Getting Started with Illumio Core REST API

Or, How I Learned to Stop Worrying and Love the Illumio API

Jeff Francis

Published 12/1/2017

Table of Contents

About the author

Jeff Francis is a Consulting Systems Engineer at Illumio, and has worked for over 25 years as a network engineer, security engineer, systems administrator, consultant, and software developer in environments ranging from small startups to some of the largest companies with some of the largest data centers in the world. Jeff lives in the farm country of Western Washington along with his wife, three kids, two goats, chinchilla, dog, and two cats named Bob (yes, both of them).

A word of thanks

While only one name goes on the cover of a book, it's never a solo effort. Many people contributed to this volume. Patrick Nolan served as technical editor, and found a truly embarrassing number of mistakes in the original manuscript. Janani Nagarajan reviewed the code for mistakes, and tested to make sure each script actually worked as specified. Finally, Jeff Vargas turned a very large Word file into the polished document you see before you. This book would not exist without each of these people. Any errors in the final draft are mine, not theirs.

Chapter 0: Who This Book is For

Congratulations! You (or somebody in your organization) just bought Illumio, and it's your job to integrate this new product into your internal systems. If you're reading this, you probably have a title that includes the words "DevOps," "SRE," "Sysops," or maybe "Network Engineer" or "Security Engineer." Whatever the title, you're responsible for automation and perhaps integrating new network and security products into existing provisioning systems. You almost certainly write code, and your concern is to make new products as useful and as invisible as possible while integrating them into your infrastructure. If any of this sounds the least bit familiar, this book is for you. This book is about using the Illumio API to become a part of existing systems.

As an engineer, you're almost certainly familiar with the term API, or Application Programming Interface. Depending on your level of programming experience, the idea of using an API may bring joy to your heart or tears to your eyes. While the term API may mean a hundred different things to a hundred-different people, the Illumio ASP REST API is about as straightforward as APIs get. The Illumio API is a RESTful API, and uses JSON over HTTPS (we'll discuss this in more detail in the next chapter). The Illumio ASP REST API is a fully-supported piece of the product, and can be used to automate nearly any task that can be done with the product.

To make the most of the information in this book, there are a few skills you're required to have, and a few more that, if you have them, you'll find the exercises a bit easier. First, you'll need a working knowledge of a reasonably modern scripting language. What's "reasonably modern"? And what's a "working knowledge"? "Reasonably modern" is a language with libraries for HTTP, REST, and JSON. Of course, it doesn't have to be modern, it doesn't have to have libraries, and it doesn't have to be a scripting language. But these features, as embodied in something like Ruby or Python, will make learning and working with the API much easier. This book was written using Ruby for the example code, though you should be able to follow along in most other similar languages, provided you know them and their libraries.

This brings us to "working knowledge." By this, it's intended that you have the ability to write at least some basic scripts without being fed each concept and statement one at a time. While the example code is complete, you'll learn more if you already understand the language itself, and you're not trying to learn both the language and the API at the same time.

You don't need prior knowledge of JSON to use the Illumio API, but you'll have to learn it along the way, if you want to be successful. It's certainly possible to learn it as you go through the examples in this book. It's not hard, and you don't have to be an expert, but you'll need to be familiar with what it is and how it's used to understand what's going on with the Illumio REST API, as JSON is the encoding used for all data transfer (in both directions). Everything sent to and everything received from the API is encoded in JSON. You'll need to know how to convert back and forth from objects in your scripting language to the JSON strings you'll be reading and writing using libraries. The exercises in each chapter will help with this part.

It will also help if you understand the idea of using GET, PUT, POST, and DELETE to manipulate a REST API over HTTP. If you understand CRUD, you're half-way there. While CRUD and REST are not the same thing, they're conceptually in the same ballpark. You can work through the tutorials without a complete understanding of REST APIs, but an understanding will help the knowledge sink in a bit better. As with JSON, the web has many excellent resources for learning the basics. At the very least, read the Wikipedia articles on both REST and JSON before proceeding if you're not familiar with both.

While the code in this book is written in Ruby, this book is not about learning Ruby, nor is it about writing robust software. Rather, the goal is to provide some real-world examples of working with the Illumio ASP's REST API. Examples are deliberately kept as simple as possible to illustrate use of the API, and generally avoid dealing with the consequences of errors in order to keep the scripts simple and readable. While the scripts in this book are useful for learning, they are not intended for production use. They are tools for learning the concepts of the Illumio ASP REST API. Before using the techniques in this book in production, it would be wise to add input checking, check the return values of all functions and API calls, and write error-handling routines to deal with failures.

Chapter 1: Getting Started With the Illumio API

Before beginning our quest for Illumio API Knowledge, there are some things that need to be done to get ready. There is software to install, credentials to obtain, and a plan to talk about.

The very first thing you'll need is the URL of your Policy Compute Engine (PCE). It will be of the form https://pce.yourorganization.example:8443 or something similar. It may or may not have the ":8443" port number (or another number) on the end. You'll need this (both the DNS name and the port number) for every exercise in this book.

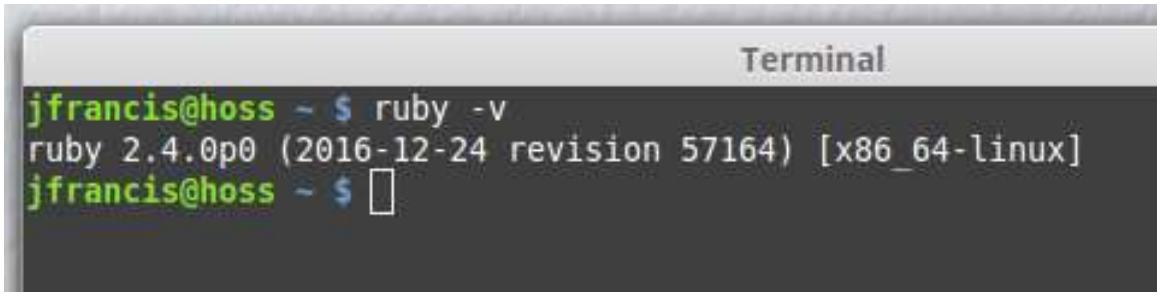
While you're ultimately going to be writing code for production use, it's wise to experiment on non-production servers. This can be a few boxes in a lab, some virtual machines in a private cloud, or maybe some boxes in the public cloud. However you choose to do it, keep in mind that you'll make mistakes while you learn, and it's generally considered poor form to take down a production datacenter while learning to use a new product.

You'll also need credentials. There are two kinds of credentials for the Policy Compute Engine. First are login credentials, which you'll use to log in via the web user interface. These will normally be supplied by the Policy Compute Engine administrator. These credentials can be used to generate a second set of credentials, your API Credentials, which are used when working with the Illumio REST API. Unless someone has already generated these for you, obtaining API Credentials will be your first task in the next chapter.

You'll need a scripting language installed on your system to work with. Unless you have a very solid knowledge of another language, such as Python, and are willing to mentally translate the exercises in the book into that language, you're probably best off using Ruby. Even if you're ultimate goal is to write Illumio API code in another language, if you follow the exercises using Ruby, they're more likely to work correctly the first time.

If you're using a Linux or macOS system, it's entirely possible you already have Ruby installed. If not, it should be simple to install with apt or yum on Linux, or with MacPorts or Brew on macOS. There are a number of options for Windows, though RubyInstaller seems to be the most popular. A quick Google search should

get you started. While it's not strictly required, the examples are geared towards at least Ruby version 2.1 (the example code in this book was developed and tested with Ruby 2.4.0 on Linux and macOS using GNU Emacs 25 as an editor). Older versions of Ruby, such as the commonly-found 1.8, may not be entirely compatible with the examples, as some of the syntax in the language, particularly around hash tables, has changed. When in doubt, open a shell/terminal/command line window and type "ruby –v" and verify the version:



In Ruby, libraries are called "gems". There are two gems you'll need for every one of the following chapters (plus one or two more later on): "json" and "rest-client". The first library is included in a standard Ruby installation (in other words, you don't need to do anything to get it). This is the library that converts Ruby data structures to JSON, as well as parses JSON data back into Ruby data structures. The second library, rest-client, does not come with Ruby by default, and must be added. On a Linux or macOS system, this library is usually added with the command "sudo gem install rest-client". Windows systems will need the same library, though it's usually added via a GUI Gem tool, instead. If prompted, be certain to include all necessary dependencies (though this is the default, and should be handled automatically).

Make sure you've got an editor you're comfortable with. Yes, you could write code with WordPad. You could also listen to the Bee Gees on an 8-track player while wearing a polyester leisure suit while you study this book. But there are more modern choices that make things a bit easier. The most important attribute of a program editor is that you're comfortable with it. If you haven't made a choice, I won't try to make one for you, but it's hard to go wrong with something like Atom. Atom has syntax highlighting, all sorts of automatic indentation and balancing features (if you type an open-parenthesis, it automatically adds a close-parenthesis after your cursor so you don't forget). Most importantly, it lets you run selected bits of code directly from the editor. Other common choices include Eclipse, Text Wrangler, emacs, vi, and yes, WordPad, if you insist. All of these

(except WordPad) are available for Linux, macOS, and Windows. Like any tool, a powerful editor takes some time and effort to learn, but don't overdo it. If you're still struggling more with your editor after the first day than you are with the code you're working on, try a different editor.

Last on our list of things to gather before we start is documentation. You'll want to log into the Illumio Support Portal and download copies of several documents. Upon logging in, click on "DOCUMENTATION". Download the following four PDF files (at a minimum, no reason not to grab all of them while you're logged in if you've got the time) and save them where you can find them easily, because you'll use them often:

- Release Notes
- PCE Web Console User Guide
- REST API
- REST API Schema

We'll use the last two of these (which we'll refer to as the "Illumio REST API Guide" and the "Schema" or "JSON Schema") extensively in the chapters to follow. Release Notes are always worth reading to understand any known issues in the current version of the Policy Compute Engine software, and the PCE Web Console User Guide is the official resource for understanding the Illumio ASP product and how it works, so you'll no doubt refer to it from time to time, as well.

As you look at the Illumio REST API documentation, you'll notice that some of the API calls are listed as "Experimental." There are, in fact, three categories of API calls in Illumio. The majority of these calls have no particular type name, they're simply the fully supported, documented calls that perform the majority of the actions you'll need to take. These API calls are well-tested, well-documented, and intended to remain stable (in other words, to not change) over long periods of time. While new calls may be added to this group and new parameters may be added to existing calls to support new functionality, the old calls and systems are intended to remain relatively static. Software written using these API calls should continue to work across one or more releases of the Illumio Policy Compute Engine. While changes do happen, they tend to be rare and for very good reasons. Any changes are reflected in the Illumio REST API Guide, as well as the Release Notes.

The term “Experimental” in certain API calls should not be cause for concern. These calls are equally well-documented and supported as their more static brethren; they’re labeled “Experimental” because there is potential for change in future releases of the code. You should not be afraid to use these calls, but if you do use them, you should carefully note any changes in the Release Notes and Illumio REST API Guide whenever the Policy Compute Engine is upgraded. These API calls are subject to change without prior notice with new functionality.

There are also Private API calls. These are calls into the Illumio REST API that are made by the web front-end that are not documented and not intended for customer use. While it’s certainly possible to reverse-engineer their functionality using modern web tools, their use is very strongly discouraged. Making use of these undocumented calls may have unintended consequences, and can cause serious issues with your system. These calls also tend to change dramatically, even with very minor software updates, rendering any software you write using them extremely brittle. If you discover some functionality that you require that is not covered by the documented API, please open a ticket on the Illumio Support Portal or talk with your Sales Engineer.

Each of the chapters in this book will start with a real-world problem to solve, then work through the steps of solving that problem using the Illumio REST API. The task will be described at the beginning of the chapter, along with a description of what you’ll learn and cover in that chapter. While a few chapters are stand-alone, most build upon both code and concepts learned in previous chapters. The complete code listing for each chapter is found in the Appendixes in the back of this book.

Attention to detail is key when working with an API. The slightest deviation from the specified format will cause the API to return a 406 error, without explanation as to the specific cause. The JSON structures sent and received from the API can be quite complex, and the tiniest mistakes in structure, while very hard to find visually, will cause the API call to fail when submitted. Here are two examples, the first correct, and the second incorrect. Can you find the error? I can assure you, the API will let you know the second one is broken:

```
{"name"=>"works",
 "enabled"=>true,
 "scopes"=>
 [[{"label"=>{"href"=>"/orgs/9/labels/1136"}},
   {"label"=>{"href"=>"/orgs/9/labels/1135"}},
   {"label"=>{"href"=>"/orgs/9/labels/1134"}}]],
 "rules"=>
 [{"enabled"=>true,
   "providers"=>[{"actors"=>"ams"}],
   "consumers"=>[{"actors"=>"ams"}],
   "service"=>{"href"=>"/orgs/9/sec_policy/draft/services/194"},
   "ub_service"=>{"href"=>"/orgs/9/sec_policy/draft/services/194"},
   "sec_connect"=>false,
   "unscoped_consumers"=>false},
  {"enabled"=>true,
   "providers"=>[{"label"=>{"href"=>"/orgs/9/labels/1084"}}],
"consumers"=>[{"ip_list"=>{"href"=>"/orgs/9/sec_policy/draft/ip_lists/115"
}}],
   "service"=>{"href"=>"/orgs/9/sec_policy/draft/services/247"},
   "ub_service"=>{"href"=>"/orgs/9/sec_policy/draft/services/194"},
   "sec_connect"=>false,
   "unscoped_consumers"=>false}]
}


---


{"name"=>"broken",
 "enabled"=>true,
 "scopes"=>
 [{"label"=>{"href"=>"/orgs/9/labels/1136"}},
  {"label"=>{"href"=>"/orgs/9/labels/1135"}},
  {"label"=>{"href"=>"/orgs/9/labels/1134"}}],
 "rules"=>
 [{"enabled"=>true,
   "providers"=>[{"actors"=>"ams"}],
   "consumers"=>[{"actors"=>"ams"}],
   "service"=>{"href"=>"/orgs/9/sec_policy/draft/services/194"},
   "ub_service"=>{"href"=>"/orgs/9/sec_policy/draft/services/194"},
```

```
    "sec_connect"=>false,
    "unscoped_consumers"=>false},
   {"enabled"=>true,
    "providers"=>[{"label"=>{"href"=>"/orgs/9/labels/1084"}}],
"consumers"=>[{"ip_list"=>{"href"=>"/orgs/9/sec_policy/draft/ip_lists/115"
}}],
    "service"=>{"href"=>"/orgs/9/sec_policy/draft/services/247"},
    "ub_service"=>{"href"=>"/orgs/9/sec_policy/draft/services/194"},
    "sec_connect"=>false,
    "unscoped_consumers"=>false}]
}
```

Here's a hint: The scope should be a list of lists, not a list. API coding is tricky, I tell you. Much like piloting an airplane, using the API isn't hard, but it's very unforgiving of mistakes.

And you will make mistakes. There's a lot of detail to get absolutely perfect, every single time. Getting the JSON correct with each of the intricate hashes, arrays of hashes, and arrays of hashes of arrays is challenging, and I invented at least three new and creative (though anatomically implausible) swear words and phrases while writing Chapter 9 alone (these colorful metaphors have been omitted from the final text of the book). The slightest error in structure, no matter how small, leaves you with a 404, or worse, a 406 error when you submit, so learning the tools available for debugging REST API calls is of vital importance.

So how in the world do you debug something this complex? It's not that hard, actually, if you use the right approach: be patient. It's important to break the problem into bite-sized pieces, and solve each one one at a time. There are some powerful tools available to you. Start with a known good working example of the JSON you want to submit. How? Let's assume you're trying to build the JSON for a Ruleset. First, build the Ruleset you want in the PCE web console, then extract the JSON for that Ruleset using the API and examine it very closely. Use this as a template. Convert the raw JSON back into objects in the language of your choice. For example, in Ruby, a call to JSON.parse (extracted_JSON) will return a set of Ruby objects to use as a starting point. This helps tremendously when trying to understand what's expected. Use this as a template, then start filling in the fields one at a time with your own data and re-submit the JSON using the PCE web console until you get the desired results. When you convert this structure back into JSON, if you've been reasonably careful, it should just work. You'll need to delete the Ruleset (or whatever you're building) between each test of your code, because many things you'll build using the API cannot be duplicates, or else the API will return errors. Later in the book, you'll learn to delete things using the API, but for simplicity when

you're getting started, simply delete them using the PCE web console each time you create something using the API. After a few iterations, you should have working Ruby data structures that produce valid JSON, which in turn produce the desired result in the PCE.

Most of the objects created in the PCE have a large number of parameters that you have no control over, and do not need to specify when creating a new object. For example, nearly everything you can access via the API has an associated user name for the user who created it, the last user to modify it, the timestamp when it was created, the timestamp for when it was last modified, and many other fields that are used internally. While you can certainly query these and use them, you cannot specify them yourself. They are created and updated automatically by the PCE as objects are manipulated. There are other fields, such as the HREF (which we'll talk more about later) that is automatically created when the object is created, and is used for all subsequent references to that specific object. You don't create it yourself, but you do use it extensively once the PCE creates it.

The important thing to understand is that when you create a new object in the PCE using the API, it is not necessary (or even possible) to supply every value for every field. In fact, most objects can be created (though they're not yet useful) by specifying nothing more than their name at creation time. When using the PCE web console to create objects in the PCE that you will then extract via the API and use as templates for new objects, make sure you understand that many of the fields (the name of the creator and modifier, as well as the creation and modification dates, for example) are things you should remove from your template, as they're not values you'll specify yourself. The Illumio REST API Guide tells you what fields you can modify, and the JSON schema files enumerates the allowed values for each field.

When you're trying to create something new via the API and it fails, a useful way of locating the failure is to create the desired object with nothing more than a name (or whatever parameters constitute the minimum for creation of an object of the desired type). It's usually simple enough to complete that step without errors. Once you have that working, delete the new object from the PCE, then add one more element, and try again. If successful, repeat until successful. If any step fails, provided you're only adding one element at a time, it's easy to find where your mistake is. It's the most recently added element. And once you've found where your error is, it's usually not hard to figure out what your error is. Go back to your template where you created what you wanted in the PCE web console, and compare the JSON from that to the JSON you're attempting to submit. The difference is usually quite clear. Learning the patience to change only one thing at a time is by far the most important skill you'll need to be a successful API programmer.

