
CyberCAPTOR-P2DS

Release 4.4.3

September 29, 2015

1	Licence	1
2	Table of Contents	3
2.1	Architecture	3
2.2	P2DS Installation and Administration Guide	3
2.2.1	Setting Up	3
	Installing the Services	3
	Installing from WAR	3
	Installing from Source	4
	Generating Key Pairs	4
2.2.2	Configuring the Services	5
	Group Management	5
	Peer	6
2.2.3	Note about persistence.xml	7
2.3	P2DS User and Programmer's Guide	7
2.3.1	Performing a Computation	7
	Creating a Group	8
	Registering the Peers	9
	Starting the Peers	11
	Giving the Peers Some Inputs	11
	Viewing the Results	12
	Useful Hints	12
2.3.2	Using the APIs	12
2.4	P2DS Workflow	14
2.4.1	Step 1: Setting Up	14
	Step 1.1: Set Up Group Management	14
	Step 1.2: Set Up Peers	14
	Step 1.2: Generate Certificates	14
2.4.2	Step 2: Define a Computation	15
2.4.3	Step 3: Define the Group	15
2.4.4	Step 4: Prepare the Data	15
2.4.5	Step 5: Launch the Computation	16
2.4.6	Step 6: Look at the Results	16

Licence

This project contains ZHAW's Contribution towards Privacy-Preserving Data Sharing (P2DS) for the Cybersecurity GE in FIWARE.

Copyright © 2015 Zürcher Hochschule der Angewandten Wissenschaften (ZHAW).

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Table of Contents

2.1 Architecture

The SEPIA library has rather fixed ideas of how communication between the various peers should happen, and it is unfortunately rather difficult to untangle these from the basic protocols.

At its base, SEPIA creates a thread for every peer, and that thread communicates with other peers through messages that signify events. When a peer waits for a message, it blocks at a socket. Needless to say, this is not nice when in the context of servlets. We solve this problem by spawning a thread for every peer, and then connecting the input of the thread to a blocking message queue. Whenever the servlet receives a message that's for a particular thread, the servlet extracts the message, verifies its signature (which has to be done separately from SSL for technical reasons) and puts it into the message queue.

2.2 P2DS Installation and Administration Guide

2.2.1 Setting Up

You need at least four parties in order for P2DS to work: three parties have data on which they want to perform a computation, and one party manages that computation. Neither party needs to trust any other party. The parties having the data host *input* and *privacy peers* and the managing party hosts the *group manager*.

Installing the Services

Installing from WAR

You should have received three WAR files as part of the P2DS distribution, or you can build them yourself; see below.

- p2ds-group-management.war contains the group manager
- p2ds-peer.war contains peers
- p2ds-receiver.war may be used for demo purposes (receives final results)

These must be deployed on application servers. Deploy p2ds-group-management.war on your group manager and deploy the p2ds-peer.war on each organisation that wants to participate in the computation. Additionally you can also deploy the receiver, but you should write your own endpoint.

Installing from Source

In order to build and compile the services from source code, first get the source, then build:

```
git clone https://github.com/fiware-cybercaptor/cybercaptor-P2DS
cd sepia
mvn install -DskipTests
cd ../p2ds
mvn package
```

Now the directories `group-management/target`, `peer/target`, and `receiver/target` will contain the respective WAR files.

Generating Key Pairs

All messages that are exchanged in P2DS are digitally signed. Additionally, all parties should employ TLS. Digital signatures are needed to be able to check the messages by the receiving parties in addition to the transport security offered by TLS. And TLS is needed because in some cases, sensitive information like an authentication token is transported in the messages.

First, get and build the P2DS key generation program:

```
git clone https://github.com/engineering.zhaw.ch/munt/p2dsKeygen.git
cd p2dsKeygen
mvn package -DskipTests
```

Next, use it to generate a key pair:

```
java -cp target/keygen-0.0.0.1.jar ch.zhaw.ficore.p2ds.keygen/Main
base64 key.private > key.private.b64
base64 key.public > key.public.b64
rm -s key.private key.public
```

The last command deletes the (unneeded) binary key files, leaving only the Base64-encoded ones.

This generates an Elliptic Curve DSA key of 409 bits, which is supposed to have the equivalent RSA strength of more than 8192 bits (see [here](#)). However, it uses Java's `SecureRandom` generator, which has had [trouble](#) in the past. So it is probably best to be on the lookout for messages about `SecureRandom`.

