developerWorks®

Building distributed applications with ZeroMQ, Part 2: Get more out of your ZeroMQ application

Make your apps secure against eavesdropping and tampering

Daniel Robbins

November 15, 2017

In Part 1 of this series, I looked at leveraging ZeroMQ to easily build applications that are distributed in nature and can exchange messages with one another over the network. Although the code I shared is suitable for a real-world application, it was lacking in some security areas. In this part, I'll show how to remedy those failings.

In part 1 of this series, "Leverage ZeroMQ to build distributed applications," I looked at leveraging ZeroMQ to easily build applications that are distributed in nature and can exchange messages with one another over the network. I concluded that first part with some example code, which implemented a fairly robust Python client and server application that can exchange messages asynchronously.

From the standpoint of robustness, I believe the code I shared is suitable for a real-world application. However, there are a few exceptions related to security. The first problem is that no encryption is used. This means that the information exchanged can be inspected in transit by a third party. Another problem is that there is no authentication—the server will accept a connection from anyone on the Internet who adheres to the protocol and the client will gladly connect to an impostor server.

“ The example code uses the DEALER-ROUTER request-reply pattern, which according to the ZeroMQ guide is an advanced pattern. However, I think it's the best place to start with ZeroMQ because it is the most forgiving pattern. ”

Let that sink in for a bit. Although this might be fine for some purposes, such as on private networks, it does not truly meet our goal of delivering a secure communications mechanism for distributed applications over the public Internet. Fortunately, using the existing code as a starting point, we can easily enhance it to address all of these failings.

CurveZMQ

Our ZeroMQ code is easily upgradeable to a more secure version thanks to the implementation of CurveZMQ. CurveZMQ is an adaptation of the *CurveCP protocol*, which was designed by Daniel J. Bernstein of the University of Illinois at Chicago. CurveCP is a more comprehensive, security-enhanced protocol for the Internet, which delivers improvements over TCP in a number of areas, and is designed to be an enhanced replacement for TCP.

CurveZMQ implements a limited subset of CurveCP within TCP. Specifically, it uses the same high-speed elliptic curve cryptography that CurveCP uses and adopts the initial CurveCP "handshake" mechanism of initial key exchange. Combined with minor changes to adapt CurveCP for a connected and message-based use provides much of the security benefit of CurveCP in a more traditional flavor that is compatible with the ubiquitous TCP protocol. See "Related topics" below for more information about CurveZMQ.

For simply leveraging the CurveZMQ protocol, understanding all the deep details of the protocol is not necessary. However, it is helpful to know what the technology protects you against. For example, CurveZMQ should be able to effectively protect your communication stream against:

- A third-party eavesdropping by monitoring live network traffic. Since communications are encrypted, that is not possible.
- The injection of fraudulent data into an established communications stream. Thanks to asymmetric key encryption, this is also not possible.
- The replaying of previously captured network transmissions to impersonate a peer. CurveZMQ uses **"nonces"** to ensure that the data that was sent at a particular time will not be valid when replayed later.
- Overwhelming a server by using amplification attacks. CurveZMQ is designed to protect against this kind of attack.
- Intercepting and proxying the client's connection to the server in a man-in-the-middle attack. Because the client is deployed with the server's public key and knows it in advance, it will communicate only with a legitimate server.
- A combination of network capture and private key theft to gain access to the contents of previously recorded network streams. Because CurveZMQ uses transient keys that exist only for the lifetime of the connection to encrypt data, such an attack is not possible.
- The identification of a specific client by analyzing network communications. The client's public key is encrypted by the server's transient public key when it is sent to the server, protecting against this.
- Resource exhaustion of the server by a denial of service attack. The speed of elliptic curve cryptography and the design of CurveZMQ mitigates against this being a significant threat.

In addition, thanks to **ZAP** (covered later in this article), the server can ensure that only authorized clients are allowed to connect to the server, and thus guard against exposing potentially sensitive data to unauthorized clients.

Asymmetric keys

To take advantage of CurveZMQ, you need to generate key pairs—a public and private key. I'll show you how to generate two key pairs, one pair for the client and one pair for the server.