In your scripts, when you make an API call, don't automatically assume success. In fact, plan for failure. There are many reasons an API call might fail, and many of

them are out of your control. Always check return values. Write code that handles exceptions and unexpected values. When writing functions, sanity check the values you've been given and ensure the caller is giving you complete and sane parameters before you pass them to the API. For example, if the API call you're about to make requires a Label, make sure you've been given something that looks like a Label before calling the API with that as a parameter.

The scripts in this book do none of this, and that's on purpose. Handling failures often takes as much as two thirds of the code you'll write, which makes understanding simple snippets of code that much harder. First, make your code work. But don't' skip the step of making it resilient, or it will fail at the worst possible time. With practice, you'll learn to interpret errors. You'll learn what right and wrong looks like when looking at the JSON. And most important, you'll start building your own working, tested functions to start building your own code library.

There are some general guidelines to keep in mind when using the Illumio REST API. The following list will keep you out of trouble, though it's not expected that you will understand every one of these Rules of thumb at this point. As you work through the exercises in this book, this list will begin to make more sense. Refer back to it from time to time, and make sure you understand each of the items before you deploy real software and real production systems:

- Generate a pairing key (Chapter 10: Building Pairing Profiles) and save it instead of creating a new key each time, so long as you don't need the time or usage limitations. Keys can be setup as time-bound or the number of uses can be limited for security. Creating the minimum number of pairing keys makes it easier when it comes time to revoke a key.
- Check return codes. Don't assume an API call was successful unless the API specifically tells you it was.
- Send a maximum of 500 standard API requests per minute, or 10 bulk API requests per minute to avoid overwhelming the PCE. Excessive API requests will result in denied queries until the system catches up. The Illumio API Guide discusses this in depth.
- Use asynchronous queries (Chapter 6: Asynchronous API Calls) if your query will return more than 500 results from an API GET.
- Use resource names (Label names, Workload names, etc.) that are fewer than 255 characters long.
- Let your Illumio Sales Engineer or Customer Success Advisor know if you are using the API. These resources can offer suggestions as well as alert you to any changes that may impact your systems.

Before we jump in and start coding, a word of caution is in order. Your Illumio ASP installation is, by it's very nature, a critical piece of infrastructure, and should be protected as one of the crown jewels of your organization. It controls all access to

and from any infrastructure it protects, and control of Illumio is, in many respects, control of your infrastructure. As a consequence, login credentials to the Policy Compute Engine, as well as API Credentials, should be carefully protected. It's tempting to hard-code credentials into API scripts. We do, in fact, do exactly this in the examples in this book. Resist the temptation to do this with production scripts. Anyone who has the ability to read your scripts has the ability to use your credentials for any purpose.

With that in mind, onwards!

Chapter 2: Generating API Credentials

There are two simple tasks to complete in this chapter. First, we'll use the PCE web console to generate a set of API Credentials (also called an API Key). Second, we'll use that key to fetch our org_href (a value we'll use in all following chapters) from the Illumio REST API.

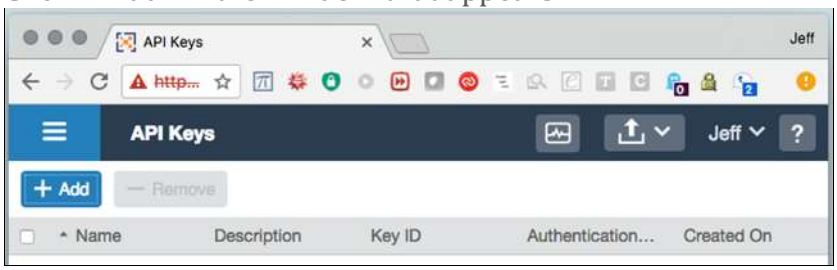
Let's start by generating an API key. Note that as of PCE version 17.2, it's not possible for SAML-authenticated users to create API keys. Keys can only be created by users who are locally authenticated to the PCE. Additionally, any API keys that were generated by a SAML-authenticated user on a PCE prior to an upgrade to 17.2 will be deleted at the time of the upgrade. SAML-authenticated users *can*, however, *use* keys that were generated by locally-authenticated users.

When using a REST API (as well as other types of APIs), an API key is analogous to a username and password, and consists of an "Authentication Username" (or login name) and a "Secret" (or password). In the Illumio world, once an API key is created, it's valid for use until it's deleted or disabled from the PCE. Any administrator of the PCE has the ability to create and delete API keys. Here's how to create an API key using the PCE web console (note that it's also possible to create an API using the API itself – see the Illumio REST API Guide for details):

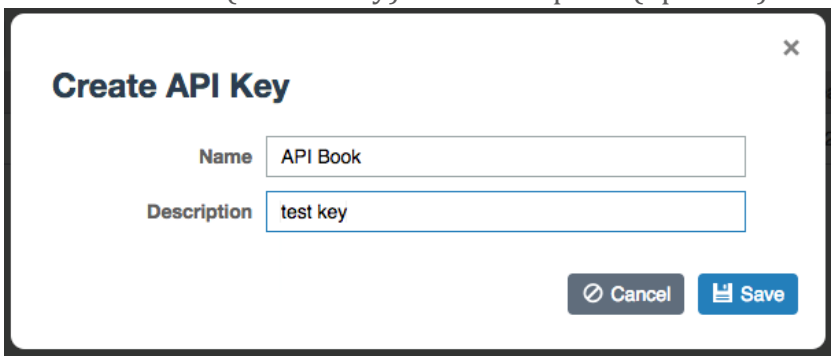
1. Log into the Illumio PCE web console using a web browser:

2. Click your name in the top right corner:

3. Click “My API Keys” in the drop-down list.
4. Click “+Add” in the window that appears:

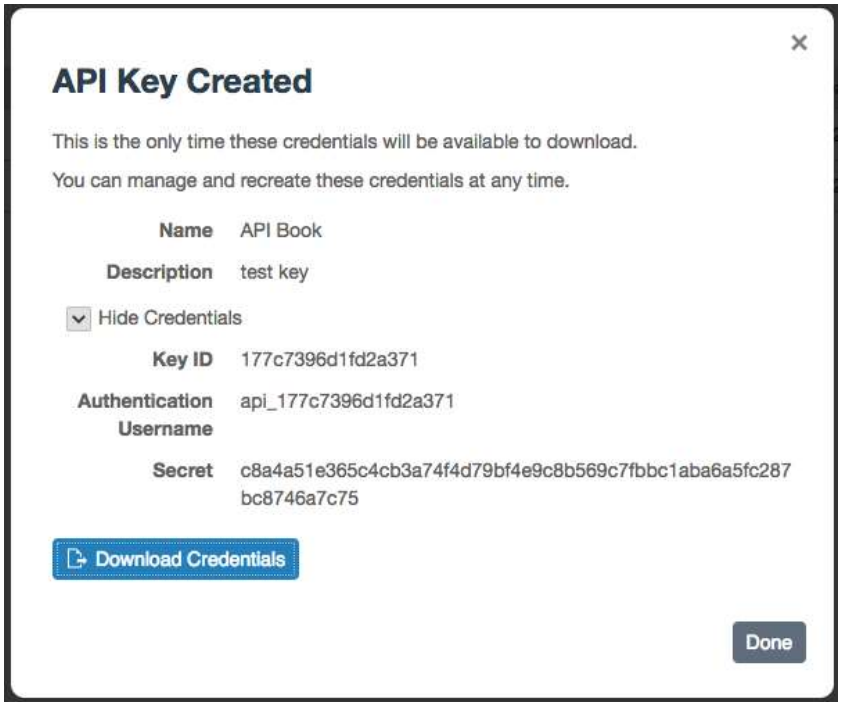


5. Fill in the Name (mandatory) and Description (optional):



6. Click Save.

7. Click “Show Credentials”:



8. Copy and paste this information into a safe place. There is no way to retrieve it once you close this screen.
9. Optionally, you may choose to download the credentials into a file by clicking on “Download Credentials”. Keep this file in a very safe place.

The Authentication Username will take the form of:

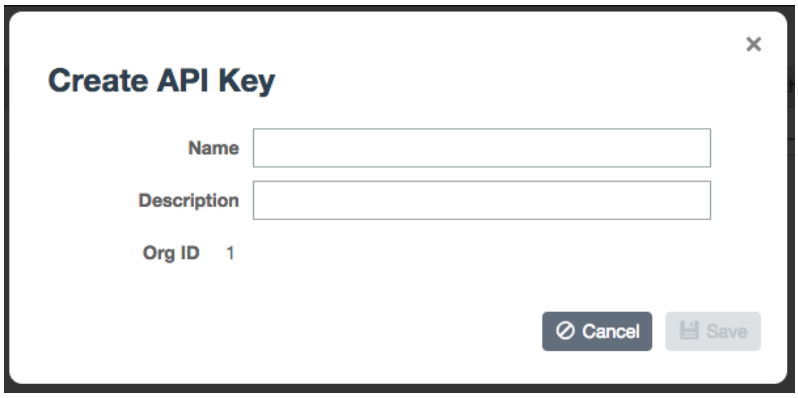
```
“api_177c7396d1fd2a371”
```

The Secret will resemble:

```
“c8a4a51e365c4cb3a74f4d79bf4e9c8b569c7fbbc1aba6a5fc287bc8746a7c75”
```

These are your API credentials. As noted in the introduction, this information should be kept in a very secure place, as use of these credentials allow for full control of your network security policy.

The remainder of this chapter is about writing our first API script to fetch our “org_href”, which is the third piece of the required information for using the Illumio PCE via the REST API. As of release 17.3 of the software, however, this information is now provided at the time you create your API key. Note the difference in the “Create API Key” dialog box below with the one illustrated in Step 5 above:



While the rest of this chapter is useful for introducing you to the basic concepts of using the API, if you're running release 17.3 or newer of the software, it's no longer strictly necessary to follow the steps below to retrieve your org_href. You can simply record it at the time you create your API key. In this example, your org_href is "1".

Assuming you're running a release older than 17.3 (or if you'd just like to get your first experience with writing code for the API), we're ready to write our first API script. This first script isn't particularly exciting, but there's information we need in order to use the rest of the API calls called the "org_href", and this first script will retrieve that value for us. The PCE server software was designed to support multiple customers on the same server, and customers are grouped into organizations which are each given an org_href to identify them uniquely. If you're running your own PCE on-site, your org_href will almost always be "/orgs/1". If you're using Illumio Secure Cloud (the Illumio SaaS service for your Policy Compute Engine), however, the integer part of this string could be most anything. Our first task is to learn our org_href by logging into the API and retrieving the value that's been assigned.

Start by opening your text editor of choice. I suggest creating a new script for each chapter so you can refer back to earlier work later (it saves typing when you can copy and paste from your own old scripts). Start with the following three lines at the top of the file:

```
#!/usr/bin/env ruby
require 'json'
require 'rest-client'
```

The first line tells Linux and macOS systems to run this as a Ruby script. It's not strictly necessary on Windows systems, but is still considered good practice (to keep your scripts portable). The second and third lines load the two required Ruby libraries, JSON and rest-client. You'll get an error when you run the script if the Ruby

interpreter is unable to find the library you requested. First, make sure you spelled it right. Accuracy counts. If you did, and it still won't load, do a bit of Google searching for debugging this problem (it's out of the scope of this guide).

Next, add lines containing your PCE login and password. Note that you use your actual PCE credentials here (the same ones you use to log into the PCE web console), not your API credentials you just created. This is the only script you'll write that uses login credentials instead of API credentials, but it's necessary to do so in order to retrieve your org_href.

```
login = "jeff.francis@illumio.com"
passwd = "SecretPassword"
```

As a final bit of setup, you need to tell the script how to find your Policy Compute Engine. This will be the URL you use to log into the PCE via the PCE web console. We'll set a variable to the DNS name of your PCE, without the "https" and slashes bit at the front. In order for the script to work, you'll also need to specify a second variable specifying the port your PCE is running on. If your PCE is running on a non-default port, the port number will be found immediately following the host name of your PCE (for example, "https://illumio.mycompany.com:8443" – 8443 is your port name). If there's no number following the host name in the URL where you log in, it's port 443 (make sure you have either 443 or your specific port in the line below, or it won't work – and don't forget the quotation marks, it won't work without them):

```
pce = "illumio.mycompany.com"
port = "443"
```

Now we're ready to make the API call to the PCE to retrieve the org_href value. If you look in the Illumio REST API Guide, it tells you that a POST to the specified URI with a parameter named "pce_fqdn" set to the DNS name of the PCE will generate an auth_token, which in turn can be supplied to a second API call to gather information on the user whose login credentials were supplied:

The Ruby statement below does this, using the variables we set earlier. It constructs a REST query, executes it, and sets the variable "response" to the data returned from the PCE. There are some important things to note in constructing the code for our REST query. First, we specify the URL we're going to query. We construct the URL by piecing together the specific DNS name and port number of your PCE, and supply the variable "pce_fqdn" set to the appropriate value. We also specify that this will be an HTTP POST operation, the login and password information (the Illumio API uses BasicAuth), and sets headers in the HTTP transaction that specify that we're sending information in JSON format, and expect our result in JSON, as well:

```
response = RestClient::Request.new(:url =>
"https://"+pce+":"+port+"/api/v1/login_users/authenticate?pce_fqdn="+pce,
                                    :method => :post,
                                    :user => login,
                                    :password => passwd,
                                    :accept => :json,
                                    :content_type => :json).execute()
```

The response we get back from the PCE is a chunk of JSON that consists of a single key:value pair. Let's first turn the raw JSON into something ruby can work with. We'll use the JSON parsing library to parse this line, and set the variable "json" to the result:

```
json = JSON.parse(response)
```

The key in the key:value pair returned is "auth_token". Let's set the variable auth_token to the value part of this key:value pair:

```
auth_token = json['auth_token']
```

auth_token is a very long cryptographically-generated token (or string) that we'll use for the next step. We'll use this string to authenticate against the API, which will return a great deal of data to us, including the "org_href" that we need for further API work. In order to retrieve this data, we'll have to resubmit this token to another API call:

URI

```
GET [api_version]/users/login
```

Params

Login Service authentication token you obtained using the Login Users API.

Much like the previous call, there is one supplied parameter, though in this case, it's a GET to the API (instead of a POST), and the parameter is specified as a header value, rather than a POST payload. Specifically, the call wants a header named "Authorization" with a value of "Token token=<value returned from the previous API call>".

This call returns considerably more information than the last, which we'll dig into below. Once again, we'll create a REST query, execute it, set the returned value to a variable, then parse the raw JSON into useful Ruby structures we can query:

```
response = RestClient::Request.new(:url =>
"https://"+pce+":"+port+"/api/v1/users/login",
                                    :method => :get,
                                    :user => login,
                                    :password => passwd,
                                    :accept => :json,
                                    :content_type => :json,
                                    :headers => {"Authorization" => "Token
token="+auth_token}).execute()
json = JSON.parse(response)
```

Let's take a quick peek at what was returned. Here's the blob of JSON we got back:

```
{"full_name":"Jeff
Francis","local":true,"type":"local","href":"/users/23","auth_username":"u
ser_23","inactivity_expiration_minutes":10,"start":"2017-05-10 18:56:17
UTC","time_zone":"America/Los_Angeles","last_login_ip_address":"172.76.134
.28","last_login_on":"2017-05-
10T18:56:17.000Z","certificate":{"expiration":"2018-01-
17T23:59:59.000Z","generated":false},"login_url":"https://demo8.illum.io:4
43/login","orgs":[{"org_id":9,"org_href":"/orgs/9","display_name":"nobody"
,"role_scopes":[{"role":{"href":"/orgs/9/roles/owner"},"scope":[],"href":"
/orgs/9/users/23/role_scopes/27"}]}],"session_token":"158ac3d95983a06e8234
3a074d6015cb07eeff51c","version_tag":"60.0.0-
860323e3c59cc4e866e43f054529576660e456d6","version_date":"Thu Apr 13
15:48:16 2017 -
0700","product_version":{"version":"17.1.0","build":"5194","long_display":
"17.1.0-5194","short_display":"17.1.0"}}
```


That's pretty dense. Fortunately, when properly formatted (do a quick Google search for JSON formatters if you'd like to be able to do this yourself), it's considerably more readable:


```
{
   "full_name":"Jeff Francis",
   "local":true,
   "type":"local",
   "href":"/users/23",
   "auth_username":"user_23",
   "inactivity_expiration_minutes":10,
   "start":"2017-05-10 18:56:17 UTC",
   "time_zone":"America/Los_Angeles",
   "last_login_ip_address":"107.178.244.221",
   "last_login_on":"2017-05-10T18:56:17.000Z",
   "certificate":{
      "expiration":"2018-01-17T23:59:59.000Z",
      "generated":false
   },
   "login_url":"https://illumio.mycompany.com:443/login",
   "orgs":[
      {
         "org_id":9,
         "org_href":"/orgs/9",
         "display_name":"nobody",
         "role_scopes":[
```

```
            {
               "role":{
                  "href":"/orgs/9/roles/owner"
               },
               "scope":[

               ],
               "href":"/orgs/9/users/23/role_scopes/27"
            }
         ]
      }
   ],
   "session_token":"158ac3d95983a06e8a343a074d6015cb07eeff51c",
   "version_tag":"60.0.0-860323e3c59cc4e866e43f054529576660e456d6",
   "version_date":"Thu Apr 13 15:48:16 2017 -0700",
   "product_version":{
      "version":"17.1.0",
      "build":"5194",
      "long_display":"17.1.0-5194",
      "short_display":"17.1.0"
   }
}
```


There's a ton of useful data inside of the returned JSON. We're looking for "org_href", which is found about half-way down the returned data. If we de-construct the JSON, we see that "org_href" is one of a number of key:value pairs in a group. The group containing this pair is part of a list (a list with only one group in it), which is the value of the "orgs" key:value pair. Let's look at this as code. We already parsed the returned JSON above, and assigned it to the variable "json". Using this, we'll first extract the contents of "orgs" from the returned JSON:

```
irb(main):015:0> json['orgs']
=> [{"org_id"=>9, "org_href"=>"/orgs/9", "display_name"=>"nobody",
"role_scopes"=>[{"role"=>{"href"=>"/orgs/9/roles/owner"}, "scope"=>[],
"href"=>"/orgs/9/users/23/role_scopes/27"}]}]
irb(main):016:0>
```

The value of the "orgs" key:value pair is a list. Inspecting the list, we see that there's only one item in it (another hash of key:value pairs). Let's extract the first (and only) item in the list:

```
irb(main):016:0> json['orgs'][0]
=> {"org_id"=>9, "org_href"=>"/orgs/9", "display_name"=>"nobody",
"role_scopes"=>[{"role"=>{"href"=>"/orgs/9/roles/owner"}, "scope"=>[],
"href"=>"/orgs/9/users/23/role_scopes/27"}]}
irb(main):017:0>
```

This looks almost exactly like the last result, only it's now not a list containing a hash, it's just a hash. Now we can extract the org_href from the key:value pair "org_id":

```
irb(main):017:0> json['orgs'][0]['org_href']
=> "/orgs/9"
irb(main):018:0>
```

"/orgs/9". That's the bit of data we'll need for all future API calls, along with the API key. It wasn't easy to get, but it's a critical value that we'll use in every subsequent API call we ever make, so extracting it was both necessary and worthwhile.