This program generates two files, `key.public` and `key.private`, both of which must be uploaded to the respective peer. They can in principle be uploaded to any directory on the application server, but we recommend a directory to which only the application server has read access. Since it is not necessary to change the key files, once uploaded, we also recommend setting the permissions on these files to read-only. On Unix-like operating systems, do this:

```
sudo cp key.public.b64 key.private.b64 /var/p2ds
rm -s key.public.b64 key.private.b64
cd /var/p2ds
sudo chown apache key.public.b64 key.private.b64
sudo chmod 444 key.public.b64
sudo chmod 400 key.private.b64
sudo chmod 500 .
```

Again, the unneeded copies of the key files are securely deleted. This is not important for the public key but very important indeed for the private key.

Here, `apache` is the system's pseudo user that runs the application server's processes.

2.2.2 Configuring the Services

Group Management

The Group Management's database configuration is described in its `persistence.xml` file:

```
<?xml version="1.0"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="p2ds-group-management" transaction-type="RESOURCE_LOCAL">
    <provider>
      org.hibernate.ejb.HibernatePersistence
    </provider>
    <class>ch.zhaw.ficore.p2ds.group.storage.Group</class>
    <class>ch.zhaw.ficore.p2ds.group.storage.Peer</class>
    <class>ch.zhaw.ficore.p2ds.group.storage.Registration</class>
    <properties>
      <property name="hibernate.connection.driver_class"
        value="com.mysql.jdbc.Driver"/>
      <property name="hibernate.connection.url"
        value="jdbc:mysql://localhost/p2ds"/>
      <property name="hibernate.connection.username"
        value="sepia"/>
      <property name="hibernate.connection.password"
        value="my=password"/>
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.MySQLDialect"/>
      <property name="hibernate.hbm2ddl.auto" value="create"/>
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.format_sql" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

The obviously configurable parameters are `hibernate.connection.driver_class`, `hibernate.connection.url`, `hibernate.connection.username`, and `hibernate.connection.password`. Change these to suit your database setup.

The group management service's configuration can be found in the `web.xml`. You only need to configure the `group/adminKey` option which is the *password* for admin functionality.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This web.xml file is not required when using Servlet 3.0 container,
  see implementation details http://jersey.java.net/nonav/documentation/latest/jax-rs.html#d4e194
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <servlet>
    <servlet-name>Jersey Web Application</servlet-name>
    <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
    <init-param>
      <param-name>com.sun.jersey.config.property.packages</param-name>
      <param-value>ch.zhaw.ficore.p2ds</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>/res/*</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>Jersey Web Application</servlet-name>
```

```
        <url-pattern>*/</url-pattern>
    </servlet-mapping>

    <env-entry>
        <env-entry-name>peer/adminKey</env-entry-name>
        <env-entry-value>default-admin-key</env-entry-value>
        <env-entry-type> java.lang.String </env-entry-type>
    </env-entry>
</web-app>
```

Additionally you may want to add some security constraints to disable the GUI from being public. You should read up on tomcat's security constraints documentation on how to setup security constraints, roles and realms. We recommend using at least http basic auth. In general everything except `/group-mgmt/*` is something you might not want to be public:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>GUI</web-resource-name>
        <description>all pages</description>
        <url-pattern>*/</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admins</role-name>
    </auth-constraint>
</security-constraint>
<security-constraint>
    <web-resource-collection>
        <web-resource-name>API</web-resource-name>
        <description>REST-API</description>
        <url-pattern>/group-mgmt/*</url-pattern>
    </web-resource-collection>
    <!-- without auth-constraint == public -->
</security-constraint>
```

Peer

The peer's database configuration is also described in the `persistence.xml`:

```
<?xml version="1.0"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
    <persistence-unit name="p2ds-peer" transaction-type="RESOURCE_LOCAL">
        <provider>
            org.hibernate.ejb.HibernatePersistence
        </provider>
        <class>ch.zhaw.ficore.p2ds.peer.storage.PeerConfiguration</class>
        <properties>
            <property name="hibernate.connection.driver_class" value="com.mysql.jdbc.Driver"/>
            <property name="hibernate.connection.url" value="jdbc:mysql://localhost/p2ds_input?us
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect"/>
            <property name="hibernate.hbm2ddl.auto" value="create"/>
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

You may and should change the properties based on your setup.