To make this process simple, I wrote a small Python application called curve-keygen. Curve-keygen generates a key pair and stores the pair in the ~/.curve directory by using the name that you specify. On the client system, run the following command:

```
          client $ ./curve-keygen --name client
```

The output looks something like this:

```
Created /home/drobbins/.curve/client.key.
Created /home/drobbins/.curve/client.key_secret.
```

The client code is configured to look for its key pair at `~/.curve/client.key` and `~/.curve/client.key.secret`. If you want to change the name of the key pair on disk, you will need to make a corresponding change to the client code so that it will find the right keys. It's very easy to do, but I think it's worth mentioning.

On the server system, which for testing purposes can be the same system as the client, run the following command:

```
server $ ./curve-keygen --name server
```

You'll see similar output to this:

```
Created /home/drobbins/.curve/server.key.
Created /home/drobbins/.curve/server.key_secret.
```

Similarly to the client, the server is coded to look for its key pair at `~/.curve/server.key` and `~/.curve/server.key_secret`. If you wish to change the name of the server key on disk, a corresponding change in the server code is required.

You now have two key pairs generated, but you're not done yet. For the client to establish a connection with the server, it needs a copy of the server's public key. If you are running client and server on the same system and user account for testing purposes, which is how this code is configured to run, the client will have access to the server's public key in `~/.curve/server.key`. However, if you are testing communications over a LAN or the Internet, you'll need to copy `~/.curve/server.key` from the server system to `~/.curve/server.key` on the client. Remember that you need only the server's public key—private keys should never be exchanged!

Copying the server public key to the client can be done with `scp`:

```
          server $ scp ~/.curve/server.key account@client:.curve/
```

If you are running both client and server on the same system for testing purposes, the `server.key` is already accessible by the client code and no copying is required.

The client code looks specifically for the server public key at `~/.curve/server.key` for establishing outgoing communications. So yes, this means that if you want to use a different name for this file, the client code needs to be modified slightly to look for the correct public key.

ZeroMQ Authentication Protocol (ZAP)

This authentication protocol may or may not be sufficient for your purposes. At this point, the client will connect only to an authentic server. But what about protecting the server from unauthorized clients? In general, that is a good idea. You wouldn't want some fraudulent client in possession of your server's public key to connect to it and attempt to extract sensitive information.

If you want your server to allow only authorized clients to connect to it, you will need to use ZAP, the ZeroMQ Authentication Protocol. For ZAP to work, one more step is required. You will create the `~/.curve/authorized_clients` directory on the server system, and will need to copy the public key of each client that will be granted access to this server into this directory. This can be performed with `scp` as follows:

```
server % mkdir ~/.curve/authorized_clients
client1 % scp ~/.curve/client.key account@client:.curve/authorized_clients/client1.key
client2 % scp ~/.curve/client.key account@server:.curve/authorized_clients/client2.key
```

If you are just following along with the sample code and trying it on a local machine as is, simply do this:

```
% mkdir ~/.curve/authorized_clients
% cp ~/.curve/client.key ~/.curve/authorized_clients/client.key
```

As noted above, each client stores its key locally as "client.key" by default, so if you copy the public key to the server, you'll need to rename each key so that they don't overwrite one another. This is fine—the file names of public keys in `authorized_clients` are not important. The server code will authorize all the public keys in the authorized_clients directory, regardless of name. As long as the names end with ".key", the code will use them to authorize clients. The *contents* of the file are the critical part.

You may be wondering whether ZAP is necessary. It is entirely up to you and your security requirements. Remember that without ZAP, any client in possession of the server's public key will be able to connect to it. With ZAP, not only must each client must possess the server's public key, but the server must also possess each client's public key within the `~/.curve/authorized_clients` directory on the server. Only those clients will be allowed to connect to the server.

For the purposes of this tutorial, you will be using ZAP so that the demonstrated code is running in the most secure possible configuration. When you look at the code, you'll see that toggling a single Boolean setting on the server side can be used to turn off ZAP if you don't need it or desire not to use it.

The code

Grab the code now

Grab the GitHub repo above and we'll now look at the code. If you view the **app_client.py** file, you will see that the code is almost identical to the code in Part 1. A notable change is that we are now importing the contents of key_monkey.py, and if self.crypto is set, we perform the following additional steps:

```
self.keymonkey = KeyMonkey("client")
self.client = self.keymonkey.setupClient(self.client, self.endpoint, "server")
```

The `KeyMonkey()` class is a class I wrote to simplify the addition of CurveZMQ to the applications. The call to `KeyMonkey("client")` takes care of locating the client's public and private keys on disk. `KeyMonkey()` uses the same file system conventions as curve-keygen to keep things simple.

