You can invoke detailed help by invoking the script with the --help flag.
E.g.

./create_keys.sh --help

Note that this invokes a much larger library and the help talks about interactive mode and batch mode. These scripts
all run in batch mode.

create_keys.sh = creates a set of standard RSA keys (both public and private parts) at various strengths.
  Output is JSON Web Key format.
Create_symmetric_keys.sh = create one (or more) symmetric keys.
create_token.sh = takes a token and simply creates the signature -- no claims added.
generate_token.sh = takes a set of claims and adds all of the standard expirations, possibly a JTI and so forth.
log.sh = prints out the tail of the current log file.
print_token.sh = prints the header and payload of a token. No validation or other checking is done.
validate_token.sh = takes a token and key and verifies the signature.
run.sh = run any command in batch mode.

For instance

```
$SCRIPT_PATH/run.sh echo foo!
```

```
foo!
```

** Batch files

The processor also has the ability to run batches of commands found in a single file. The syntax of the file is designed to be as minimal as possible:

* a line that starts with a hash (#) is a comment and is ignored at run time
* Commands may span many lines with as much whitespace as you like, but the end of command marker is a
 semi-colon (;) and as soon as that is found, the command will execute. Note that all lines will be
 put into a single line with a single space between each by the command preprocessor.

These typically end with an extension of .cmd. You can either feed a command file directly to the interpreter:

```
java -jar jwt.jar -batchFile file_name.cmd
```

Or set your environment and use the run-cmd.sh shell script. E.g. to rpint out the sample token:

```
./run-cmd.sh ex_print_token.cmd
```

and a sample would be printed.

Environment variables.

The command processor has any number of features including the ability to have an environment set either as a Java properties file or as a JSON object. The distro contains two examples

```
test_env.props
more_props.json
```

How do these work? The key/value pairs are read in and the you simple use ${key} wherever you want.
Note that the substitutions happen before any processing of the command line, so you can literally replace anything you want, including command line switches.

For instance, if your properties file consists of the single line

kid=ABC123

(Or the equivalent JSON file of

{“kid”:”ABD123”}

would work.)

Then the following command

```
generate_keys -kid ${kid} -jti...
```

would become

```
generate_keys -kid ABC123 -jti ...
```

and then get passed to the interpreter. Note that there is no checking of any sort done -- it is just a straight up
string substitution, so you can have something like **MYKEY${kid}456** ---> **MYKEYABC123456** for instance.

You can import environment variables in interactive mode and in batch files. Batch mode is still not supported though
that may change. At this point, you must pass in all parameters directly (again since single commands are intended to be run as part of a shell script, the management of parameters is done there).

Here is an example to show how to use environment variables with a command file.
This will write the properties to the console so you can see that it works. In the following file are the commands that import an environment and print out some of it:

ex_env.cmd

Invoke

```
./run-cmd.sh ex_env.cmd
```

to run the command file (a file containing a set of commands).