The input peer's configuration is likewise in its `web.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This web.xml file is not required when using Servlet 3.0 container,
     see implementation details http://jersey.java.net/nonav/documentation/latest/jax-rs.html#d4e194
-->
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <servlet>
    <servlet-name>Jersey Web Application</servlet-name>
    <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
    <init-param>
      <param-name>com.sun.jersey.config.property.packages</param-name>
      <param-value>ch.zhaw.ficore.p2ds</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Jersey Web Application</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>

  <env-entry>
    <env-entry-name>peer/url</env-entry-name>
    <env-entry-value>http://localhost:12001/p2ds-peer</env-entry-value>
    <env-entry-type> java.lang.String </env-entry-type>
  </env-entry>

  <env-entry>
    <env-entry-name>peer/adminKey</env-entry-name>
    <env-entry-value>default-admin-key</env-entry-value>
    <env-entry-type> java.lang.String </env-entry-type>
  </env-entry>
</web-app>
```

You only need to configure the `peer/url` and `peer/adminKey` environment entries. `peer/url` is the url under which the peer service can be contacted (the url you host it at) and `peer/adminKey` is the admin key for REST-API methods only to be used by the admin.

2.2.3 Note about persistence.xml

```
<property name="hibernate.hbm2ddl.auto" value="create"/>
```

The `hibernate.hbm2ddl.auto` property set to `create` will re-create the database (deleting existing entries) at every launch of the services. This is a good setting if you are just experimenting with P2DS but it's not a production setting. You may leave the property on `create` for the setup phase but once you go *live* you should absolutely remove it.

2.3 P2DS User and Programmer's Guide

2.3.1 Performing a Computation

This part manual is written mainly for the person who runs the group manager. If you run a peer, start reading from [here](#).

Creating a Group

First, locate your administration key. This can be given to you by the person who installed the group manager WAR file, since it is contained in the web services `web.xml` file. This administration key should always be appended to any administration URL. For example, if the group manager is installed on `grpman.example.com`, then the initial URL should be `https://grpman.example.com:8080/p2ds-group-management?adminKey=key`. (Obviously, substitute the correct port numbers and key value.)

Navigate to the group manager GUI (typically `.../p2ds-group-management/`). There you will initially see a simple form field, asking you to supply a group name. Let's say you enter "SimpleGroup" and click on "Create group!". When you return to the main page and click on the name of your group, you will see something like this:

Group Management

Groups

Click on the name of a group (headings) to expand and show more information!

Group creation:
Name:

SimpleGroup

Actions

Configuration

Members

Open registration codes

Status

Group status: Running. All peers are running.

Fig. 2.1: Overview

Next, you should set up a configuration. This configuration says what you want to compute and how to compute it. If you click on "Configuration", you should see the following.

Configuration

Field:	<input type="text" value="1013"/>
MaxElement:	<input type="text" value="1000"/>
MpcProtocol:	<input type="text" value="additive"/>
NumberOfItems:	<input type="text" value="2"/>
NumberOfTimeSlots:	<input type="text" value="2"/>
ResultBufferSize:	<input type="text" value="0"/>

Set Configuration

Fig. 2.2: Configuration

The first element to fill in is "MpcProtocol". This answers the question "what do you want to compute?". The only protocol we have definitive demand for is "additive", meaning that you can compute a vector sum. So you have to put "additive" in that row. (Other protocols include *top-k*, where you can compute the top *k* elements of each column. For example, the maximum element would be top-1. However, at the time of writing, we have no demand for that protocol, and, while it is implemented, we do not support it.)

The next element you should fill in is “MaxElement”. This is the largest intermediate result you envisage. For “additive” it’s the largest value you think will occur in a result vector. For example, if you sample “number of attacks” in 1-minute intervals, you should hardly expect more than 1000 attacks, so 1000 is a good value.

Next should be “Field”. This is a prime number larger than MaxElement. In our case, 1013 does the trick. If you are in doubt, you can put the value $263 - 5 = 9223372036854775783$ in that field, even though computations in that field will take longer.