The following two lines in your script will extract that value and print it:

```
org_id = json['orgs'][0]['org_href']
puts("Your org href is: #{org_href}")
```

In the next chapter, we'll take the API key we generated earlier along with the org_href we just obtained and use them to start making API calls. We'll also talk a bit about HREFs in general, which are the Illumio APIs "handles" used to refer to most everything in the API. This next chapter will be all about extracting data from the API. Following that, there will be chapters where we insert some data into the API. Finally, we'll follow that up with practical examples that automate previously tedious tasks by making use of the API.

It's important to note that you'll have problems with not only this script, but all scripts in this book if you're using a self-signed cert to secure your PCE. There are

several ways to work around this, the easiest being to add the following parameter to every RestClient::Request call:

```
:verify_ssl => false
```

This additional parameter will tell the SSL/TLS library to ignore bad or otherwise non-compliant certs. This is very strongly discouraged for production systems. If certs are not properly validated, there's very little point in using them in the first place.

The full source code for this chapter can be found in Chapter 2 Source Code.

Chapter 3: Extracting Data

In this chapter, we're going to write a script to extract some data from the Policy Compute Engine using the API. Specifically, we're going to extract a list of all Labels along with their types and their HREFs (their global identifiers). We're also going to start writing a few functions that will make our work with the API easier and a little bit more straightforward than what we did in the previous chapter. For this chapter, as well as all subsequent chapters, you'll need your API Credentials as well the org_href from Chapter 2: Generating API Credentials.

If you don't already have some Labels defined in your Policy Compute Engine, log in via the PCE web console and create a few (once logged in, click the three bars in the top left corner (commonly referred to as "the hamburger"), then choose "Policy Objects -> Labels"), and go crazy.

The first few lines should be almost the same as the previous chapter, except instead of using your PCE login credentials, we're going to use our API key credentials (which is what we'll be using from now on). Also, we've added the org_href (remember to put it in quotation marks):

```
#!/usr/bin/env ruby


require 'json'
require 'rest-client'


login = "api_1ea0799f8bd486c8e"
passwd =
"6c1dbf2971aa1ff1a45b142fa86fc6040eeb9bbbe1957d4a977225bdf02d400d"
pce = "illumio.mycompany.com"
port = "443"
org_href = "/orgs/9"
```

We want to extract Labels, so a quick look through the Illumio REST API Guide leads us to this entry, which tells us how to work with Labels using the API:

Labels

Functionality	HTTP Verb	URI
Get a collection of Labels	GET	[api_version][org_href]/labels
Get an individual Label	GET	[api_version][label_href]
Create a Label	POST	[api_version][org_href]/labels
Update a Label	PUT	[api_version][label_href]
Delete a Label	DELETE	[api_version][label_href]

We want to fetch the entire collection of Labels from the API, so we'll use the query from the first line in the table above ("Get a collection of Labels"). The table tells us that this is an HTTP GET and specifies the URI to use. The API version is always "1", which is expressed as "/api/v1", and we'll use our org_href value of "/orgs/9" from our previous blog post (obviously, you'll substitute your own here). Putting these together results in the string we submit to the API:

```
/api/v1/orgs/9/labels
```

This, of course, needs to be combined with the PCE's DNS name, resulting in a full URI like this:

```
https://illumio.mycompany.com:443/api/v1/orgs/9/labels
```

There is no payload to send, we simply do an HTTP GET on the URI above, and the API will return the data we need encoded in JSON. Here's the Ruby code that requests the data from the Policy Compute Engine and assigns that data to a variable:

```
response = RestClient::Request.new(:url =>
"https://"+pce+":"+port+"/api/v1"+org_href+"/labels",
                                   :method => :get,
                                   :user => login,
                                   :password => passwd,
                                   :accept => :json,
                                   :content_type => :json).execute()
```

...which we'll immediately parse using the JSON library:

```
json = JSON.parse(response)
```

Let's have a look at what was returned (I'm going to leave out a few of the Labels for brevity):


```
[
   {
      "href":"/orgs/9/labels/1084",
      "key":"role",
      "value":"Nginx_LB",
      "created_at":"2017-02-17T00:57:00.731Z",
      "updated_at":"2017-02-17T00:57:00.731Z",
      "created_by":{
         "href":"/users/23"
      },
      "updated_by":{
         "href":"/users/23"
      }
   },
   {
      "href":"/orgs/9/labels/1087",
      "key":"env",
      "value":"Development",
      "created_at":"2017-02-17T00:57:39.808Z",
      "updated_at":"2017-02-17T00:57:39.808Z",
      "created_by":{
         "href":"/users/23"
      },
      "updated_by":{
         "href":"/users/23"
      }
   }
]
```

Ignoring some of the key:value pairs, like creation and update timestamps and users, the returned JSON is actually quite simple. We get an array of Labels, each containing various key:value pairs. Each Label has a "handle," which we call the HREF. The HREF is how this specific Label (and in fact, nearly every object used in the PCE) is referred to throughout the rest of the API and can be thought of as a global identifier. HREFs are very important in the Illumio REST API. For example, when specifying a list of Labels for a Workload, the Labels are specified as a list of HREFs, each pointing to a specific Label. A Label has a type, which the API refers to as the "key". These are the same four types found in the PCE web console: "role", "app", "env", and "loc", which are the API representations for Role, Application, Environment, and Location. The name of a Label is called the "value" in the API. This is the English name for the Label that's used in the User Interface.

Now let's write some code to iterate through our list and produce a nice summary of each Label:

```
json.each do |j|
  puts "href: #{j['href']}"
  puts "type: #{j['key']}"
  puts "name: #{j['value']}"
  puts ""
end
```

When we run the script, we get:

```
jfrancis@hoss ~/api $ ./api_lesson_2.rb
href: /orgs/9/labels/1084
type: role
name: Nginx_LB

href: /orgs/9/labels/1087
type: env
name: Development

jfrancis@hoss ~/api $
```

That's it! It's that simple to extract objects from the Illumio API. There are some advanced topics we didn't touch on, such as pulling data from the API when there are more than 500 results to a query. That requires the use of the Asynchronous API calls, which we'll cover in a future chapter. In the next chapter, we'll look at pushing data into the system via the API by adding some Labels that we create in an Excel spreadsheet.

The source code for this chapter can be found in Chapter 3 Source Code.

Chapter 4: Inserting Data

The task in this chapter is the opposite of that in the last. Instead of extracting Label information from the Policy Compute Engine using the API, we're going push Label information into the PCE via the API. In other words, we're going to create some new Labels.

Copy and paste the following lines from the previous chapter's code to get started:

```
#!/usr/bin/env ruby


require 'json'
require 'rest-client'


login = "api_1ea0799f8bd486c8e"
passwd =
"6c1dbf2971aa1ff1a45b142fa86fc6040eeb9bbbe1957d4a977225bdf02d400d"
pce = "illumio.mycompany.com"
port = "443"
org_href = "/orgs/9"
```

Here's another look at the table from the Illumio REST API Guide regarding Label calls in the API. In this case, we're interested in the "Create a Label" line:

Labels

Functionality	HTTP Verb	URI
Get a collection of Labels	GET	[api_version][org_href]/labels
Get an individual Label	GET	[api_version][label_href]
Create a Label	POST	[api_version][org_href]/labels
Update a Label	PUT	[api_version][label_href]
Delete a Label	DELETE	[api_version][label_href]

We'll be using a POST, and submitting a JSON data structure representing the new Label to the same URL as last time:

```
/api/v1/orgs/9/labels
```

The interesting challenge this time around is that we need to create a JSON object to represent the new Label we want to create, which we'll submit as the payload of the POST operation. Creating a Label using the API is very simple, and there are only two mandatory parameters: the key and the value (or, in English, the Type and the Name). The other values (the HREF, as well as the user and timestamp information) are created by the PCE when the Label is created; they're not something we supply. How do we know this? Again, we refer to the Illumio REST API Guide:

Request Properties

Property	Description	Data Type	Required
key	This string indicates the type of Label you want to get, of which there are four types: app, env, role, loc	String	Yes
value	Value of the Label that you are updating for the specified Label.	String	Yes
external_data_set	The data source from which the resource originates. For example, if this Label's information is stored in an external database.	String	No
external_data_reference	A unique identifier within the external data source. For example, if this Label's information is stored in an external database.	String	No

That's pretty straightforward. There are two mandatory parameters (key and value) and two optional parameters we're not going to bother with (they can be used to store arbitrary values, but we have no need for them in this exercise). So how to create a JSON object in ruby? The easiest way is to start with a hash containing the key:value pairs, then use the JSON library to transform that hash into a JSON string. In this specific case, we need a hash containing two values, "key" and "value". For this exercise, let's create a Role Label with the name "web". We do this by creating a hash with the two required parameters as keys, and the value for those keys as their respective values, then converting the hash to JSON:

```
irb(main):003:0> {"key" => "role", "value" => "web"}.to_json
=> "{\"key\":\"role\",\"value\":\"web\"}"
irb(main):004:0>
```

That's the payload we need to supply to the REST call. Note that the JSON we create is submitted as the ":payload" parameter to the REST call. This should look somewhat familiar:

```
response = RestClient::Request.new(
  :url => "https://"+pce+":"+port+"/api/v1"+org_href+"/labels",
  :payload => {"key" => "role", "value" => "web"}.to_json,
  :method => :post,
  :user => login,
  :password => passwd,
  :headers => {:accept => :json,
               :content_type => :json}).execute()
```

When you execute the statement above, you receive some JSON back from the API in response to your POST (see below). The JSON you receive back should look almost exactly like what we extracted in the previous chapter. The API returns the new Label object with some additional fields filled in (the key and value fields, which we supplied, of course, as well as the HREF that was assigned and some ownership and date values).

As your scripts get more advanced, you'll find that it's worthwhile to parse and store data from API calls for later use. For example, storing the returned HREF along with the type and name of the Label saves you the effort of having to look that data up next time you need it in your script (to use a newly-created Label on a Workload, for example). Note that you can only create a given Label once. If you try to create another with the same type (key) and name (value), the API call fails with an error (and not a particularly useful error, at that). Below is a success, then failure (due to duplicate Labels). Obviously, as you progress in your script writing, you'll want to handle these errors and take appropriate action. The first time, it works as expected:

```
jfrancis@hoss ~/api $ ./api_lesson_3.rb
{"href":"/orgs/9/labels/1124","key":"app","value":"test14","created_at":"2
017-05-11T16:36:08.852Z","updated_at":"2017-05-
11T16:36:08.852Z","created_by":{"href":"/users/23"},"updated_by":{"href":"
/users/23"}}
jfrancis@hoss ~/api $
```

The second time, we get an error:

```
jfrancis@hoss ~/api $ ./api_lesson_3.rb
/home/jfrancis/.rbenv/versions/2.4.0/lib/ruby/gems/2.4.0/gems/rest-client-
2.0.2/lib/restclient/abstract_response.rb:223:in
`exception_with_response': 406 Not Acceptable (RestClient::NotAcceptable)
from /home/jfrancis/.rbenv/versions/2.4.0/lib/ruby/gems/2.4.0/gems/rest-
client-2.0.2/lib/restclient/abstract_response.rb:103:in `return!'
from /home/jfrancis/.rbenv/versions/2.4.0/lib/ruby/gems/2.4.0/gems/rest-
client-2.0.2/lib/restclient/request.rb:809:in `process_result'
from /home/jfrancis/.rbenv/versions/2.4.0/lib/ruby/gems/2.4.0/gems/rest-
client-2.0.2/lib/restclient/request.rb:725:in `block in transmit'
from
/home/jfrancis/.rbenv/versions/2.4.0/lib/ruby/2.4.0/net/http.rb:877:in
`start'
from /home/jfrancis/.rbenv/versions/2.4.0/lib/ruby/gems/2.4.0/gems/rest-
client-2.0.2/lib/restclient/request.rb:715:in `transmit'
from /home/jfrancis/.rbenv/versions/2.4.0/lib/ruby/gems/2.4.0/gems/rest-
client-2.0.2/lib/restclient/request.rb:145:in `execute'
from ./api_lesson_3.rb:20:in `<main>'
jfrancis@hoss ~/api $
```

The API returns a 406 “Not Acceptable” response. Understanding these errors is not intuitively obvious, but you’ll get the hang of parsing errors as you progress (and the codes that the PCE returns are all documented in the Illumio REST API Guide). In this case, the 406 error is referring to the attempt to create a duplicate Label.

We’ve got all the pieces now to create a useful script to build Labels for us. In the next chapter, we’ll do exactly that.

The source code for this chapter can be found in Chapter 4 Source Code.

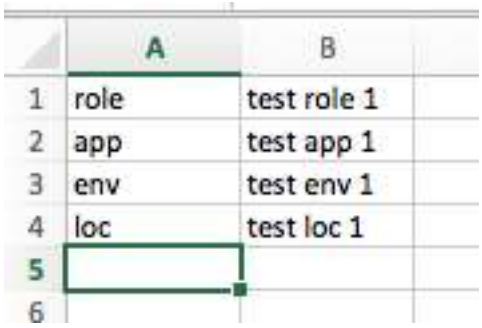
Chapter 5: From Spreadsheets to Labels

This chapter will expand on the last, and is an example of the first truly useful tool we'll write. Rather than adding a single hard-coded Label using the Illumio API, the code in this chapter will load a list of Label names and types from a spreadsheet, then create a Label for each line in the spreadsheet using the Illumio REST API.

This chapter will require a new Ruby gem (called "csv") and will require a file of CSV data representing the Labels to be created. This file can be created manually using a text editor, but is easier to create using Excel or one of the open source spreadsheets such as Open Office.

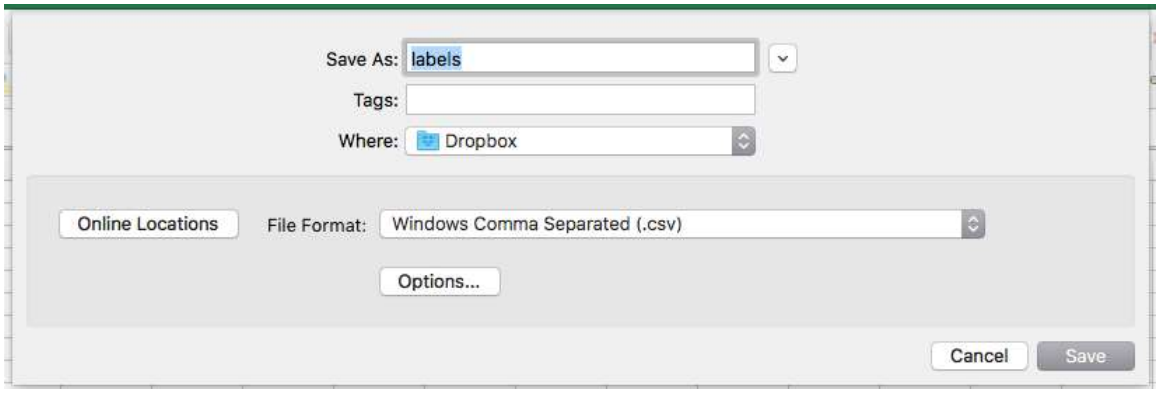
Let's start with the CSV file. As stated above, this is most easily created using a spreadsheet, though you can certainly create a CSV file using the text editor of your choice. In this tutorial, we'll do it the easy way, using Excel, then exporting to CSV. The spreadsheet will be very simple, containing only two columns: "type" and "name". The "type" column will contain the types "role", "app", "env", and/or "loc". The "name" column will contain the name of the desired new Label.

For this very simple example, I've chosen to forgo column names at the tops of the columns, as this is data we'd have to later remove in our code (in order to prevent attempting to create a Label with the type "type" and name "name"). More advanced code could automatically detect and discard column header names. In my spreadsheet, I chose to create one Label of each of the four types with descriptive names (to help ensure that the type matches the name and easily find bugs in my program). My spreadsheet looks like this:



	A	B
1	role	test role 1
2	app	test app 1
3	env	test env 1
4	loc	test loc 1
5		
6		

While there are certainly ruby gems (libraries) capable of reading raw Excel spreadsheets, we're going to keep this simple, and export the spreadsheet as CSV data, as CSV data is considerably easier to parse than an xls/xlsx file. Exporting a spreadsheet as a CSV varies depending what spreadsheet program (and what version) you're using, but for Excel, you save your spreadsheet as a CSV by clicking "File" then "Save As…". Name the file "labels", and choose the file format. Don't choose anything with "UTF-8" in the "File Format" field. We want the output to be plain, straight ASCII:



Once created, copy the labels.csv file to the same directory where you've created your scripts, as the simple script we're writing won't be smart enough to look for the input file anywhere but the current directory (expanding this feature, along with doing some error checking on both the spreadsheet data and the API results, would be a great exercise to increase the utility of the script).

