SAS Protocol

The SAS (Subject - Action Service) protocol allows for running command line applications remotely via a web service. Your local client talsk to the service and commands are executed there, with the reusults displayed locally. It allows for multiple users to register and they may have multiple sessions with their own I/O. All exchanges are secure. You must implement the commands to run, as this is just a framework and servlet that allows that. Any command line program can be converted to an SAS version. What happens is that the IO is replaced by the SAS version, allowing the entire application to operate oblivious to SAS. The user just uses the application but the calls are executed on the service and returned.

Prerequisites

The client registers (out of band, at this point so an administrator can vet them) and client sends the server a public key. The client will use the private key to encrypt all exchanges. In the initial exchange, a symmetric key (henceforth called the **session key)** is created and exchanged and all subsequent communications occur with that. This is a minor variation on PGP.

The basic protocol is POST only and the body of the post is a based64 string which consists of a JSON Object (details on format below) that has been converted to a string then encrypted. The body of the post is never in plain text, though the content type is text.

Information is sent as headers. The initial logon requires sending Authorization Basic and after that, the session id must be included as **`session-id`**

Request format

A request is of the form

```
                                                                {"sas":
   {"action" : action,
    "content" : base64 string
 },
"comment": string
}
```

Actions:

- logon - initial logon
- execute - take the content, execute it
- logoff - end the session.

The entire request is encrypted with the client's private key.

API

Initial logon

Request

```
{"sas":{"action":"logon","executable_name":name }}
```

Headers

Basic authorization header. There is no password.

Note that the executable_name is assigned by the server and may be omitted if the server only serves up one type of executable.

Response

```
{
 "status":int,
 "session_id":uuid
}
```

The returned session_id must be sent with each future request in the header name **`session-id`**.

Execute

Request

```
{"sas":
  {"action":"execute",
    "content": base64string
  }
}
```

Header

```
session-id
```

Response

```
{ "status": int,
  "content": base64string,
  "prompt": base64string
}
```

Logoff

Request

```
{"sas":
```

```
  {"action":"logoff"}
}
```

Header

```
session-id
```

Response

```
{ "status": int,
  "message": base64string
}
```

New Key

This is a request from the client for a new session key. Clients should request a new one ever so often if they want to ensure very high levels of security. The old key is invalidated and all encryption occurs with the new session key.

Request

```
{"sas":
  {"action":"new_key"}
}
```

Header

```
session-id
```

Response

Configuring a server

The Server configuration

The basic server configuration is much like any configuration in the NCSA security library.

```
<config>
  <sas name="default"
       alias="debug"
       accessList="localhost"/>
  <sas name="debug"
       enabled="true">
  <fileStore path="/home/ncsa/apps/sas/storage">
     <clients/>
  </fileStore>

  </sas>
</config>
```

This has outer tag for **`config`** and each configuration has a tag of **`sas`**. The name is passed in so that there can be many configurations. The main section is the storage. This should point the location

where client configuration can be stored. Note that *independent* of the ambient server, you can set the **`accessList`** to be a list of hostnames that are restricted. No omitting it or no entries means no restriction is done. The elements are comma separated.

Adding a client.

First step is that the client must have configured itself and have a public key. Once you are in the SAS CLI, you simply issue the create command and follow the prompts. You can enter the key information in a variety of formats.

Once configured, the client is ready for use.

Running the CLI

The SAS CLI is a simple command line tool that can be extended to do work with any SAS extensions you care to write. Flags

flag	Description
-cfg	Path to the configuration file
-edit [filename]	Whether to edit the given file. If no file is given you will be prompted for one.
--help	Show general help for the CLI
-v	Increase verbosity.
-print_key	Print the public key then exit. Options are jwk (for JSON web key format) or pkcs for PKCS 5 format. Either of these can be used to register the client. Note that this requires the -cfg parameter to so it knows where to get the key.

Client configuration

Fortunately, the basic command line client has a feature to create a configuration. The client startup options for configuration are

```
-edit [filename]
```

If the file exists, it will allow you to set each of the properties in turn. Follow the instructions on the screen, but the basic functionality is that

```
property “old value”:>
```

which means that whatever the *property* is has the current *”old value”*. If you hit return only, then nothing is changed, otherwise, type in the new value.
You will also be prompted to create keys. There are options for JSON webkey or PKCS format. JWK is the more modern and is (as of this writing) supplanting the older PKCS formats, but both are still fine.

Invocation examples

Basic startup

```
sas -cfg fileName
```

Start up the client with the given fileName. Note the fileName must be the complete path

Create a new configuration

```
sas -edit
```

Edit an existing configuration

```
sas -edit fileName
```

Get help

```
sas --help
```

Print the public key

```
sas -print_key jwk -cfg fileName
```

Extending SAS

So the entire aim of SAS is to be able to remotely run some piece of code. You need to implement the Executable interface, then extend the SASServlet and override the createExecutable method to return your Executable. The system ships with a really silly EchoExecutable that just echos what the user types in, but does do all of the correct logon and logoff actions, so it is a good test to see if you can register a client and use one before you get down to writing your own.

Converting your application to a SAS application.

The major factor is that it should IO such as **`System.out`** or **`System.in`** or the Sec-Lib IOInterface (which models those).. The aim then is to intercept input, wrap it up and send it to the server where it is executed, then print out the result. The user experience is exactly as if using a command line.

For the server-side executable:

1. Extend the class, implementing the Executable interface.
2. Intercept all IO to use IOInterface. The BasicIO class in Sec-Lib just fronts all of the standard Java IO, so rewrite with that first if needed. The you can just swap it out. Ideally in your extension, you just set the IOInterface and that is the only tweak you need.

For the Servlet

1. Extend the SASServlet. Note that this is inclued in a jar with the assumption you will extend it as part of creating your own web apps.
2. You need to override the createExecutable() method in the SAS servlet. This returns an instance of the server-side executable.