The second line of additional code performs specific steps to set up the client connection. The last argument of `setupClient`, "`server`", specifies the name of the public key that legitimately identifies the server you're connecting to.

Now, let's take a look at the server. The server is a bit more sophisticated because it has two options: the ability to enable CurveZMQ as well as the option to enable ZAP. By default, both are enabled. Let's look at the additional code in the AppServer constructor:

```
self.keymonkey = KeyMonkey("server")
self.server = self.keymonkey.setupServer(self.server, self.bind_addr)
```

Similarly to how we used `KeyMonkey()` in the client, `KeyMonkey("server")` locates the server private and public key pair on disk. The second line calls the `setupServer()` method to initialize the server socket for CurveZMQ communications. If we stopped here, the server would accept connections from any client, but we will address this next by initializing ZAP.

The `start()` method of the server has a few additional lines:

```
          # Setup ZAP:
        if self.zap_auth:
            if not self.crypto:
                print("ZAP requires CurveZMQ (crypto) to be enabled. Exiting.")
                sys.exit(1)
            self.auth = IOLoopAuthenticator(self.ctx)
            self.auth.deny(None)
            print("ZAP enabled.\nAuthorizing clients in %s." % self.keymonkey.authorized_clients_dir)
            self.auth.configure_curve(domain='*', location=self.keymonkey.authorized_clients_dir)
            self.auth.start()
```

These lines set up a separate ZAP thread that perform authentication for incoming client connections. The `self.auth.deny(None)` line ensures that no client will be denied based on its IP address. After ZAP is enabled, however, we will restrict client connections to only those whose public keys we happen to have cached in `~/.curve/authorized_clients`.

If you are running the client and server on the same system, you can leave the code as-is and run the client and server in separate terminals as follows:

```
% ./app_client.py

% ./app_server.py
```

You should see them behave identically to the example code in Part 1: the client connects to the server and communicates with it. If you do not see output indicating that communication was successful, you'll want to check your private and public key configuration to ensure that you followed the steps correctly.

In addition, if you'd like to use the sample code to test over the LAN or Internet, you simply need to modify `AppClient.endpoint` in the client to point to the location of the server, and `AppServer.bind_addr` in the server to bind to a non- localhost IP address. Then, as long as you have the keys set up properly, you will be able to successfully run the sample code and see the client and server communicate. Everything should function identically to the code in Part 1, except this time the communications stream is protected by CurveZMQ.

Appreciating what you've done

So far, I haven't delved too deeply into the code, intentionally. I wanted you to get a feel for ZeroMQ and how it works before I got into the details. However, the example code used in this tutorial and Part 1 of this series, while concise, is actually doing a few advanced things that are worth pointing out.

The first is the use of ZMQStream. You'll notice that in both the client and server code, we create a ZeroMQ socket, and then wrap it in a ZMQStream. What is the significance of this? Well, the ZMQStream allows you to use an asynchronous coding model, so that you can register an `on_recv()` method that will be called when you receive a message. The ZMQStream allows you to start an `ioloop` (which `pyzmq` inherits from the Tornado Python package) with both the `periodictask()` and the `on_recv()` methods registered to run at various times. You don't need to write the main event loop, as the IOLoop takes care of that and calls the right methods at the right time.

You could have written very similar code and omitted the ZMQStream wrapper. In that case, however, you would not be able to register an `on_recv()` method or create a `PeriodicCallback`. Nor would you call `self.loop.start()`. Instead, you would create a loop similar to this on the server side:

```
while True:
  msg = HelloMessage.recv(self.server)
  # we have received a message
```

Above, the `recv()` method will block—that is, the application will pause—until it receives a message. This makes things a bit more complicated if you actually want to perform other tasks while waiting for a message to arrive, such as the `periodictask()` method allows you to do. As you might guess, there are ways around this. Here's one way, by using a ZeroMQ Poller:

```
poller = zmq.Poller()
poller.register(self.server, ZMQ.POLLIN)
keep_going = True

while keep_going:
  sockdict = dict(poller.poll(0))
  if sockdict and sockdict.get(self.server) == zmq.POLLIN:
   msg = HelloMessage.recv(self.server, zmq.NOBLOCK)
   # process message
  # ... or do other stuff
```

Note that above, you specify an argument of zero to `poller.poll()` which causes the poller to return immediately if no messages are available. You can also specify `-1` to wait indefinitely, or a positive value, which is interpreted as the number of milliseconds to wait for a message to show up before timing out. As you can see, this is a bit more complex than the approach the code currently uses, but in some instances it can be handy.