Let's start with a few lines of Ruby to open the file, read the records, and break them into their constituent pieces. While we could probably get away with using split() to break apart our CSV lines, we'll use the csv gem, as that's what it's designed for (and it handles some oddball use cases, like having commas embedded in the text):

```
require 'csv'

File.open("labels.csv").each do |line|
  CSV.parse(line) do |stuff|
    puts "type: #{stuff[0]} name: #{stuff[1]}"
  end
end
```

This is just a quick loop to read in the data. It opens the input file (remember, it's only smart enough to look in the current directory, and it requires the name to match perfectly), then iterates through each line of the file, one at a time. For each line, the csv Ruby gem breaks the line into "chunks," based on the commas, and stores the resulting data into the array "stuff". As there are two items per line (at least there are supposed to be), stuff should contain the type in stuff[0], and the name in stuff[1]. As before, this is a great place to add some error checking in future versions of this script. It's probably also wise to check that type is one of the four allowed types ("role", "app", "env", or "loc"), else the API insert will fail. When we run this code, we get the following output:

```
jfrancis@hoss ~/api $ ./api_lesson_4.rb
type: role name: test role 1
type: app name: test app 1
type: env name: test env 1
type: loc name: test loc 1
jfrancis@hoss ~/api $
```

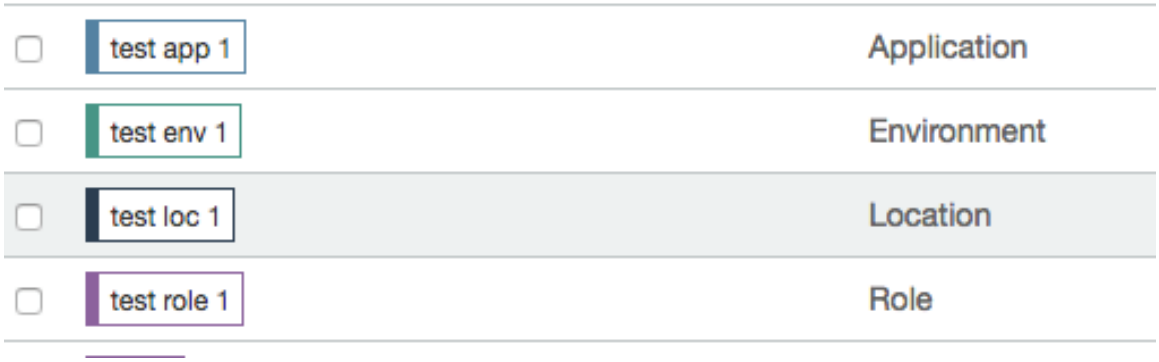
Looks good, yes? Now it's time to combine this code with the code from the previous chapter, to generate Labels. While we're at it, we're going to make the call and parse the resulting JSON all in one go, rather than as separate steps. We'll also print the JSON returned by the API for each Label, just as a sanity check. In more advanced code, you could parse the returned JSON and make note of the returned HREF, for example:

```
File.open("labels.csv").each do |line|
  CSV.parse(line) do |stuff|
    json = JSON.parse(
      RestClient::Request.new(
        :url => "https://"+pce+":"+port+"/api/v1"+org_href+"/labels",
        :payload => {"key" => stuff[0], "value" =>
stuff[1]}.to_json.to_str,
        :method => :post,
        :user => login,
        :password => passwd,
        :headers => {:accept => :json,
                     :content_type => :json}).execute())
    puts json
  end
end
```

And there you have it. When you run the code, you get something along the lines of the following output:

```
jfrancis@hoss ~/api $ ./api_lesson_4.rb
{"href"=>"/orgs/9/labels/1125", "key"=>"role", "value"=>"test role 1",
"created_at"=>"2017-05-11T19:03:25.359Z", "updated_at"=>"2017-05-
11T19:03:25.359Z", "created_by"=>{"href"=>"/users/23"},
"updated_by"=>{"href"=>"/users/23"}}
{"href"=>"/orgs/9/labels/1126", "key"=>"app", "value"=>"test app 1",
"created_at"=>"2017-05-11T19:03:25.503Z", "updated_at"=>"2017-05-
11T19:03:25.503Z", "created_by"=>{"href"=>"/users/23"},
"updated_by"=>{"href"=>"/users/23"}}
{"href"=>"/orgs/9/labels/1127", "key"=>"env", "value"=>"test env 1",
"created_at"=>"2017-05-11T19:03:25.651Z", "updated_at"=>"2017-05-
11T19:03:25.651Z", "created_by"=>{"href"=>"/users/23"},
"updated_by"=>{"href"=>"/users/23"}}
{"href"=>"/orgs/9/labels/1128", "key"=>"loc", "value"=>"test loc 1",
"created_at"=>"2017-05-11T19:03:25.797Z", "updated_at"=>"2017-05-
11T19:03:25.797Z", "created_by"=>{"href"=>"/users/23"},
"updated_by"=>{"href"=>"/users/23"}}
jfrancis@hoss ~/api $
```

If you log into your PCE and have a look, you should see the four new Labels you just created:



Congratulations! You've just automated a fairly tedious task (at least if you're doing dozens or hundreds of Labels) using the API. In following chapters, we'll look at some more complex issues, such as asynchronous system calls and creating and labeling Workloads.

The source code for this chapter can be found in Chapter 5 Source Code.

Chapter 6: Asynchronous API Calls

Up until now, we've been using Synchronous API calls. This is fine for many tasks, but if you expect a result set back from an API call that results in more than 500 items, you'll need to switch to Asynchronous API calls. This chapter is about when and how to make these calls.

In Chapter 3: Extracting Data, we learned that when doing a GET via the API, the maximum number of results returned is 500. If there are more than 500 of the item being queried in the system (Workloads, Rulesets, Labels, etc.), only the first 500 are returned. This is to prevent excessive load on the Policy Compute Engine, as doing large queries on the back-end database consumes a lot of system resources, and it's easy to cause performance issues when executing large numbers of API calls that return large datasets. Excessive load, in turn, causes updates to the firewall policy for the Workloads in your network to slow down, resulting in a less secure network.

So how do we solve this? The Illumio answer is asynchronous API calls. Synchronous API calls (the ones we've used so far) are analogous to buying fries at your local McDonalds drive-through. You drive up to the window, order your fries, receive your fries, and drive away happily munching your wonderful salty fried potato snack. It was a simple, quick transaction where you asked for what you wanted, and immediately received it.

Asynchronous API calls can be compared with dropping your dry cleaning at your local shop. You drop off your shirts, and you're given a ticket to claim them when they're done. The person behind the counter gives you an estimate of when your order will be ready, along with a phone number to check their status before you drive all the way over. Once the estimated time has passed, you call the dry cleaners, verify that your order is done, then return to the shop for your nice clean shirts.

Normally, you make a call to the API, and a short time later (typically measured in tens of milliseconds), the result of your call is returned. With asynchronous calls, it's a bit different. You make a call with a special header flag requesting an asynchronous result, and instead of the result of your query, you get back a special URL. You then periodically check the status of that URL while your query runs at a reduced priority in the background (allowing for normal high-speed processing of security policy in the Policy Compute Engine). Once your query is done, the result of checking the special URL will be yet another URL. If you do a GET on this third URL, you get the result for the original query.

Granted, this is a bit more work than a simple call, but this method allows you to make large queries while maintaining system performance, and putting all of this code into a function hides the complexity, making asynchronous API calls just as easy as synchronous calls (though considerably slower).

There are seven items in the current version of the PCE that can be retrieved asynchronously, per the Illumio REST API Guide: Workloads, Rulesets, Labels, Label Groups, Services, IP Lists, and Pairing Profiles. This may change with future versions of the PCE as new features are added. It's always wise to check both the Release Notes and the current API REST Guide for each PCE version to see what has changed.

To do an async API call, use the same URL as a standard synchronous query, with the addition of a header of "Prefer: respond-async". In this chapter, we'll write a function to make standard synchronous queries (making queries like in the previous postings simpler), then build on top of that to also be able to do asynchronous queries.

In this example, let's assume that there are more than 500 Labels, and that we want to retrieve the whole set, not just the first 500. The first question is, how do we know if there are more than 500 results? One strategy is to simply use asynchronous calls all the time. While this does technically work, it makes your code slow, as asynchronous queries are considerably slower (by at least an order of magnitude, sometimes more) than standard synchronous queries. So you need to be adaptable.

Unfortunately, there's no flag that comes back with the results from a GET query that tells you there is more data available than what was returned. One strategy is whenever one of the seven queries above returns 500 or more results, to set a flag in your code and switch to doing async queries for that category of objects. If the number of results from an asynchronous query drops below some value, say 475 results, then change the flag back to doing synchronous queries for that particular class of objects. It's more bookkeeping overhead to keep track of, but it keeps your code faster than simply doing the safe thing and relying on either type of query for everything.

The standard query to return all Labels (assuming the same first few lines of code we've been using in previous postings) is:

```
response = RestClient::Request.new(
  :url => "https://"+pce+":"+port+"/api/v1"+org_href+"/labels",
  :method => :get,
  :user => login,
  :password => passwd,
  :headers => {:accept => :json,
               :content_type => :json}).execute()


Turning this into an async query requires the addition of one additional
header, "Prefer: respond-async":


response = RestClient::Request.new(
  :url => "https://"+pce+":"+port+"/api/v1"+org_href+"/labels",
  :method => :get,
  :user => login,
  :password => passwd,
  :headers => {"Prefer" => "respond-async",
               :accept => :json,
               :content_type => :json}).execute()
```

When making an asynchronous query, there are additional steps involved in getting your results. The process is detailed below, but the concept is simple. You first make a query using the same format as always, but with the additional header "Prefer: respond-async" telling the API you'd like this to be an asynchronous query. The API then returns a URI that provides the status of your query (among other things). You then periodically do a GET (using a standard synchronous query) on that URI until it either succeeds or times out (fails). Once it succeeds (which can take multiple seconds for a query that returns thousands of results), the API call then returns yet another URI, which you then do a synchronous GET on to get your final results. This allows the query to run to completion at a lower priority, keeping the load on the Policy Compute Engine lower.

The initial asynchronous API call returns three fields we care about, "Status", "Location", and "Retry-After". The first tells us whether or not the request was successfully accepted (the response will be "202 Accepted" if it is).

The second is the URI to poll to find out if the query is complete. The third is a suggested amount of time (in seconds) to wait before making the first follow-up query to check for your data. The catch is, rather than being returned as JSON in the body of the reply, these three variables are all returned as part of the header of the

result. Meaning it's just a bit harder to get at the data and will require a new trick. If we look at the headers, we see the following:

```
[2] pry(main)> response.headers
=> {:transfer_encoding=>"chunked",
 :status=>"202 Accepted",
 :location=>"/orgs/9/jobs/603e893e-8a20-45f8-986e-5d65ed9ed946",
 :retry_after=>"5",
 :x_request_id=>"60875e3d-e02d-4079-b4f9-7f86a37467a8",
 :date=>"Tue, 16 May 2017 19:09:02 GMT",
 :content_encoding=>"gzip",
 :cache_control=>"no-store"}
[3] pry(main)> response.headers[:location]
=> "/orgs/9/jobs/603e893e-8a20-45f8-986e-5d65ed9ed946"
[4] pry(main)> response.headers[:status]
=> "202 Accepted"
[5] pry(main)> response.headers[:retry_after]
=> "5"
[6] pry(main)>
```

The :status value tells us that our request was successful. The :location value tells us what URL to poll to let us know when we're done, and the :retry_after value tells us to wait an estimated five seconds for a result (you can certainly check prior to five seconds, as the returned value is nothing more than an estimate, so be prepared to check more than once in case the estimate is optimistic).

Let's write some code to wait the suggested amount of time, then poll the specified URL until it either reads "done" or "failed". In our example code, we assume that all queries eventually succeed. In production systems, it's definitely best practice to check the return code and handle failure appropriately.

To wait the suggested amount of time, first, we convert the suggested wait time from a string to an integer, then sleep for that number of seconds. We then loop until :status indicates the query is done, sleeping for one second between each query. Finally, we do a GET on the URI where the Policy Compute Engine has stored our final result:

```
sleep(response.headers[:retry_after].to_i)

status=""
location=response.headers[:location]
while (status!="done" and status!="failed")
  response = JSON.parse(RestClient::Request.new(
                          :url
=>"https://"+pce+":"+port+"/api/v1/"+location,
                          :method => :get,
                          :user => login,
                          :password => passwd,
                          :headers => {:accept => :json,
                                       :content_type => :json}).execute())
  status=response['status']
  puts status
  sleep 1 unless (status=="done" or status=="failed")
end

url=response["result"]["href"]
```

At this point, the query has either failed, or has produced a result. Let's go fetch the contents of the variable url and display the resulting Labels (this code should look familiar):

```
response = JSON.parse(RestClient::Request.new(
                        :url => "https://"+pce+":"+port+"/api/v1"+url,
                        :method => :get,
                        :user => login,
                        :password => passwd,
                        :headers => {:accept => :json,
                                     :content_type => :json}).execute())
response.each do |resp|
```

```
  puts "href: #{resp['href']}"
  puts "type: #{resp['key']}"
  puts "name: #{resp['value']}"
  puts ""
end
```

Here's what running the code looks like:

```
jfrancis@hoss ~/api $ ./api_lesson_5.rb
pending
pending
pending
pending
running
running
running
done

href: /orgs/9/labels/1084
type: role
name: Nginx_LB

href: /orgs/9/labels/1087
type: env
name: Development

jfrancis@hoss ~/api $
```

You'll notice that the status is "pending" for the first four seconds. This indicates that the Policy Compute Engine is busy doing something of higher priority than our query. Our query then starts running, and runs for three seconds before it's done, and our results are available to fetch.

That's it! Now you can do asynchronous queries. In the next chapter, we'll package these up for more convenient use.

The source code for this chapter can be found in Chapter 6 Source Code.

Chapter 7: Cleaning Up Our Code

Functions and objects are fundamental concepts of programming. Let's build a few functions to clean up our code to make writing future code easier.

Before we go any further, let's stop and clean things up a bit. We've learned about pushing things into the API to create objects in the PCE, and we've learned two ways to pull things out (synchronously and asynchronously). But our code is getting a bit verbose. Let's stop and use this chapter to create an object (to hold PCE credentials) and two functions (one to make standard synchronous API calls, and a second to make asynchronous GET API calls). This will be a little bit of work, but will collapse future API calls down to a single line of code. This will make our future code both easier to read and less likely to contain errors.

First thing, let's create an object to hold the credentials for a Policy Compute Engine. This object should contain the URL of the PCE, the port to connect to, your login and password, and your org_id. In addition, let's create two object methods. The first, given the resource you want to access, will return a full URL to connect to, including the org_href. The second will be the same, but lacking the org_href. Why? Because some calls require the org_href, and others will use the org_href already embedded in the HREF we supply to the call (the Illumio REST API Guide will tell you when to use each).

Here's a nice Creds object to play with:

```
class Creds
  attr_accessor :login, :passwd, :pce, :port, :org

  def initialize(login, passwd, pce, org, port=443)
    @login = login
    @passwd = passwd
    @pce = pce
    @port = port.to_s
    @org = org
  end

  def url_with_api(rest)
```

```
    if(rest[0] == "/")
      rest = rest[1..-1]
    end
    return("https://#{@pce}:#{@port}/api/v1/#{rest}")
  end

  def url_with_org(rest)
    if(rest[0] == "/")
      rest = rest[1..-1]
    end
    return("https://#{@pce}:#{@port}/api/v1#{@org}/#{rest}")
  end
end
```

Assuming you keep the first few lines of your code the same (where you specify login, passwd, etc.), you can create a new credentials object like this, then query the object for things such as the passwd and the proper URL to fetch Labels:

```
[1] pry(main)> creds=Creds.new(login,passwd,pce,org_href,port)
=> #<Creds:0x007fee3b914c78
 @login="api_1ea0799f8bd486c8e",
 @org="9",

@passwd="6c1dbf2971aa1ff1a45b142fa86fc6040eeb9bbbe1957d4a977225bdf02d400d"
,
 @pce=" illumio.company.com ",
 @port="8443">
[2] pry(main)> creds.login
=> "api_183791c52058d3571"
[3] pry(main)> creds.url_with_org("/labels")
=> "https://illumio.company.com:8443/api/v1/orgs/31/labels"
[4] pry(main)>
```

Now instead of a bunch of nasty global variables, we can pass a Creds object to the functions we write to talk to the API, and those functions can get everything they need from the objects. Cleaner, safer, and less prone to error. Note that there's code in there to make sure there's no double "//" when specifying the resource (i.e., you can specify either "labels" or "/labels", and the method will do the right thing).

The first function, doing a synchronous API call, is pretty straightforward. It's exactly what we've been doing all along, but we're going to package it up in a nice neat function. We already know we need to pass the Creds object, but what else? Happily, the answer is "not much." We'll need to specify the type of API call this is (:get, :put, :post, or :delete). We'll call that variable "type". Next, we need to know what resource we want to manipulate in the API. For example, to fetch all of the Labels, the resource is "/labels". We'll call this "resource". We need to know if the final URL should include the org_href or not, so we'll have a Boolean value of "nil" or "true" for our variable "org" ("true" means include it, and "nil" means don't). Finally, there's an optional payload. If there is no payload, you can supply a "nil" or simply don't specify the final parameter.

Here's a simple sync API call function:

```
def sync_api(creds,type,resource,org,payload = nil)
  api = ""
  if (org)
    api = creds.url_with_org(resource)
  else
    api = creds.url_with_api(resource)
  end

  response = RestClient::Request.new(
    :url => api,
    :method => type,
    :payload => payload,
    :user => creds.login,
    :password => creds.passwd,
    :verify_ssl => false,
    :headers => {:accept => :json,
                 :content_type => :json}).execute()

  if (response.to_s() != "")
    return(JSON.parse(response))
  else
    return "ok"
  end
end
```

Now we can make calls to the API in a much easier manner. For example, to fetch the Labels, you simply call:

```
sync_api(creds,:get,"/labels",true)
```

In return, you get a parsed JSON data structure with the data you asked for.

The asynchronous API call is a bit more complex. It's what we did before when we learned about asynchronous API calls, but a bit shorter, because we're at least partially able to take advantage of our new sync_api() call to shorten and simplify the code:

```
def async_api(creds,type,resource,org,payload = nil)
  api = ""
  if (org)
    api = creds.url_with_org(resource)
  else
    api = creds.url_with_api(resource)
  end

  response = RestClient::Request.new(
    :url => api,
    :method => :get,
    :user => creds.login,
    :password => creds.passwd,
    :headers => {"Prefer" => "respond-async",
                 :accept => :json,
                 :content_type => :json}).execute()

  sleep(response.headers[:retry_after].to_i)
```

```
  monitor_url = response.headers[:location]
  status = ""
  while (status != "done" and status != "failed")
    response = sync_api(creds,:get,monitor_url,nil)
    status = response['status']
    sleep 1 unless (status == "done" or status == "failed")
  end

  return (sync_api(creds,:get,response["result"]["href"],nil))
end
```

We still need to make the initial API call in this new function the long way because we have to do two things different: we have to specify an extra header to request an async reply, and we have to parse the returned headers to find out where to check for status (normally, we discard the returned headers). While we could certainly do this with extra parameters to the sync_api() call, in the interest of keeping the rest of our code a bit cleaner, we'll take the hit and do one call the hard way here.

While this was a bit of a diversion from learning the API itself, we've now got some useful functions. For example, you can now get a list of Labels in two ways:

```
labels = sync_api(creds,:get,"labels",true)
```

or

```
labels = async_api(creds,:get,"labels",true)
```

The first does a synchronous API call, and the second does an asynchronous call.