Next you should take care of NumberOfTimeSlots and NumberOfItems. What this means is that of you have $N = m \times n$ items, you will serve these items n at a time, m times. For example, let’s say you have 6 numbers in your vector. You can choose to compute the vector sum by adding up two numbers, three times; three numbers, two times; one number, six times; or six numbers, once. They will all give the correct result, but some settings will be faster than others, and some settings will also be more robust. For example, if you have more items (large n), the computation will generally be faster, but if you have more time slots (large m), the computation will be more robust against peers failing occasionally. (In R terms, you can have a few “N/A” terms in your result vector.)

A special value for NumberOfTimeSlots is “-1”. This value will cause the peers to run in *streaming mode*. Usually, peers will run for NumberOfTimeSlots time slots and then shut down. When peers are running in streaming mode, they will forever wait for the next batch of numbers to process. Upside: you don’t have to initiate a new computation when you have new data. Downside: one downed or stuck peer will cause the whole computation to stall.

Finally, you can set ResultBufferSize to control in how many chunks the results are sent to the configured result endpoint (see the configuration guide on how to configure this). If you have ten timeslots and set the result buffer size to 1, then ten final results (one per timeslot) will be sent to the result endpoint. If you configure the result buffer size to be 10, only one large result will be sent, after all timeslots have completed.

If the computation fails, no results are sent. If a peer fails, restart the computation.

Registering the Peers

Next, talk to the administrators of all the participating organisations and give them the URL of your group manager service. They should deploy and configure a peer service on one of their application servers.

Now you can generate registration codes. These codes are one-time codes that peers use to authorise their registration. If you plan on having m input peers and n privacy peers, click the “Create registration” button $m + n$ times:

SimpleGroup

Actions



Fig. 2.3: Actions

For each registration code, you should see one “Open registration code”:

Give one code per peer to the admins. They will need these codes to register the peers with the group manager. Eventually, you will find a picture like this:

Here we show only one peer, but every peer should be present. Peers have registered, and in registering, they have used up one of the registration codes. There should therefore be no open registration codes at this point.

Peers have registered, but they have not been activated. Only activated peers can participate in the protocol. To do that, talk to each of the admins of each participating organisation and read the public key of their peers to them. These

Open registration codes

Code: TEST

Status

Group status: Stopped. All peers stopped.

Fig. 2.4: Registration

SimpleGroup

Actions

Configuration

Members

peerhans

- Last status: 1
- PeerType: 1
- Pid: 1
- URL: http://localhost:12001/p2ds-peer/peer
- RegistrationCode: TEST
- Verified: false

PublicKey

MH4wEAYHkoZIZj0CAQYFK4EEACQdagAEA3lg6xxX45uMESlRB2ADn7T7CgyH7Lxbxy/oS5XhIElBPwz/40cwDAc/VgGbDKa+HGBc/AGzw5lSc0DHc7WA1tSkRUkaW/LL9NbA6gIZJLMw+FV3RPor0VpJIoFVCAaV6WI1r99v8Y=

Fig. 2.5: Members

should be identical to the public keys of those peers that the admins have configured for their peers. Once that is done, click on “Mark as verified”. Rinse and repeat until all peers are verified.

The reason there is an additional verification step is that in principle, everyone can register a peer, but only you, the group manager, gets to decide who can participate in a computation. This step also serves as an authentication step to make sure that the public keys you got from the peers are authentic. This saves you from the hassle that is PKI.

Starting the Peers

Starting the peers is very simple: go to “Actions” and click on “Start peers”:

SimpleGroup

Actions



Fig. 2.6: Actions

Once that is successful, you should see that “All peers are running”:

Group Management

Groups

Click on the name of a group (headings) to expand and show more information!

Group creation:

Name:

SimpleGroup

Actions

Configuration

Members

Open registration codes



Status

Group status: Running. All peers are running.

Fig. 2.7: Running

Giving the Peers Some Inputs

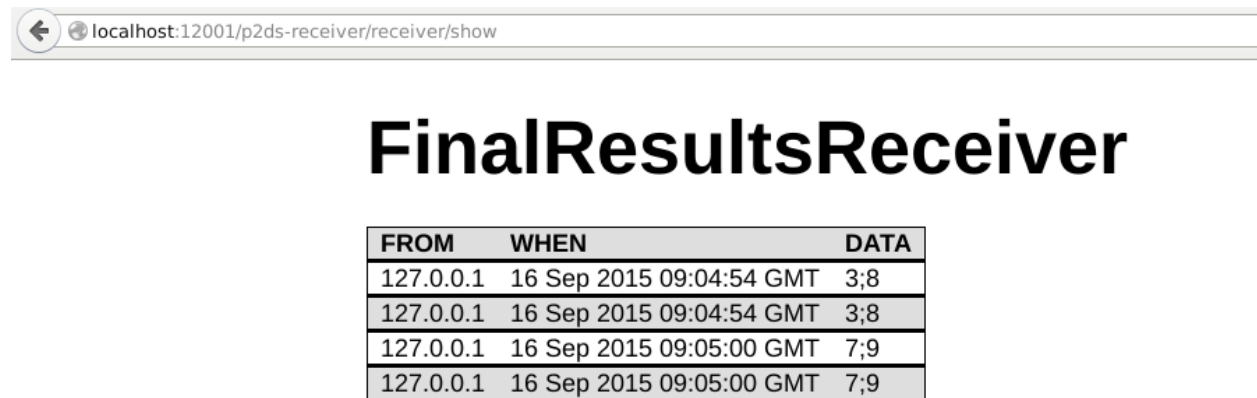
Once you have created input and privacy peers (refer to the API guide how to do that), you need to give them some inputs. Ideally, you have a web service that does that for you, but if not, you can do that by hand, using `curl`. Let's say that your peer is called `swisscom:ddos-input-peer-1` and that your `NumberOfItems` in the peer configuration (see the Group Manager guide above) is 6, since you are sampling every 10 minutes per hour. You

have seen 1, 4, 2, 8, 9, and 23 DDoS-attempts, respectively, so there is clearly something going on. Your peer runs on `p2ds.swisscom.ch:8080` and your registration code is `JKSdh3h7njs` (this is part of the peer service's configuration). Here is how to supply the peer with that input row:

```
curl -i -v -X POST --header "Content-Type: application/json" \
  -d '{"peerName":"swisscom:ddos-input-peer-1","data":"1;4;2;8;9;23"}' \
  https://p2ds.swisscom.ch:8080/p2ds-peer/input?registrationCode=JKSdh3h7njs
```

Viewing the Results

The input peers report results back to a URL that is set in the peer's configuration (the `finalResultsURL` property). We deliver a very simple results viewer with P2DS, which runs under `/receiver`. If you configure your input peer's `finalResultsURL` to read <https://p2ds.swisscom.ch:8080/receiver/receive> (substitute correct protocol, hostname, and port), then your results might look like this:



FROM	WHEN	DATA
127.0.0.1	16 Sep 2015 09:04:54 GMT	3;8
127.0.0.1	16 Sep 2015 09:04:54 GMT	3;8
127.0.0.1	16 Sep 2015 09:05:00 GMT	7;9
127.0.0.1	16 Sep 2015 09:05:00 GMT	7;9

Fig. 2.8: Results

Useful Hints

- *Always* use `https`, *never* use plain `http` when feeding data to the input peers. The value of P2DS would be greatly diminished if the supposedly secret data were sent in plain text. The protocol between the peers can be plain `http`, since P2DS uses encrypted shares already and additional encryption doesn't buy you much security.
- *Always* use `https` for talking to the group manager.

2.3.2 Using the APIs

Please refer to the [API Documentation](#). Exemplary usage of the APIs can be found in the file `grp.sh` in the source code's main directory, available from [the github page](#).

Here is a more verbose version of that file (with some parts skipped to keep this guide shorter). We assume that the admin key is `default-admin-key` and that the registration codes for all peers is `TEST`.

Please don't use `default-admin-key` and `testmode` in a production setting.

If you just want to experiment and/or play around otherwise on the shell using curl setting testmode to on can be quite convenient. If in testmode the registration codes generated by the group management service will not be random and *Please don't use testmode=on in production.*

```
#Set group-mgmt service to test mode
#Don't do that in production!
curl -i -v -X POST http://localhost:12001/p2ds-group-management/group-mgmt/testmode?on=true&adminKey=
```

To get started we need to create a group on the group management service:

```
curl -i -v -X POST --header "Content-Type: application/json" -d '{"name":"SimpleGroup"}' http://localhost:12001/p2ds-group-management/group-mgmt/group
```

Once we have a group we can generate registration codes for peers. The registration code is used by the peers to sign-up for a group. From the output of the previous command (see above) we know the id of the group (in this case 1)