More appreciation

The next thing I should point out is that the example code uses the `DEALER-ROUTER request-reply` pattern, which according to the *ZeroMQ guide* is an advanced pattern. However, I think it's the best place to start with ZeroMQ because it is the most forgiving pattern: client and server can communicate without any restrictions. Both the client and server can send a message whenever they want, and they can choose to reply or not reply to any message they receive. This is not always true for other ZeroMQ patterns. This feature makes the `DEALER-ROUTER request-reply` pattern the easiest place to start when playing around with ZeroMQ, because with it you are least likely to run into restrictions or roadblocks.

Alternative request/reply patterns

Now would be a good time to point out that there are many other socket types in ZeroMQ, and there are quite a few other legitimate request/reply patterns that are useful. Below is a list of some that you might consider using in your own code, along with their various behaviors. The *ZeroMQ Guide* is an excellent source of detailed examples of these patterns.

- REQ-REP. This is a very basic request/reply pattern that has many more restrictions than the DEALER-ROUTER pattern. The REQ client sends a single request and then must wait for a reply from the REP server. When the server receives a request from one of the connected clients, the next reply it sends goes automatically to that client. It can send only one reply message to a client, when it first has received a request.
  With REQ-REP, the server cannot initiate a request to the client: communication happens in a ping-pong fashion with the client always initiating communication. This pattern works analogously to a set of gears meshed together. It is a good pattern for when the server is single-threaded and non-asynchronous and handles each client request by sending a reply before processing any other incoming messages. It can also be a good pattern for inter-process communication on a single host.
  Sometimes using REQ-REP can simplify your code, but for many scenarios, it is too limiting. Also, when the server receives a message, it is not prefixed by an identity like it is with a ROUTER. This is unnecessary, since the only time a server can (and must) reply to a

particular client is immediately after it has received a message from that client. The next message the server sends automatically goes back to this client. While not having to track identity can make the server code a bit cleaner, it can be a significant restriction.
- PUB-SUB. This probably is one of the most famous distributed patterns. It allows you to have a publisher that can send messages to all subscribers. The publisher isn't aware of who is subscribed, and cannot receive messages from subscribers. Of course, it is possible for the server to have both a PUB socket and a ROUTER or REP socket to process direct requests from clients.
- PUSH-PULL. Designed for work queues, this pattern can be used to create processing pipelines. Producers can push jobs via their PUSH socket to workers, and work will be evenly distributed among all workers. Workers will receive the work on their PULL socket, and can in turn PUSH their results to the next step into a pipeline or to a PULL socket that is set up to collect the final work results. PUSH-PULL can be a nice way to create pipelines. You may also consider using a REQ-REP pattern for this purpose, where the worker requests work from a server, and the server replies immediately with the next job from a queue that it manages.

Both the *ZeroMQ Guide* and the pyzmq documentation are excellent resources for exploring different types of request-reply patterns. As you gain experience with ZeroMQ, you will get a feel for the benefits and drawbacks of the different patterns and will be able to choose patterns that minimize the complexity and maximize the quality of your network communications code.

Here's some advice when it comes to choosing reply-request patterns: there is more than one pattern that can be used to accomplish any goal. If you are implementing a PUB-SUB system, you aren't obligated to use the PUB-SUB request-reply pattern. It may be a good starting point, but depending on the particulars of your task, there may be a better option. The beauty of ZeroMQ is that you can select a communications paradigm that suits your particular needs, rather than being shoe-horned into something that may not be suitable for what you are doing.

Similarly, you are not limited to just a *single* request-reply pattern. You might find it useful to use multiple types of patterns. A server or client can connect to or open multiple sockets of different types, and the example code provided here will work well even if you happen to combine multiple socket types in a single client or server.

Conclusion

In this article, we took our existing client/server ZeroMQ code and adapted it to be secure. We also looked a bit behind the scenes at the more complex details of ZeroMQ message processing.

In the next and final part of this series, I'll show you a useful ZeroMQ application and demonstrate some of the additional functionality described above. See you then.

Related topics

- ZeroMQ Guide
- CurveCP
- CurveZMQ - Security for ZeroMQ
- pyzmq documentation
- Python
- Tornado