In the next chapter, we're going to use our new functions to do something a little more complex than we've done before. We're going to build some Unmanaged Workloads. This is more complex because it will involve multiple API calls, as Unmanaged Workloads need to have Labels. The JSON that's supplied for Unmanaged Workloads is also a bit more complex, so we'll explore ways to simplify things.

The source code for this chapter can be found in Chapter 7 Source Code.

Chapter 8: Creating Unmanaged Workloads

Though it was small and simple, what we built in Chapter 5: From Spreadsheets to Labels was our first real tool. In this chapter, we're going to build our first real, full application that leverages the API. Creating Unmanaged Workloads is a major task with most Illumio deployments. In a typical deployment, there are often twice as many Unmanaged Workloads as there are Managed. This translates to potentially thousands of mouse clicks. While we do already have a full-featured command line tool that uses the API to solve this problem (as well as many others) called the Go Absorption Toolkit (available for free download from the Illumio Support Portal), the purpose of this exercise is to learn by doing.

Our goal for this chapter is to take an Excel spreadsheet filled with Workload data and turn it into properly Labeled Unmanaged Workloads in the PCE. For each Unmanaged Workload (which I'll refer to as a "UMW" from here on out to save space), we'll need a name, a hostname (which can optionally be the same as the name, or empty), an IP address, and zero to four Labels (for Role, Application, Environment, and Location). As with our earlier Chapter 5 application, we'll skip putting Labels at the top of columns to simplify our code, but the columns are Name, Hostname, IP, Role, App, Env, Loc:

	A	B	C	D	E	F	G
	Name Box	foo.company.com	10.0.0.1	web	bazinga	pre-production	earth
2	bar	bar.company.com	10.0.0.2	middleware	bazinga	pre-production	earth
3	baz	baz.company.com	10.0.0.3	profit	bazinga	pre-production	earth
4	qux	qux.company.com	10.0.0.4	database	bazinga	pre-production	earth
5	quux	quux.company.com	10.0.0.5	database	bazinga	pre-production	earth

For any (or all) Label(s) you want empty, simply leave the space blank. Remember to export your Excel spreadsheet to CSV, but do not select the UTF-8 variant of CSV, or your file will be unreadable by this script.

We're going to try to be smart about this. For any Label specified that does not already exist, we'll create and use a new one. If a Label already exists in the PCE, however, we'll use the old one. Note that capitalization is critical; "Web" is not the same as "web". As you can see from the spreadsheet above, we're going to create five machines with various roles, but all parts of the Bazinga Application are in Pre-Production on Earth.

We're going to use our Creds object and two functions from Chapter 7 for accessing the API to build on top of. In addition, we'll need a few new functions before we start. The first thing we'll need is a list of all of the Labels that already exist in the PCE. That way, we'll know what Labels we can re-use, and what Labels we'll need to create from scratch. We'll need a function to grab and store all of the existing Labels, and we'll also need a function to create new Labels. First, pull out your code from last time, save it into a new file called api_lesson_7.rb, and keep everything down to (and including) the line where you create the Creds object.

First, let's write some code to pull out the existing Labels. We're going to store the Label data in a hash of hashes, called "labels". The hash inside of "labels" will contain four hashes, called "role", "app", "env", and "loc". These, in turn, will contain a key value of the name of the Label, and a value of the HREF of the Label. This makes it easy to find a Label's HREF (which we'll need to construct UMWs) with nothing but a type and name:

```
[3] pry(main)> labels['env']['Production']
=> "/orgs/31/labels/1302"
[4] pry(main)>
```

Here's a function that pulls the Label data from the API, and returns a hash of hashes which we can assign to "labels":

```
def get_all_labels(creds)
  response = sync_api(creds,:get,"/labels",true)
  if (response.length == 500)
    response = async_api(creds,:get,"/labels",true)
  end

  labels = Hash.new()
  labels['role'] = Hash.new()
  labels['app'] = Hash.new()
  labels['env'] = Hash.new()
  labels['loc'] = Hash.new()

  response.each do |resp|
    if (resp['key'] == "role")
```

```
      labels['role'][resp['value']] = resp['href']
    end

    if (resp['key'] == "app")
      labels['app'][resp['value']] = resp['href']
    end

    if (resp['key'] == "env")
      labels['env'][resp['value']] = resp['href']
    end

    if (resp['key'] == "loc")
      labels['loc'][resp['value']] = resp['href']
    end
  end

  return(labels)
end
```

As you can see, the code first constructs the hash of hashes, then iterates through the data returned by the API. For each returned Label, it determines which of the four types the Label is, then stores it in the appropriate hash. At the end, the entire hash of hashes is returned. While there are more efficient methods of doing this, this makes for clean code that's easy to read, and this process will only ever be done once per run of the program, so memory utilization and garbage collection is the least of our worries. Note that we make a synchronous call to the API to get our data. If exactly 500 items are returned, we make the bet that there may very well be more than 500, and do a second, asynchronous call to fetch the full data set. You could simply change the code to always take the safe bet and do an async call, but it will run much more slowly if you have fewer than 500 Labels in your PCE.

We also need a function to create a new Label. This one is trivial:

```
def create_label(creds,type,name)
  return(
    JSON.parse(sync_api(creds,:post,"/labels",true,
                        {"key" => type, "value" =>
name}.to_json).to_json))
end
```

In this function, we simply call the API with the two necessary parameters (name and type), plus the proper API credentials.

Next, we need a function to create Unmanaged Workloads. The secret formula for the required JSON is documented in the Illumio REST API Guide, but the code below gives you a pretty good idea of what you'll need. This function will require credentials, a name, a hostname (can optionally be the same as name or blank), an IP address, and zero to four Labels, specified by their HREFs, in any order.

The code first checks to see what Labels, if any, you've supplied, then creates an array of hash values to include as part of the JSON we'll later feed to the API. Then we simply create a blob of JSON with all the right fields filled out and submit it to the API:

```
def create_umw(creds,name,hostname,ip,lab1=nil,lab2=nil,lab3=nil,lab4=nil)
  lab = Array.new()

  lab.push ({"href" => lab1}) if (lab1)
  lab.push ({"href" => lab2}) if (lab2)
  lab.push ({"href" => lab3}) if (lab3)
  lab.push ({"href" => lab4}) if (lab4)

  wl = {"name" => name,
        "hostname" => hostname,
        "public_ip" => ip,
        "interfaces" =>
        [{"name" => "eth0",
          "address" => ip,
          "cidr_block" => 32,
          "link_state" => "up"}],
        "online" => true,
        "labels" => lab }

  return(
    JSON.parse(
      sync_api(creds,:post,"/workloads",true,wl.to_json)))
end
```

That pretty much sums up the required new functions. Next, we'll read the data from the CSV file, and create that which needs creating. First step, though, is to use our shiny new function to pull the Label data and suck in all the Labels. Put this right below creating your Creds object:

```
creds = Creds.new(login,passwd,pce,org_href,port)
labels = get_all_labels(creds)
```

Now we've got Label data to work with, and credentials to talk to the PCE.

This next bit of code gets a little bit hairy, so let's talk through the process before we look at too much code. First thing we do is open the file and read it line by line, first splitting each line into it's constituent fields using the CSV gem, and finally assigning each line's values to a set of variables that are easier to remember than referring to the fields by names like "stuff[3]". As with our earlier code that reads CSV files, we have hard-coded the name to "workloads.csv", which must be in the current directory. It's a useful exercise for the reader to make this more flexible:

```
File.open("workloads.csv").each do |line|
  CSV.parse(line) do |stuff|
    name = stuff[0]
    hostname = stuff[1]
    ip = stuff[2]
    role = stuff[3]
    app = stuff[4]
    env = stuff[5]
    loc = stuff[6]
```

Now we head into the loop and check each Label to see if it already exists. If it does, the corresponding variable (role, for the first case) is set to the HREF for the specified role Label. On the other hand, if this role does not already exist, we use our new Label creation function and create it, then store the HREF of the brand new Label in the variable "role", as well as adding it to the hash of known Labels (for use next time this Label is required). Finally, if there is no role Label, "role" is set to nil so it will be skipped later on down the code:

```
if (role != nil)
  if (labels['role'][role])
    puts "Role #{role} already exists."
    role = labels['role'][role]
  else
    puts "Creating role #{role}..."
    href = create_label(creds,"role",role)['href']
    labels['role'][role] = href
    role = href
  end
else
  puts "Role not specified."
  role = nil
end
```

Now we do this three more times in line, for each of app, env, and loc. At the end of these four tests, we have the role, app, env, and loc variables set to the HREF of their respective Labels (or to nil).

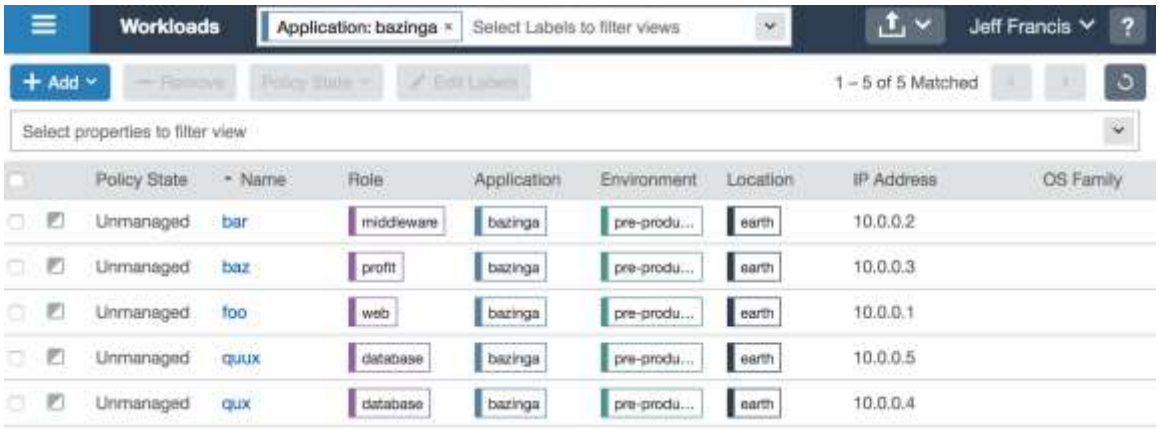
Last, but certainly not least, we create the Unmanaged Workload using all of the data we've accumulated:

```
wl = create_umw(creds,name,hostname,ip,
                role,app,env,loc);
```

And that's it. We can now create Unmanaged Workloads by the thousands. Here's a sample run of the code:

```
jfrancis@hoss ~/src $ ./api_lesson_7.rb
Creating role web...
Creating app bazinga...
Creating env pre-production...
Creating loc earth...
Creating unmanaged workload foo...

Creating role middleware...
App bazinga already exists.
Env pre-production already exists.
Loc earth already exists.
Creating unmanaged workload bar...

Creating role profit...
App bazinga already exists.
Env pre-production already exists.
Loc earth already exists.
Creating unmanaged workload baz...

Creating role database...
App bazinga already exists.
Env pre-production already exists.
Loc earth already exists.
Creating unmanaged workload qux...

Role database already exists.
App bazinga already exists.
Env pre-production already exists.
Loc earth already exists.
Creating unmanaged workload quux...

jfrancis@hoss ~/src $
```

You'll see that the code is now "smart" enough to re-use pre-existing Labels, and create new ones when needed. If you look in your PCE after this code was run, you'll see something like this:



The source code for this chapter can be found in Chapter 8 Source Code.

Chapter 9: Quarantining Workloads

In this chapter, we're going to talk about how to quarantine a Workload. When anomalous behavior is noted in your network (whether by wild new red lines in Illumination, reports by a user, or information from a malware detection product), it's generally considered to be a best practice to isolate that machine from the rest of your data center to prevent the potential spread of infection. In the Olden Days of Networking, that usually meant removing power to the machine, pulling it's LAN cable, and/or disabling the switch port the machine was plugged into. In the new Enlightened Age of Illumio, you have a new option: you can quarantine the Workload, and isolate it from the rest of your network using firewalls, but still allow carefully restricted remote access for cleaning, repairing, or debugging the affected machine.

In order to quarantine a Workload, you need an Application Group to move the Workload into (i.e., change it's Labels to match those of a Quarantine group). This Application Group (the Quarantine group) should have a corresponding Ruleset that, as an example, allows zero traffic in or out, other than incoming connections (perhaps from a small number of carefully specified workstations) on a few limited ports, perhaps Windows Remote Desktop and ssh.

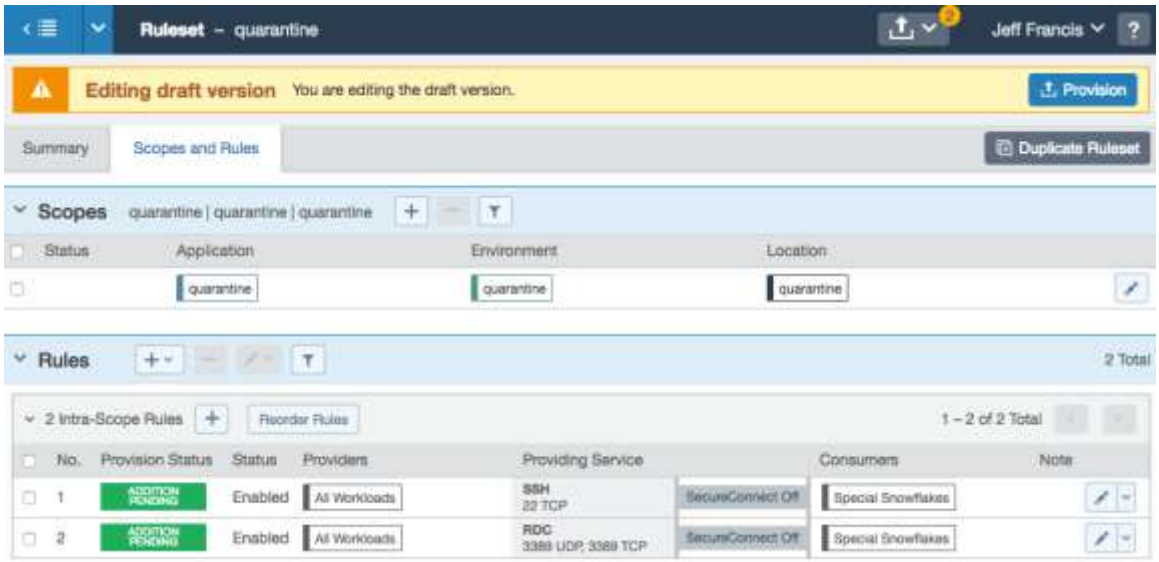
Changing the Labels on an infected machine immediately causes that machine to inherit a new policy, disallowing outgoing communications, and only listening to ports necessary to debug and analyze the problem. Even more importantly, it immediately updates the policy on every other system in your data center that runs the Illumio VEN to block all attempts to communicate with the infected system. It's almost like pulling the LAN cable, but can be done with a few API calls to the Illumio Policy Compute Engine. And that is our task for today.

Your first task is to create the Labels you'd like to use for your quarantine zone. There are any number of ways to go about this, but for this example, we're going to create Application, Environment, and Location Labels all with the same name, "quarantine". This task only needs to be performed once for the life of the PCE.

This will result in an application bubble for our quarantined Workloads, though it will not show on the screen in Illumination until there is at least one Workload with those Labels. The code itself doesn't do this, as it's assumed that you'll want to also

create a Ruleset for the quarantine zone with a few Rules allowing specific system management (rdp/ssh) traffic inbound.

You'll want to create this Ruleset (which I called "quarantine" in my example) with the scope of your quarantine zone (in this example, "quarantine" "quarantine" "quarantine"). In my example, I added two Rules. The first will allow traffic (preferably from a small number of systems used to diagnose bad machines) to talk to the quarantined Workloads on TCP 3389 and UDP 3389. This will allow Remote Desktop, assuming the machines are running Windows. The second should allow TCP 22 to allow for logging into machines running Linux. Here's what my example looks like in the PCE web console:



From here, we'll proceed with the code we wrote in the last installment for creating the Unmanaged Workloads. We'll be re-using some of that code, and adding a few new things.

