# Introduction

Hey! My name’s Trevor, and in this video I’ll be going over the IDA application that Team Transparency has developed for Assignment 2 – Coding Output of the Secure Software Development course at the University of Essex Online.

First, we’ll go through how to set up and start the project. This will just touch on getting IDA and the ISS microservice running.

Then we’ll have an overview of the user experience, from both user and admin perspective; we’ll also have a quick, high-level overview of the whole system here. We’ll also very briefly go over the tables and schema in the database so we can build up some context.

After that we’ll go into a bit more detail on the ISS microservice. We’ll talk about the threading component, the request that it sends to IDA, and how IDA handles that data. We’ll also briefly talk about the encryption that we’re using, and I’ll probably point out some nice-to-haves or areas where security should be extended.

And finally, we’ll have a deep dive into the IDA codebase. We will walk through some endpoints, including login and logout, and we’ll talk a bit about sessions.

# Setup

So, to get IDA and the ISS microservice running, we first need two terminal sessions, one for each. We have to activate the virtual environment for both. We can see that a virtual environment has been enabled when you see this little (venv) output here, but this will be named whatever you name your virtual environment on setup, which you can see the README steps for a bit more context on that.

<ACTIVATE BOTH VENV>

So now that we’ve activated both our virtual environments, we can start up IDA here on one of them.

<START IDA>

I’m going to hold off starting the ISS microservice here, we’ll get to that in the overview, but it’s the same command as IDA.

Now we’ve started IDA, let’s move on to the overview.

# Overview

## UX

The user experience of IDA was built to be intuitive, and there are only a handful of different interfaces.

<ACCESS HOME>

Here we have the main page. Because we aren’t logged in, we don’t see much, so let’s log in.

<LOG IN TO IDA USING USER ACC>

So now that I’ve logged in under a non-administrative user account, we can see that we have the records view available to us.

I’m going to click that <CLICK> and here we have the records table shown to us. This is pulling data from our records endpoint using the GET method, and it’s just fetching all records in the database and presenting them here. I’ll show the HTML for this, it’s using Jinja templating as that is what Flask uses.

<SHOW RECORDSVIEW JINJA TEMPLATE>

So here we have the HTML, and Jinja lets us make this really dynamic with conditionals; for example, the delete button on a record is only shown if the user is an admin.

Now let’s talk about the system as a whole.

## System

From a very high level, the ISS sends encrypted data to IDA to create a record. Users can access the web interface as I’ve just demonstrated to view the records that have been transmitted.

If the user is an admin, they can edit and delete records and users.

The endpoints are all secured, all require some form of authentication, whether that’s a user logging in or a secret token that the ISS uses to send data.

Some endpoints, as well as the users and logs views, also specifically require that the user is an admin.

## Database

<BRING UP SQLITE DATABASE VIEWER>

Here we can see the four tables that are used. The schema was designed to be clear and easily understood.

The user\_ref table here is used for associating a UUID, a Universally Unique Identifier, to a user by their id.

Let’s go over the ISS microservice now and see how that interacts with IDA.

# Deep Dive – ISS

Let’s start up the ISS microservice now.

<START ISS MICROSERVICE>

So now that the ISS microservice is running we will see that, by default, every 10 seconds a record is sent from the ISS and we see the response from IDA here.

Let’s take a look at the ISS code.

<OPEN ISS/ISS.PY>

Here we see that we’re just crafting some random data.

The name is always prefixed with “RECORD-“, and then has a random alphanumeric string concatenated.

Datetimes are always stored as an int-casted unix timestamp, which just means that the milliseconds portion of the timestamp is truncated (or removed).

Here, the file portion of the record is base64 encoded, which returns a bytes object, then we decode it to a string. This portion can be a bit confusing, hence all of the detailed comments here.

We then:

* compile our random data into a dictionary
* dump it to a JSON object
* encrypt it using Fernet encryption and the shared key that only the ISS microservice and IDA application know
* which turns the whole record into a bytes object that we then decode to a string
* and then we send it over to IDA as a text type, as you can see from the headers here.

This token is used on the IDA endpoint to verify that this request has in fact come from the ISS microservice. We need this because this microservice uses threading to send the requests so as to not block the main process.

<SCROLL TO \_\_MAIN\_\_>

So here we have the main entry, we are starting a thread that will begin a loop of requests to IDA. We start it as a daemon because we want the thread to die when we exit the process.

<SCROLL TO START\_RECURRING\_PROCESS>

While True here basically means we are creating an infinite loop as long as this thread is alive. We acquire the thread lock, call the send\_data function, release the lock, and then delay another request for whatever this environment variable is set to.

<SHOW RECORDS POST ENDPOINT>

Here we can see the records post endpoint, it’s fairly straightforward. It takes that token and verifies that it’s the token that it is expecting, and only allows the record to be inserted into the database if this token is present and valid.

# Deep Dive – IDA

So I want to speed this part up a bit as this has probably already gone longer than I anticipated, so I’m just going to briefly go through the main entry, the endpoints, and the routes.

Here is the main entry for IDA, we’re collecting our environment variables, setting a secret key which the Flask framework uses for sessions, registering our Blueprints which I’ll clarify when I get to routes and endpoints but it’s just so that we can split up our routes into different files and Flask will still know about them.

Then we run the app here.

Let’s look at the models briefly.

<SHOW MODELS>

There’s not much to show here, the LogEvent currently is unfortunately unused but would be used in a more developed prototype.

The routes here are separated into concerns. The pages contains the home, login and logout, and view routes.

The other files here are all for operations on the database basically, let’s look at the most relevant one, the records endpoint.

<OPEN ROUTES/RECORDS.PY>

Here we can see the records endpoints and the allowed methods for each. At the top we have an IP restriction method that uses the “before\_request” decorator, which does what it says on the tin and runs anything inside here before any request is handled to any route on the “records\_bp” blueprint.

In Flask, the best practice is to make I guess sort of “monolith” functions that contain all of the allowed HTTP methods on a particular endpoint.

# Wrap-up

So, this was probably a bit longer of an overview than was expected, but hopefully that gives you an idea of how the system works and a bit of detail into the structure and logic of data and endpoints.

For more detailed information, you can review the updated design spec and diagrams, which are included in the “documents” folder. These have been updated for the second part of this assignment, considering the framework change from Bottle to Flask, among other changes.

Thanks for watching and have a great day!