```
curl -i -v -X POST http://localhost:12001/p2ds-group-management/group-mgmt/registration/1?adminKey=
```

The above command will give the group management admin a registration code that the admin needs to communicate to the peer operator. In this case the registration code is TEST. The peer operator can now create a peer (named peerhans) on the peer service:

```
curl -i -v -X POST --header "Content-Type: application/json" -d @./demo/files/peerhans.json http://localhost:12001/p2ds-peer/start/hanspeer?registrationCode=TEST
```

./demo/files/peerhans.json contains the configuration of the peer:

```
{
  "finalResultsURL":"http://localhost:12001/p2ds-receiver/receiver/receive",
  "peerType":1,
  "name":"peerhans",
  "privateKey":"MFECAQAwEAYHKoZIzj0CAQYFK4EEACQEOjA4AgEBBDNyjBeP85atxkIfiYqW+0kUB2H3guXcQWXT/tXVktbn.
  "publicKey":"MH4wEAYHKoZIzj0CAQYFK4EEACQDagAEAjig6xXX4SuME5lRB2ADn7T7CgyH7LXbxy/oS5XhIElBPwz/40cwD
  "registrationCode":"TEST",
  "groupMgmtURL":"http://localhost:12001/p2ds-group-management/group-mgmt"}
```

privateKey and publicKey must be PKCS resp. X.509 encoded as base64. *Please generate your own keypairs.* peerType=1 refers to an input peer where as peerType=2 refers to a privacy peer.

The peer will automatically sign-up for group membership and will be a member of the group on the group management service but marked as unverified. The peer operator and group admin should exchange fingerprints of the public key to verify the identity of the peer. If the group admin has verified the keys he can then mark the peer as verified:

```
curl -i -v -X POST http://localhost:12001/p2ds-group-management/group-mgmt/verify/hanspeer?adminKey=
```

Before we can start any peers we need to set a group configuration. The group configuration defines the parameters of the computation to do:

```
curl -i -v -X POST --header "Content-Type: application/json" -d '{"field":"1013","gid":"1","maxElement"
```

Please note that you will need at least three privacy peers and two input peers for the cryptographic protocol to work. You can start peers individually or let the group management service start all peers together. You can start individual peers by doing:

```
curl -i -v -X POST http://localhost:12001/p2ds-peer/start/hanspeer?registrationCode=TEST
```

Please note that the registration code is used as a means of *authentication* to prevent anybody from starting a peer. It is thus important that registration codes remain secret and are only known the the group management admin and the peer operator of the corresponding peer.

Once we have started all peers (in the case of `grp.sh` you will have two input peers and three privacy peers) we can add inputs:

```
curl -i -v -X POST --header "Content-Type: application/json" -d '{"peerName": "hanspeer", "data": ["1;3"
```

2.4 P2DS Workflow

Recall the graphic from the README:

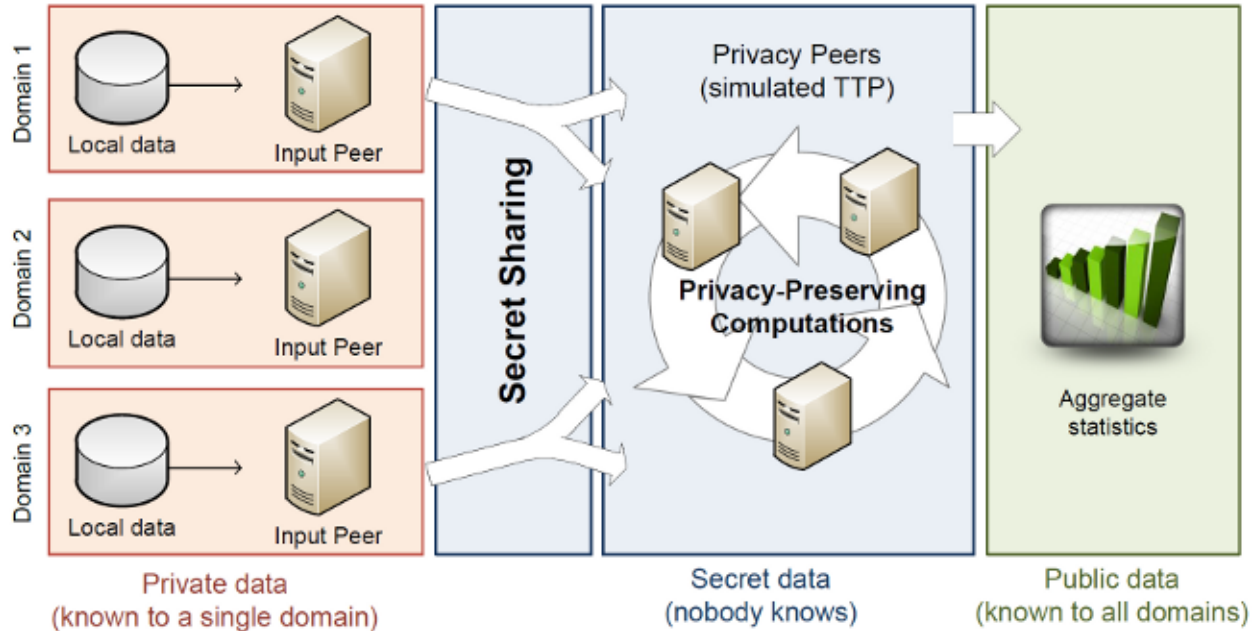


Fig. 2.9: Scenario

Let's say we have three organisations, called Domain 1, Domain 2, and Domain 3 in the graphic, that want to know the total number of attacks seen in the last 24 hours, with a granularity of five minutes. In mathematical terms, what these organisations want is $x_1 + x_2 + x_3$, where x_1 , x_2 , and x_3 are vectors with $24 \cdot 60/5 = 288$ elements, and they want to do this without revealing their own x_i to any of the other domains. Here is how the three domains could use P2DS for their needs.

2.4.1 Step 1: Setting Up

This section describes how to set up the group management and the various peers. Some actions have to be done only once, whereas others have to be done for each new computation.

Step 1.1: Set Up Group Management

Step 1.2: Set Up Peers

Step 1.2: Generate Certificates

This process is also described in SEPIA's [User Manual](#).

First, each organisation generates a public key certificate for each input and privacy peer. There must also be a certificate for the group management service.

Key generation and management is something that is done vastly differently from organisation to organisation. We sketch here a process that will work with Java's keytool; if your organisation follows a different process, use that. The important thing is that at the end, you have a Base64-encoded self-signed X.509 certificate.

In this example, we first generate a 2048-bit RSA key, which conforms to current best practices; Each organisation should replace the names `privacypeer01` with a name that they are more comfortable with. Each organisation *must* replace `privacypeer01KeyPass`, which will be the password that unlocks the private key and `privacypeer01StorePass`, which will be the password that unlocks the keystore.

This command line will generate the key pair, asking for some information about the organisation in the process. This information will be embedded into the key pair.

```
keytool -genkey -v \ -alias privacypeer01Alias \ -keystore privacypeer01KeyStore.jks \ -storepass
-keysize 2048
```

Next, generate a self-signed certificate, again replacing the various names with the ones that your organisation has chosen.

```
~[STRIKEOUT:bashkeytool -export -alias privacypeer01Alias -keystore privacypeer01KeyStore.jks
-storepass privacypeer01StorePass -file privacypeer01Certificate.crt ~]
```

We will later describe how to upload these certificates to the Group Management.

2.4.2 Step 2: Define a Computation

Now, all the organisations need to get together and define what they want to compute. In this example, it's $x_1 + x_2 + x_3$, where each x_i is a 288-elements, and they want to do this without revealing their own x_i . This must be done outside the P2DS framework, and there are important details to consider that are not part of the framework.

One such detail is the question of time synchronisation. If the local clocks of the IDses are skewed, then so will be the x_i . Worse, if the clocks also show significant drift, the x_i will also drift and the final computation will be more noisy and more unreliable. We recommend to connect each timestamping participant to an NTP stratum 1 server (or to connect a stratum 0 time source to it of course).

Once that is decided... (describe here group management -> New Computation -> ...)

2.4.3 Step 3: Define the Group

Now it's time for the organisations to decide what the peers are that will take part in that computation. They need to drill holes in their firewalls, set up the peers, ...

Then the group manager can either create a new group in the Group Management GUI or reuse an existing one and link the computation defined in the previous step with that group.

2.4.4 Step 4: Prepare the Data

Now the Input Peers need access to the data.

2.4.5 Step 5: Launch the Computation

2.4.6 Step 6: Look at the Results

Information about development is also available [in the README file](#).