Just like last time, we'll need a list of all existing Labels (so that we can find the quarantine Labels), but we'll also now need a list of all Workloads. We'll need to scan through all of the Workload data in order to find which of those Workloads has the IP address that was specified as the machine to be quarantined. So let's start with a new function to fetch all of the Workloads, then store them in a hash indexed by HREF:

```
def get_all_workloads(creds)
  workloads = Hash.new()

  response = sync_api(creds,:get,"/workloads",true)
  if (response.length == 500)
    response = async_api(creds,:get,"/workloads",true)
  end

  response.each do |wl|
    workloads[wl['href']] = wl
  end

  return(workloads)
end
```

As with our function to fetch Labels, we automatically switch from a sync API call to an async call if 500 results are returned, making the assumption that if there are 500, there may be more. We then iterate through the list, adding each Workload to a hash that will be returned to the user, using the HREF of the Workload as the key, and the parsed JSON data as the value. This function is called very much like the corresponding Label function:

```
workloads = get_all_workloads(creds)
```

Now we need a function to match a specified IP address with any and all potential IP addresses in a Workload. While we often think of a system as having a single IP address, this is often not the case. It's quite common for a server (or workload) to have more than one network interface, each with a different address, as well as having virtual network interfaces on the physical interfaces. We need to check each of these for a match. Let's start with the JSON data returned for a single network device, and see where and how we find this data:

```
[2] pry(main)> workloads["/orgs/31/workloads/1a81d99f-1196-4377-8ecb-
6095d7fed168"]
=> {"href"=>"/orgs/31/workloads/1a81d99f-1196-4377-8ecb-6095d7fed168",
 "deleted"=>false,
 "name"=>nil,
 "description"=>nil,
 "hostname"=>"hrm-web-prod3",
 "service_principal_name"=>nil,
 "public_ip"=>"54.153.21.11",
 "external_data_set"=>nil,
 "external_data_reference"=>nil,
 "interfaces"=>
  [{"name"=>"eth0.public",
    "address"=>"54.153.21.11",
    "cidr_block"=>32,
    "default_gateway_address"=>nil,
    "link_state"=>"up",
    "network_id"=>46,
    "network_detection_mode"=>"manual",
    "friendly_name"=>nil},
   {"name"=>"eth0",
    "address"=>"10.10.100.5",
    "cidr_block"=>nil,
    "default_gateway_address"=>"10.0.1.255",
    "link_state"=>"up",
    "network_id"=>46,
    "network_detection_mode"=>"single_private_brn",
    "friendly_name"=>"Friendly eth0"}],
 "ignored_interface_names"=>[],
 "service_provider"=>"amazonaws.com",
 "data_center"=>"us-west-2.amazonaws.com",
 "data_center_zone"=>"us-west-2c",
 "os_id"=>"ubuntu-x86_64-precise",
 "os_detail"=>"agent simulator 2017-01-08 05:46:38 +0000",
 "online"=>true,
 "labels"=>[{"href"=>"/orgs/31/labels/1296"},
{"href"=>"/orgs/31/labels/1342"}, {"href"=>"/orgs/31/labels/1302"},
{"href"=>"/orgs/31/labels/1339"}],
 "services"=>{},
```

```
 "agent"=>
  {"config"=>{"log_traffic"=>false, "visibility_level"=>"flow_summary",
"mode"=>"enforced"},
   "href"=>"/orgs/31/agents/28618",
   "status"=>
    {"uid"=>"0289091c-9828-4eae-aa4e-7777c58780b5",
     "last_heartbeat_on"=>"2017-05-19 16:31:24.657003",
     "instance_id"=>"i-01234568",
     "managed_since"=>"2017-03-30T03:43:18.161418Z",
     "fw_config_current"=>true,
     "firewall_rule_count"=>12345,
     "security_policy_refresh_at"=>"2015-04-27T16:13:00Z",
     "uptime_seconds"=>"6",
     "status"=>"active",
     "agent_version"=>"1.12.0-20140729143905",
     "agent_health_errors"=>{"errors"=>[], "warnings"=>[]},
     "agent_health"=>[]}},
 "created_at"=>"2017-03-30T03:43:18.157839Z",
 "created_by"=>{"href"=>"/orgs/31/agents/28618"},
 "updated_at"=>"2017-05-11T23:25:01.227857Z",
 "updated_by"=>{"href"=>"/orgs/31/agents/28618"}}
[3] pry(main)>
```

That's a lot of data, but you'll notice there are two places where IP addresses are specified. The first is in a key called "public_ip". This is the network address that's used by the device to talk to the Policy Compute Engine, and the address this particular Workload is known by. This address is certainly worth testing. Here's how we grab it:

```
[3] pry(main)> workloads["/orgs/31/workloads/1a81d99f-1196-4377-8ecb-
6095d7fed168"]['public_ip']
=> "54.153.21.11"
[4] pry(main)>
```

However, it's not the only address of the system. If you look further down the returned data, there's a key called "interfaces" which contains an array with one entry per physical or virtual interface on the device. Within this hash is a key called

"address". Here's the hash, and how we grab the IP of the first interface (obviously, we'll iterate through all interfaces in our "real" code):

```
[4] pry(main)> workloads["/orgs/31/workloads/1a81d99f-1196-4377-8ecb-
6095d7fed168"]['interfaces']
=> [{"name"=>"eth0.public",
  "address"=>"54.153.21.11",
  "cidr_block"=>32,
  "default_gateway_address"=>nil,
  "link_state"=>"up",
  "network_id"=>46,
  "network_detection_mode"=>"manual",
  "friendly_name"=>nil},
 {"name"=>"eth0",
  "address"=>"10.10.100.5",
  "cidr_block"=>nil,
  "default_gateway_address"=>"10.0.1.255",
  "link_state"=>"up",
  "network_id"=>46,
  "network_detection_mode"=>"single_private_brn",
  "friendly_name"=>"Friendly eth0"}]
[5] pry(main)> workloads["/orgs/31/workloads/1a81d99f-1196-4377-8ecb-
6095d7fed168"]['interfaces'][0]
=> {"name"=>"eth0.public",
 "address"=>"54.153.21.11",
 "cidr_block"=>32,
 "default_gateway_address"=>nil,
 "link_state"=>"up",
 "network_id"=>46,
 "network_detection_mode"=>"manual",
 "friendly_name"=>nil}
[6] pry(main)> workloads["/orgs/31/workloads/1a81d99f-1196-4377-8ecb-
6095d7fed168"]['interfaces'][0]['address']
=> "54.153.21.11"
[7] pry(main)>
```

And there it is, the address of the first interface. One of the actual enumerated interfaces should be a duplicate of the Public IP, but it's best to check all of them, regardless, just so we don't miss anything.

Now that we know how to get at the IP addresses, let's write a function that, given a specific IP and a list of all Workloads, returns an array of Workload HREFs with an interface that matches the address. Why an array instead of a single HREF? Because mistakes happen, and it's not impossible to have two machines with the same IP address. This is usually a mistake, and not deliberate design, but either way, it's impossible to know which of multiple Workloads is the "right" one, so we simply return all of them.

We'll create an array to return, then iterate through each Workload. For each Workload, we'll check for a match with the Public IP, then iterate through all existing interfaces looking for matches. Any time a match is found, the HREF is pushed into the array. At the very end, we return a de-duplicated list of matching HREFs (de-duplicated, because it's likely that a match will be found in both the Public IP and list of interfaces for any given Workload):

```
def find_ip_in_workloads(ip,workloads)
  matches = Array.new()

  workloads.keys.each do |wl|
    if(workloads[wl]['public_ip'] == ip)
      matches.push(wl)
    end

    workloads[wl]['interfaces'].each do |iface|
      if(iface['address'] == ip)
        matches.push(wl)
      end
    end
  end

return(matches.uniq())
end
```

When this function is run, it works as expected, returning an array containing the HREF for a single workload:

```
 [7] pry(main)> find_ip_in_workloads("10.10.100.5",workloads)
=> ["/orgs/31/workloads/1a81d99f-1196-4377-8ecb-6095d7fed168"]
[8] pry(main)>
```

The code needs to know the Application, Environment, and Location Labels we're using for our Quarantine Zone. There's no particular need to call them "quarantine", though that's what we'll do here, just for simplicity:

```
qapp = "quarantine"
qenv = "quarantine"
qloc = "quarantine"
```

So now that all of the various pieces are in place, let's talk about what we're actually going to do with this script. First, we'll read in all of the Labels from the PCE. We search these Labels to find the HREFs of the Labels to be applied to a Workload being placed into quarantine. Next, we read in all of the Workloads from the PCE. We then search all Workloads for the specific IP address of the machine to be quarantined, and return that Workloads (or Workloads) as an array of HREFs, assuming the Workload already exists. Finally, we iterate over the list of Workload HREFs (which will normally be a single Workload), and apply the new quarantine Labels to it. Assuming the Workload doesn't already exist, we'll create it as an Unmanaged Workload, with the appropriate quarantine Labels in place.

Let's get to it! Let's first grab the Labels and find the HREF(s) we need, and exit if we're unable to locate them:

```
labels = get_all_labels(creds)

qapp_href = labels['app'][qapp]
qenv_href = labels['env'][qenv]
qloc_href = labels['loc'][qloc]
```

```
unless(qapp_href and qenv_href and qloc_href)
  puts "Unable to locate specified quarantine labels."
  exit(1)
end
```

Now let's load all of the Workloads and find the "bad" one to be quarantined. In our example code, we specify the IP address of the miscreant machine at the top of our script. In the real world, you'd probably pass this another way, most likely as a command-line parameter:

```
infected_ip = "10.10.10.10"

workloads = get_all_workloads(creds)
infected = find_ip_in_workloads(infected_ip, workloads)
```

Before we start searching, let's go ahead and build the array we'll use to specify the Labels when we make that API call. Regardless of whether the Workload already exists, or we have to create it from scratch, we'll need this data:

```
lab = Array.new()

lab.push ({"href" => qrole_href})
lab.push ({"href" => qapp_href})
lab.push ({"href" => qenv_href})
lab.push ({"href" => qloc_href})
```

As we said earlier, the Workload may or may not already exist. If it exists, it will be returned as part of the list from find_ip_in_workloads(). Otherwise, that function will return an empty array. So let's see what we've got:

```
if(infected == [])
  # We'll need to create an Unmanaged Workload
else
  # Change the labels on the existing workload
end
```

Easy enough. Let's take the first case, first. We'll create an unmanaged workload with the proper labels and network interface:

```
  wl = {"name" => "infected_host_" + infected_ip,
        "hostname" => "bad",
        "public_ip" => infected_ip,
        "interfaces" => [
          {"name" => "eth0",
           "address" => infected_ip,
           "cidr_block" => 32,
           "link_state" => "up"} ],
        "online" => true,
        "labels" => lab }

  sync_api(creds,:post,"/workloads",true,wl.to_json)
```

On the other hand, if all we need to do is change the Labels, that's even easier. Note that in this example, we only change the Labels on the first Workload returned:

```
  wl = { "labels" => lab }
  sync_api(creds,:put,infected[0],nil,wl.to_json)
```

It's really that simple on the code side. Note that there are a few things to think about. First, as with the other examples in this book, there is no error handling here. More importantly, there is no logging. It would be wise to write details about each run of this script to a log, and note a time/date, as well as the Workload that was changed, along with the specific Labels prior to the change to "quarantine". Because eventually, you're going to want to put those Labels back the way they were, and having a record of the original Labels makes that easier.

A more advanced script might even store the "before" Labels along with a Workload HREF, and provide a command line interface where the script could also be used to "un-quarantine" a Workload, given the IP and/or HREF. It would probably also be wise to iterate through all of the Workloads returned by find_ip_in_workloads(), as it's possible there's more than one. This code only modifies the first returned result in order to keep the code simple and readable. Also note that in PCE versions prior to 16.09, when changes are made only to Unmanaged Workloads that do not change any normal VENs, changes are not pushed until the next event that changes VEN policy. Meaning that if you're running one of the affected versions of the PCE, your network is not actually protected immediately. The solution is to make a gratuitous change to something, then committing that change (you'll see how to commit changes in Chapter 11).

But there's something more important here than the code. As the American Lt. Col. Carlos A. Keasler once famously noted, "Just because you can, doesn't mean you should." A great deal of thought needs to be put into this problem before you simply plug this code into some sort of malware-detection API. Very simple errors, in any one of a dozen places, can take down your entire data center. Every malware detection engine has, at least once in it's history, had a set of signatures pushed out that identified something benign as a threat. Imagine if your malware detection engine identified a new update of the NGINX binary as malware. Without safeguards, this script would happily iterate through every NGINX box in your network and completely disable it's network connectivity, likely completely crippling your ability to provide services to your customers. It's easy to imagine scenarios where this process could run amuck and cause massive outages. Think before you implement a script of this power. It might be wise to create a list of machines that can't be automatically quarantined. Or pass all detection along for human intervention before automatically doing a quarantine. In a 1962 Spider-Man comic book, Uncle Ben wisely observed that, "With great power comes great responsibility."

The source code for this chapter can be found in Chapter 9 Source Code.

Chapter 10: Building Pairing Profiles

This chapter is about creating Pairing Profiles, and creating Pairing Keys from those profiles. A Pairing Profile is a way to define the default options that are selected (such as Labels) when installing the VEN on a Workload. Installing a VEN with the Labels already set allows that machine (provided Rules have already been established in the Policy Compute Engine for the appropriate Labels) to boot up for the first time and immediately become part of your security ecosystem with no further actions required.

You'll definitely want to read through the Illumio documentation on Pairing Profiles before tackling this chapter to understand what we're doing. Trust me on this. The various parameters will be meaningless without understanding the background material in the documentation.

There are a fair number of options for creating a Pairing Profile; more than anything we've looked at in the past. To begin, pull in the code we've used in the past (all of the "require" statements, the credentials, and the functions we've written). Up at the top, let's add the variables we're going to use to create the Pairing Profile:

```
pprofile_name = "Pair Me"
pprofile_app = "Boondoggle"
pprofile_env = "Production"
pprofile_loc = "Addis Ababa"
```

Our goal is to create a Pairing Profile called "Pair Me" for Workloads in my Boondoggle application in my production environment in my Addis Ababa data center. Other options we'll want to set (we'll talk about these below) is that we want to lock (or prevent the end user from changing) the Labels for Application, Environment, and Location, but allow the user to change the Role as he wishes. We want to allow unlimited uses of the key we build from this profile, but only for 24 hours. We also want to lock the profile to force logging and to provide full flow data (so nobody can do anything sneaky without us seeing the data). We're going to make the assumption that you have fewer than 500 Labels in your PCE. If that's not the case, change the sync_api() call that fetches Labels into an async_api() call.

The Pairing Profile JSON is complex enough that we're going to build it piecemeal, rather than trying to generate a full blob of JSON in one shot. We'll start with a hash, add the various elements, then convert the hash to JSON and submit it. We'll then use the Pairing Profile to generate a Pairing Key. First, let's set some of the options and required fields:

```
pprofile = Hash.new()
pprofile["name"] = pprofile_name
pprofile["enabled"] = true
pprofile["mode"] = "illuminated"
pprofile["key_lifespan"] = 86400
pprofile["allowed_uses_per_key"] = "unlimited"
pprofile["role_label_lock"] = false
pprofile["app_label_lock"] = true
pprofile["env_label_lock"] = true
pprofile["loc_label_lock"] = true
pprofile["mode_lock"] = false
pprofile["log_traffic"] = true
pprofile["log_traffic_lock"] = true
pprofile["visibility_level"] = "flow_summary"
pprofile["visibility_level_lock"] = true
```

Most of these are pretty self-explanatory, though it may not be immediately apparent what all of the options are. The JSON schema files will be very valuable here, in addition to the Illumio REST API Guide. We'll skip some of the obvious fields, but a few are worth talking about. The first of these is "key_lifespan". A key can be generated that is good forever, or it's lifespan can be limited to a specific number of seconds. If you want it to be unlimited, use a value of "unlimited". Note the quotation marks; they're mandatory. If you want to limit key use to a specific amount of time, you specify the number of seconds. For example, a day has 86400 seconds. Note that you must not use quotation marks around the number of seconds. Only use the quotation marks if you're specifying "unlimited", or you'll get a 406 error. "allowed_uses_per_key" is similar. It's either "unlimited", or an integer value with no quotation marks.

"mode" can be "idle", "illuminated", "test", or "enforced", depending on what enforcement state you want the Workload to begin in. So long as you don't set "mode_lock" to true (no quotation marks!), you can always change it later. Finally,

"visibility level" can be "flow_summary", "flow_drops", or "flow_off", for full flow data, data only on dropped flows, or no flow data at all. This can also be changed later, unless "visibility_level_lock" is set to true.

The Pairing Profile may have anywhere from zero to four Labels pre-set, and these can be left as changeable by the user or locked in place for a given Profile/Key. We're going to add three Labels, lock them (see above), and leave the Role Label to be changed by the user, either at installation time, or later. We add the four Labels to an array, then add the array to the Pairing Profile hash:

```
lab = Array.new()
lab.push({"href"=>labels["app"][pprofile_app]})
lab.push({"href"=>labels["loc"][pprofile_loc]})
lab.push({"href"=>labels["env"][pprofile_env]})
pprofile["labels"] = lab
```

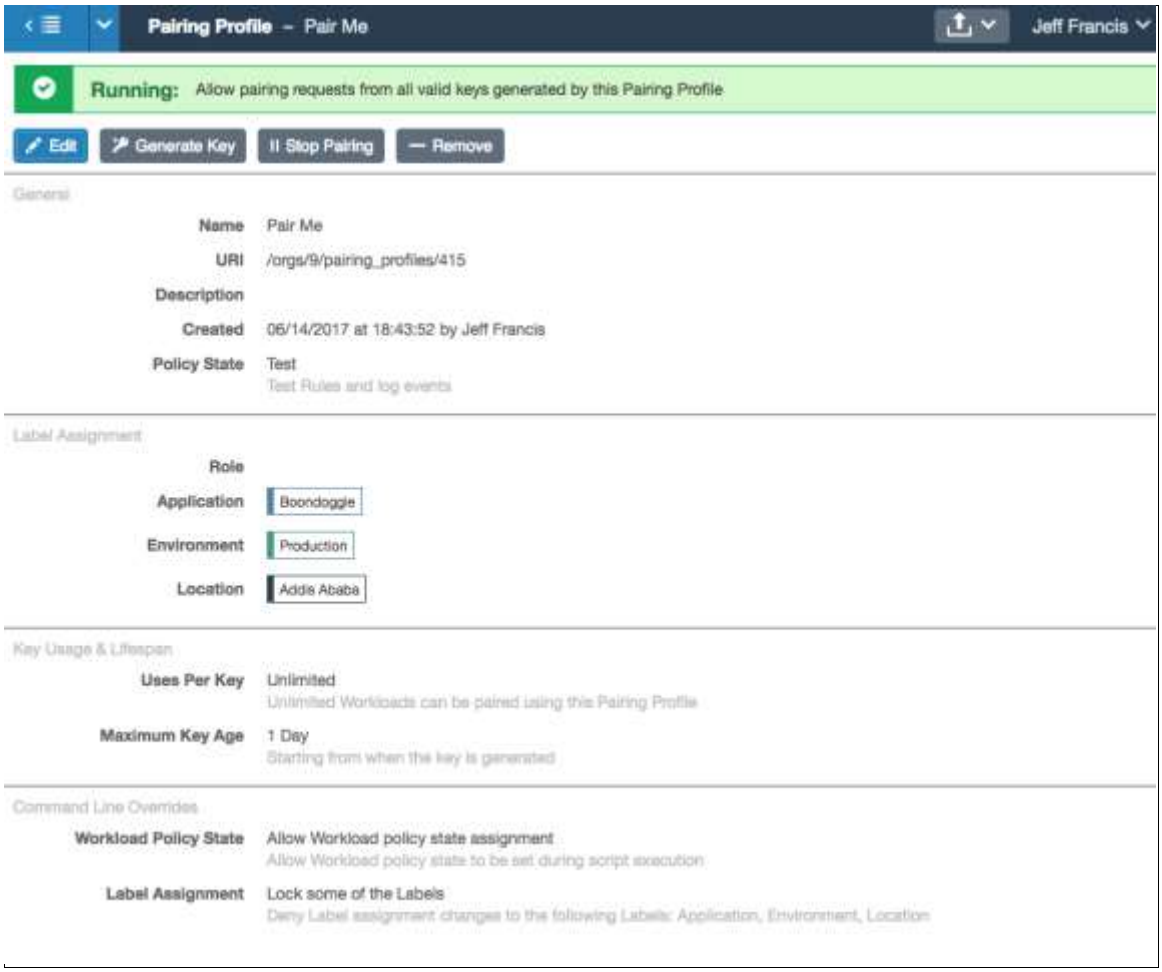
Let's have a look at the hash we've build before we convert it to JSON and push it to the API:

```
{"name"=>"Pair Me",
 "enabled"=>true,
 "mode"=>"illuminated",
 "key_lifespan"=>86400,
 "allowed_uses_per_key"=>"unlimited",
 "role_label_lock"=>false,
 "app_label_lock"=>true,
 "env_label_lock"=>true,
 "loc_label_lock"=>true,
 "mode_lock"=>false,
 "log_traffic"=>true,
 "log_traffic_lock"=>true,
 "visibility_level"=>"flow_summary",
 "visibility_level_lock"=>true,
 "labels"=>[{"href"=>"/orgs/9/labels/1152"},
{"href"=>"/orgs/9/labels/1154"}, {"href"=>"/orgs/9/labels/1153"}]}
```

Looks good! Let's push it to the API, and save the HREF that's returned (we'll need the HREF to generate the pairing key):

```
result = sync_api(creds, :post, "/pairing_profiles",
                  true, pprofile.to_json)["href"]
href = result["href"]
```

Assuming this was successful (yes, you should always check if you're writing real production code and not just learning), the variable HREF should be set to the HREF of the Pairing Profile you've just created. You can also see it in the PCE web console:



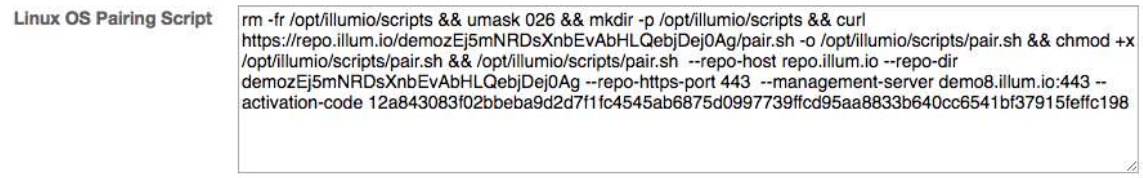
Now we can take the HREF and generate a Pairing Key with it. The key will be returned as the value for "activation_code":

```
result = sync_api(creds, :post, href + "/pairing_key",
                  false, {}.to_json)
activation_code = result["activation_code"]
```

It'll look something like this:

```
{"activation_code"=>"1bd76f13377c8c25c56ddcf96ba026a9aee011a25a38d1f4cf355
5b1a42047901b61cdd5f0dea2481"}
```

So now what do we do with the Pairing Key? Among other things, it can be plugged into the script that fetches and installs the VEN on the machines in your data center. You can get a template for this by clicking "Generate Key" in the Pairing Profile screen in the PCE web console. You'll get a sample script for both Windows and Linux that you can paste into the command line. Here's an example of a Linux installation script (the Pairing Key (or "Activation Code") is the last line in the script):



Obviously, generate your own script in the PCE web console, as this script won't work on your system. And remember to paste in your Pairing Key in place of the long key on the last line of the script. That's it. You can adapt this script into Chef, Ansible, Puppet, or Salt scripts, or just run it by hand on each machine you want to install the VEN on with this particular Pairing Profile.

The source code for this chapter can be found in Chapter 10 Source Code.

Chapter 11: Building Rulesets and Rules

This is the final chapter in the tutorial, and in it we will tackle the most difficult task that is commonly done via the API: automated policy creation. There are no new concepts here, and each of the individual tasks are all relatively straightforward, the resulting JSON is quite complex and every detail must be precisely perfect, or else the API will spit back a "406 Not Acceptable (RestClient::NotAcceptable)" error. Accuracy is key.

Like Chapter 10: Building Pairing Profiles, it's worth refreshing your memory of Rulesets and Rules using the Illumio documentation, lest this chapter sound like gibberish.

Let's dive in. There's going to be a lot of code compared to most previous chapters. This will be the longest chapter in the book by far, but the result will be code that can automatically generate policy for new applications as they're brought on line or migrated to Illumio. The specific task we're going to tackle is what we call HVA (or "High-Value Application") Ringfencing. It's the logical equivalent of sticking a group of machines on their own VLAN, then firewalling that VLAN from the rest of your network. We're going to build a policy to protect the refrigeration system at McMurdo Station in their Production environment. The only traffic we'll allow in is UDP port 42, which is the port our mythical refrigeration manufacturer uses for their remote-control application called "Fridgeomatic". We'll also turn on SecureConnect (IPsec encryption) for the traffic between the protected systems. Traffic will only be allowed from Fridge Command Central, a specific machine at address 10.246.0.1/32.

As always, let's start with the code we've developed to date (yes, you're correct, it would be great to package this up in a library or gem – consider that your homework assignment). Below the usual code at the top where we specify the required gems and the credentials to the PCE, there are some new variables defined for the Ruleset and Rules we're going to build:

```
ring_app = "Refrigeration"
ring_env = "Production"
ring_loc = "McMurdo"
ring_role = "thing"
ring_ruleset = "fridge"
ring_port = "42"
ring_proto = "17"
ring_service = "fridgeomatic"
ring_allowed_net = "10.246.0.1"
ring_allowed_netmask = "32"
ring_allowed_name = "fridge_command_central"
```

These are all of the things we're going to build in the PCE to build our new Ruleset. Most of these should be self-explanatory, but we'll talk about each variable as we use it in the code.

The first task we'll do in code is write a wrapper around our sync_api() and async_api() calls. Normally, you specifically call one or the other, depending on whether or not you expect <= 500 results back. We're going to define a simple call that tries sync_api(), and if 500 results are returned, automatically retries with async_api(). It's bad practice to use this as-is in production, but it makes our code cleaner and easier to read, so let's just go with it for now:

```
def call_api(creds, type, resource, org, payload = nil)
  response = sync_api(creds, type, resource, org, payload)
  if (response.length >= 500 and type == :get)
    response = async_api(creds, type, resource, org, payload)
  end
  return(response)
end
```

We do allow for the possibility of more than 500 items returned in a sync_api() get, as it's always possible the Illumio API may change in the future, and we don't want our code to break in strange ways.

We'll have lists of all IP Lists, Services, and Labels (see below), but we'll need a way to search these lists for specific entries. For Labels, that's simple. We created a hash structure in a previous lesson that organizes Labels so that they can be retrieved by

something like Label['role']['foo']. For the IP Lists and Services, we'll need a search function. The function below requires the name of the JSON variable you want to search for, the value of that variable to match, the array itself, and whether you want the whole match, or just the HREF(s). Note that the returned value is an array, not a single value. Why? Because it's possible to have IP Lists, Services, and Labels with duplicate names. So make sure you've got the one you really want, or really odd things can happen:

```
def find_var(var, value, stuff, href = false)
  if href
    return((stuff.select {|thing| thing[var] == value}).map {|thing|
thing["href"]})
else
    return(stuff.select {|thing| thing[var] == value})
  end
end
```

Here are two new functions. They work in the same way as functions we've created in the past, so should look familiar. The first creates a Service, and the second creates an IP List. Note that neither of these functions exploit the full capability of their respective API calls. Services can have many ports and IP Lists can have many hosts and/or subnets. Both functions, for simplicity, create one single entry each. Production code should almost certainly allow for multiple Services/addresses for efficiency. Note that the Illumio API wants IP protocols specified as an integer, not a name. These can be found on the Internet in RFC 1700, among other sources (do a quick Google search on "RFC1700"). The short version is that TCP is 6 and UDP is 17.

```
def create_simple_service(name, port, proto, creds)
  return(call_api(creds, :post, "/sec_policy/draft/services", true,
                  {"name" => name,
                   "service_ports" =>
                   [{"port" => port.to_i,
                     "protocol" => proto.to_i}]}.to_json))
end



def create_simple_iplist(name, address, subnet, creds)
```

```
  addr = address + "/" + subnet
  return(call_api(creds, :post, "/sec_policy/draft/ip_lists", true,
                  {"name" => name,
                   "ip_ranges" =>
                   [{"from_ip" => addr,
                     "exclusion" => false}]}.to_json))
end
```

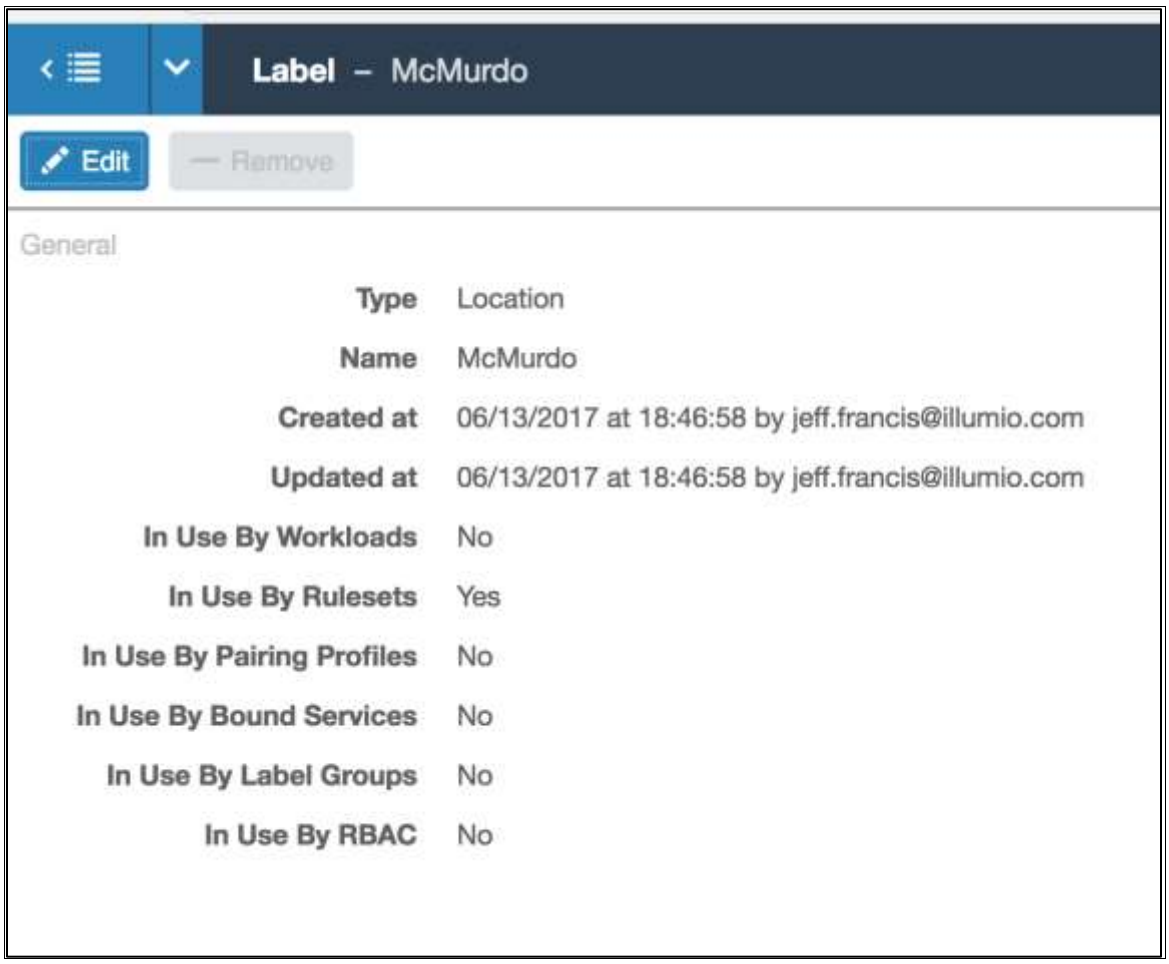
As in previous chapters, let's create a creds object to pass around for API calls. Then let's read in the Labels, IP Lists, and Services from the PCE:

```
creds = Creds.new(login, passwd, pce, org_href,port)
labels = get_all_labels(creds)
services = call_api(creds, :get, "/sec_policy/draft/services", true)
iplists = call_api(creds, :get, "/sec_policy/draft/ip_lists", true)
```

Ok, we're done with all of the setup. Time to get some real work done. First thing, let's create any of the Labels we'll need that don't already exist. Note that our function we created to make Labels doesn't check to see if a Label already exists, so we do it here ourselves before calling the function to create a new Label. Notice that we're also adding each Label as it's created to our global "labels" hash so we don't have to re-query the PCE for new Label data later:

```
unless labels["app"][ring_app]
  labels["app"][ring_app]=create_label("app", ring_app, creds)["href"]
end
unless labels["env"][ring_env]
  labels["env"][ring_env]=create_label("env", ring_env, creds)["href"]
end
unless labels["loc"][ring_loc]
  labels["loc"][ring_loc]=create_label("loc", ring_loc, creds)["href"]
end
```

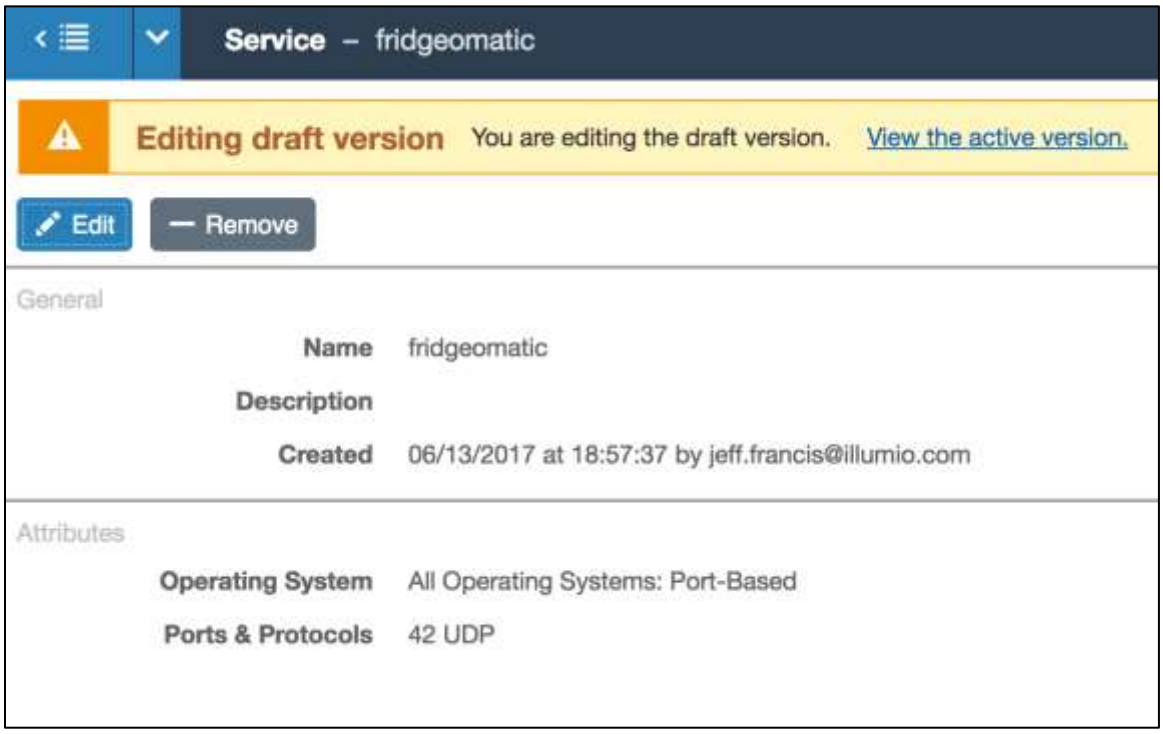
Once the Labels have been added, they'll show up in the PCE web console as something like this:



Now it's time to create our "Fridgeomatic" Service. Like our simple Label creation function, we need to check for pre-existing Services before we try to create one. Also note that, just like Labels, we save the returned data for efficiency:

```
if (find_var("name", ring_allowed_name, iplists, true).length == 0)
  iplists.push(create_simple_iplist(ring_allowed_name,
                                    ring_allowed_net,
                                    ring_allowed_netmask,
                                    creds))
end
```

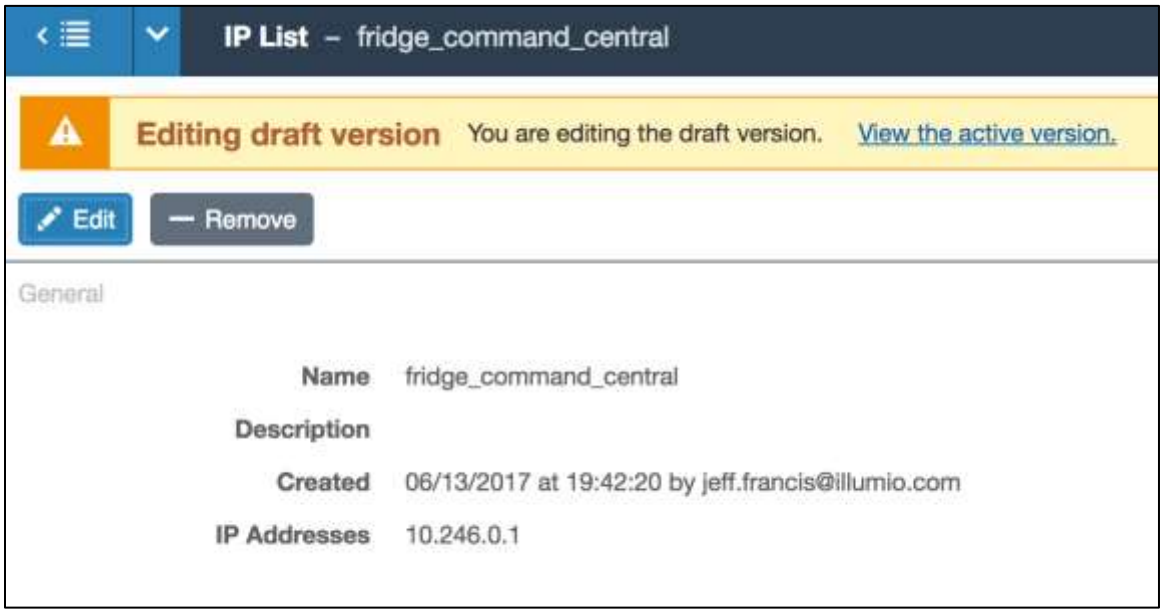
The Service will show up in the PCE web console like this:



We'll need to define the special subnet that's allowed to talk to the refrigeration system. As above, we first check, then save the returned data once it's built:

```
if (find_var("name", ring_allowed_name, iplists, true).length == 0)
  iplists.push(create_simple_iplist(ring_allowed_name,
                                    ring_allowed_net,
                                    ring_allowed_netmask,
                                    creds))
end
```

The IP List will look something like this in the PCE web console:



All of the required pieces are now built. It's time to build the actual Ruleset. The Ruleset has three basic pieces:

1. Some basic bits, like the name, and whether the Ruleset is enabled.
2. The Scope (or Scopes).
3. The actual Rules.

We'll build the Ruleset by stuffing everything into a hash named "ruleset". Once all the proper ingredients are added, we'll convert the hash to JSON and submit it to the API. First, let's create the hash, and put the first two small items into it:

```
ruleset = Hash.new()
ruleset["name"] = ring_ruleset
ruleset["enabled"] = true
```

In order to add the Scope, we'll create a "scope" array, add the various Scope pieces to it, then add the Scope to the Ruleset:

```
scope = Array.new()
scope.push({"label"=>{"href"=>labels["app"][ring_app]}})
scope.push({"label"=>{"href"=>labels["loc"][ring_loc]}})
scope.push({"label"=>{"href"=>labels["env"][ring_env]}})
ruleset["scopes"] = [scope]
```

We'll add Rules the same way. We'll create a Rules array to contain the individual Rules, then create a hash for each individual rule. After creating each rule, we'll copy the rule hash contents to the Rules array, and repeat for each rule. Once all individual Rules have been added to the Rules array, we'll add that array to the Ruleset hash. Here's the Rules array:

```
rules = Array.new()
```

Now let's build up the first rule and add it to the Rules. The first rule we'll create is the "All Workloads" can talk to "All Workloads" on "All Services". We'll also turn on SecureConnect for encryption. There are lots of fun things to take note of below. When you add a new rule, make sure it's enabled (unless you have reason not to). In this case, make sure sec_connect is true (false is off, or normal unencrypted traffic). The values for true and false must not be in quotation marks, or you'll get a 406 error (told you this was tricky). We also need to specify "unscoped_consumers". If set to true, this rule will be an Extra Scope rule, and if false, an Intra-Scope rule.

For providers and consumers, there are a wide array of things they can be set to in Rules, including Labels and IP Lists to name a few (the REST API Guide has complete guidance on this). For our purposes, we want to include "All Workloads" for both. In Illumio API parlance, "All Workloads" is represented as an array containing the hash {"actors" => "ams"} ("ams" is legacy parlance for "All Managed Servers"). Next, we want to specify that "All Services" are allowed. "All Services" is pre-populated in the PCE, so we search for it with our find_var() function, and blindly accept the first match returned (remember that you can have duplicate names, so enhancing this functionality is a good project for production versions of this code). Finally, we need to set "ub_service". "ub_service" is a bit of an oddball, and is likely to go away in future versions of the Illumio API. In the meantime, unless you're working with Bound Services, always set ub_service to the HREF for "All Workloads" (see the REST API Guide for more information).

We end creation of this rule by pushing it into the Rules array.

```
rule = Hash.new()
rule["enabled"] = true
rule["sec_connect"] = true
rule["unscoped_consumers"] = false
rule["providers"] = [{"actors" => "ams"}]
rule["consumers"] = [{"actors" => "ams"}]
rule["service"] = {"href" => find_var("name", "All Services", services,
true)[0]}
rule["ub_service"] = {"href" => find_var("name", "All Services", services,
true)[0]}
rules.push(rule)
```

Next, we'll build the second rule. This rule allows traffic from the outside to come in, but only on UDP port 42 (the "fridgeomatic" Service we created earlier). This rule is similar, but not the same as that above. First, we don't want SecureConnect for this rule, so sec_connect is set to false. Also, the provider is a Label (a role Label specifically, named in the variable ring_role). We pull the HREF for that out of the Label hash. The consumer is an IP List. Like the provider Label, it's the IP List we created above, which we pull out by name (named by the variable ring_allowed_name). Note that it's not uncommon in a ringfencing rule to want "All IPv4 and IPv6 Addresses" as a consumer. If you wanted that, the consumer line would be the same, but instead of searching the ring_allowed_name variable (which is "fridge_command_central"), you'd search for the magic pre-built IP List called "Any (0.0.0.0/0 and ::/0)".

As above, we push this rule into the Rules array for later inclusion in the Ruleset.

```
rule = Hash.new()
rule["enabled"] = true
rule["sec_connect"] = false
rule["unscoped_consumers"] = false
rule["providers"] = [{"label" => {"href" => labels['role'][ring_role]}}]
rule["consumers"] = [{"ip_list" => {"href" => find_var("name",
ring_allowed_name, iplists, true)[0]}}]
rule["service"] = {"href" => find_var("name", ring_service, services,
true)[0]}
rule["ub_service"] = {"href" => find_var("name", "All Services", services,
true)[0]}
rules.push(rule)
```

With both Rules done, let's add the Rules to the Ruleset:

```
ruleset["rules"] = rules
```

The Ruleset has quite a few things in it now. Before we add the Ruleset via the API, let's have a look at the contents:

```
[1] pry(main)> ruleset
=> {"name"=>"fridge",
 "enabled"=>true,
 "scopes"=>
  [[{"label"=>{"href"=>"/orgs/9/labels/1146"}},
    {"label"=>{"href"=>"/orgs/9/labels/1148"}},
    {"label"=>{"href"=>"/orgs/9/labels/1147"}}]],
 "rules"=>
  [{"enabled"=>true,
    "sec_connect"=>true,
    "unscoped_consumers"=>false,
    "providers"=>[{"actors"=>"ams"}],
    "consumers"=>[{"actors"=>"ams"}],
    "service"=>{"href"=>"/orgs/9/sec_policy/draft/services/194"},
    "ub_service"=>{"href"=>"/orgs/9/sec_policy/draft/services/194"}},
   {"enabled"=>true,
    "sec_connect"=>false,
    "unscoped_consumers"=>false,
    "providers"=>[{"label"=>{"href"=>"/orgs/9/labels/1084"}}],
"consumers"=>[{"ip_list"=>{"href"=>"/orgs/9/sec_policy/draft/ip_lists/130"
}}],
    "service"=>{"href"=>"/orgs/9/sec_policy/draft/services/273"},
    "ub_service"=>{"href"=>"/orgs/9/sec_policy/draft/services/194"}}]}
[2] pry(main)>
```

So we've got a complete Ruleset now. Let's submit it:

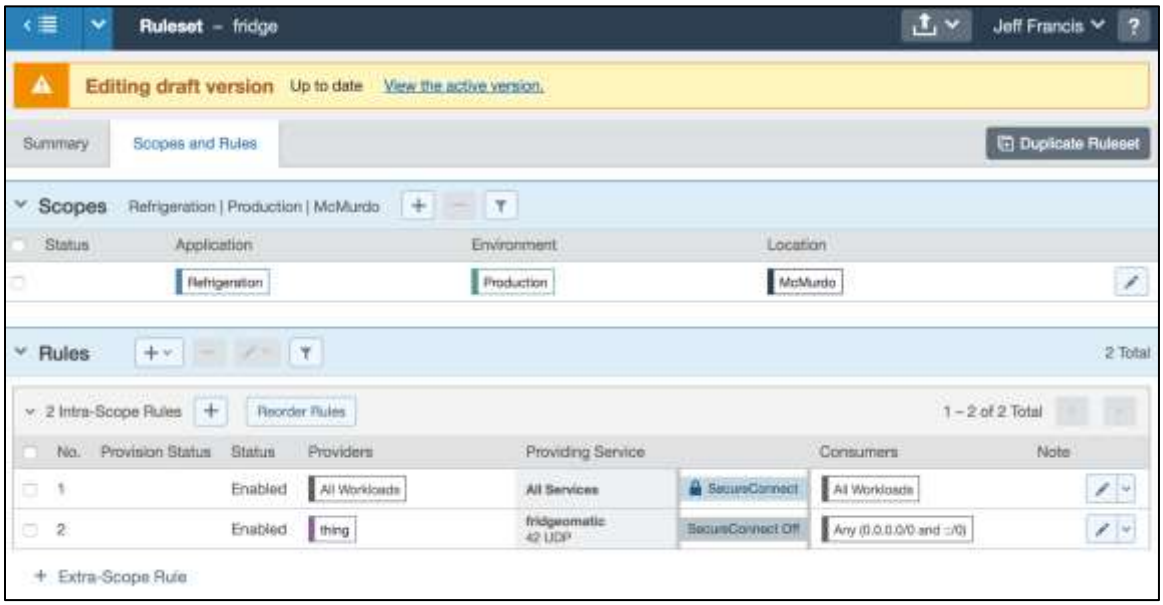
```
response = call_api(creds, :post, "/sec_policy/draft/rule_sets", true,
ruleset.to_json)
```

If you've got no errors (and it's not a duplicate), you should get back a blob of JSON indicating (among other things) the HREF of the new Ruleset. As always, real production code should check for success, and either fix, or at least signal to the user, any errors.

We're actually not done. The Ruleset we just created is actually not yet pushed to the Workloads. We need to commit (or Provision in Illumio-speak) the changes before they take effect. There's a whole section in the REST API Guide regarding Provisioning, but we're going to take the easy route and simply provision all outstanding changes in one go. Keep in mind that this could get a bit exciting (and not necessarily in a good way) if there are other unprovisioned changes that were made by other users of the system pending. Possibly even including other users only partially through their configuration steps. So use this with caution, unless your script is certain to be making the only changes to the system at the time it is run:

```
response = call_api(creds, :post, "/sec_policy", true, {"commit_message"
=> "penguins rule"}.to_json)
```

Assuming "response" indicates success, your new policy should be enforced in your network some time in the next minute or so. If you log into the PCE web console and view your new Ruleset, it should look similar to this:



The source code for this chapter can be found in Chapter 11 Source Code.

Chapter 12: The API Rosetta Stone

An example can make all of the difference in the world when learning. While the vast majority of DevOps/Sysops/SRE work we see at our customers is done in Ruby or Python, there are certainly exceptions to the rule. The tutorial in this book was written in Ruby, as that seems to satisfy Pareto's Law (more commonly known as "the 80/20 Rule"). That being said, Go and Bash are also commonly found in this space, and even Perl, C, and C++ pop up from time to time.

Not everybody will know (or want to learn) Ruby, and your organization may do DevOps/Sysops/SRE in another language. The following pages have one example per page of making an API call and returning a list of all Labels in various languages. This should be enough to get you "over the hump" to applying the lessons in the previous tutorial to your own environment.

The beauty of an open API is that it uses open standards for access, meaning that nearly any programming language can be used, provided you either have, or are willing to write, the proper libraries. If your preferred language is not listed in the following pages, feel free to construct a similar very small example of calling the Illumio API in your language and send it to us (file a ticket n the Illumio Support PagePortal), and we'll include it in future editions of this book (with an attribution to you as the submitter, if you give us permission).

Ruby

```
#!/usr/bin/env ruby


# Returns all known labels as a parsed Ruby data structure.


require 'json'
require 'rest-client'
login = "api_1ea0799f8bd486c8e"
passwd =
"6c1dbf2971aa1ff1a45b142fa86fc6040eeb9bbbe1957d4a977225bdf02d400d"
pce =
"https://"+login+":"+passwd+"@illumio.mycompany.com:443/api/v1/orgs/9/"


p JSON.parse(RestClient.get(pce+"labels",
                            {:accept => :json,
                                        :content_type => :json}))
```

This example requires the open source "rest-client" gem. On Linux/macOS systems, this can be installed with "sudo gem install rest-client". On Windows systems, it can be installed using rubyinstaller (or the gem manager of your choice).

Python

```
#!/usr/bin/python


# Returns all known labels as a parsed Python data structure.


import requests

login = "api_1ea0799f8bd486c8e”

passwd =
"6c1dbf2971aa1ff1a45b142fa86fc6040eeb9bbbe1957d4a977225bdf02d400d”

pce =
"https://"+login+":"+passwd+"@illumio.mycompany.com:443/api/v1/orgs/9/”


response = requests.get(pce+"labels",auth=(login,passwd))


print response.json()
```

Perl

```
#!/usr/bin/perl

# Returns all known labels as a JSON string.

use LWP::UserAgent;
use HTTP::Request;

my $login="api_1ea0799f8bd486c8e";
my
$passwd="6c1dbf2971aa1ff1a45b142fa86fc6040eeb9bbbe1957d4a97725bdf02d400d";
my $pce="https://illumio.mycompany.com:443/api/v1/orgs/9/";

my $browser=LWP::UserAgent->new();my $req=HTTP::Request->new();
$req->method('GET');$req->uri($pce . "labels");
$req->header("Content_Type"=>"application/json");
$req->header("Accept"=>"application/json");
$req->authorization_basic($login,$passwd);

print($browser->request($req)->content);
```

C

```
#include <stdio.h>
#include <curl/curl.h>

// Returns all known labels as a JSON string.

int main(void){
  CURL *curl;
  char *url="https://illumio.mycompany.com:443/api/v1/orgs/9/labels";
  CURLcode res;
  curl = curl_easy_init();
  curl_easy_setopt(curl, CURLOPT_URL, url);
  curl_easy_setopt(curl,CURLOPT_USERPWD,
"api_1ea0799f8bd486c8e:6c1dbf2971aa1ff1a45b142fa86fc6040eeb9bbbe1957d4a977
225bdf02d400d");
  curl_easy_setopt(curl, CURLOPT_SSL_VERIFYPEER, 0L);

  // The following function prints to STDOUT by default.

  curl_easy_perform(curl);
  curl_easy_cleanup(curl);
  return(0);
}
```

This example requires the open source libcurl library (https://curl.haxx.se/libcurl/). libcurl is also available as part of most Linux distros via the package manager, as well as the MacPorts and HomeBrew systems for macOS. For Windows, it can be downloaded and installed from the libcurl home page.

Go

```
package main

// Returns all known labels as a JSON string.

import (
_ "crypto/sha512"
"fmt"
"io/ioutil"
"net/http"
)

func getLabels(api_user string, api_key string, pce string) string {
client := &http.Client{}
req, err := http.NewRequest("GET", pce+"/labels", nil)
req.Header.Set("accept", "application/json")
req.Header.Set("content-type", "application/json")
req.SetBasicAuth(api_user, api_key)
resp, err := client.Do(req)
bodyText, err := ioutil.ReadAll(resp.Body)
return (string(bodyText))
}

func main() {
fmt.Printf("%+v\n", getLabels("api_1ea0799f8bd486c8e",
                  "6c1dbf2971aa1ff1a45b142fa86fc6040eeb9bbbe1957d4a977225b
df02d400d",
                  "https://illumio.mycompany.com:443/api/v1/orgs/9/"))
}
```

Go includes everything you need to talk to the Illumio API out of the box, but note the very unusual "_" on the "crypto/sha512" line in the source. Miss that, and your code will fail.

Common Lisp

```
(ql:quickload :drakma)
(ql:quickload :cl-json)


; Returns all known labels as an A-List.


(defparameter *login* "api_1ea0799f8bd486c8e")
(defparameter *passwd*
"6c1dbf2971aa1ff1a45b142fa86fc6040eeb9bbbe1957d4a977225bdf02d400d")
(defparameter *pce* "https://illumio.mycompany.com:443/api/v1/orgs/9/")


(defun get-labels (login passwd pce)
  (let ((result (drakma:http-request (concatenate 'string pce "labels")
                                                  :method :get
:accept "application/json"
:content-type "application/json"
:basic-authorization (list login passwd))))
(json:decode-json-from-string (map 'string #'code-char (nth-value 0
result)))))


(princ (get-labels *login* *passwd* *pce*))
```

This example requires the Quicklisp package manager (https://www.quicklisp.org/beta/), which in turn requires the Drakma library (for HTTP/HTTPS/REST) support as well as cl-json for JSON parsing.

Bash/Curl

```
#!/bin/bash

# Returns all known labels as a parsed JSON string.

LOGIN="api_1ea0799f8bd486c8e”
PASSWD="6c1dbf2971aa1ff1a45b142fa86fc6040eeb9bbbe1957d4a977225bdf02d400d”
PCE=https://illumio.mycompany.com:443/api/v1/orgs/9/
FLAG=””

curl -s -k -u ${LOGIN}:${PASSWD} \
     -H "Accept: application/json" \
     -X GET ${PCE}"labels" | ./JSON.sh | while read -r LINE; do

    TMP=`echo ${LINE} | awk '{ print $1 }' | tr -d '[]'`
    ID=`echo ${TMP} | awk -F, '{ print $1 }'`
    TYPE=`echo ${TMP} | awk -F, '{ print $2 }' | tr -d '\"'`

    if [[ ${ID} != ${FLAG} ]]; then
                HREF=""
                KEY=""
                VALUE=""
                FLAG=${ID}
    fi

    case "$TYPE" in
                href )
                    HREF=`echo ${LINE} | awk '{ print $2 }' | tr -d '\"'`
                    ;;
                key )
                    KEY=`echo ${LINE} | awk '{ print $2 }' | tr -d '\"'`
                    ;;
                value )
                    VALUE=`echo ${LINE} | awk '{ print $2 }' | tr -d '\"'`
                    ;;
    esac
```

```
    if [[ ${KEY} != "" && ${VALUE} != "" && ${HREF} != "" ]]; then
                echo "KEY = ${KEY}"
                echo "VALUE = ${VALUE}"
                echo "HREF = ${HREF}"
                echo ""
                FLAG=""
    fi
done
```

Note that while a simple Bash example is provided, Bash is not a recommended way to use the Illumio API for any but the most trivial tasks. This example makes use of the Bourne Shell JSON parser (linked below). Use of shell with the Illumio REST API is not recommended for any but the most visionary (or desperate) customers.

https://github.com/dominictarr/JSON.sh

Chapter 13: Conclusion

If you've made it this far in the book, you should now be at least somewhat comfortable tackling the Illumio API for automating security in your environment. While the requirements of using the API are exacting, they're certainly within the reach of anyone willing to take the time to learn. With a moderate amount of effort, it's possible to integrate Illumio into your automated flow-through provisioning system, and create security for new systems in real time as the systems themselves are built and deployed.

While every effort has been made to validate the examples used in this book, it's likely that there are mistakes. They may be mistakes that slipped past the proofreading of the book, or they may be changes required by changes to the API itself. This book was completed using version 17.1 of the PCE, just prior to the publication of 17.2. It's expected that this book will be updated regularly, but keep in mind that there are minor differences between revisions of the PCE. Fortunately, the vast majority of those revisions involve additional API calls and additional options to existing calls, not major changes to the pre-existing API.

While the code in this book is written in Ruby, the principles involved in using the API apply equally to a wide variety of languages. Illumio has customers who have deployed automated systems involving many tens of thousands of VENs in Ruby, Python, and Go. Your author typically writes API code in Common Lisp as a first choice, demonstrating that the techniques in this book can even apply to programming languages more than 60 years old (that being said, I wouldn't want to try the same thing in Fortran or COBOL).

As you write more and more API code, you'll start building your own library of commonly used functions. Save these, and refine them. Add error-checking and bounds-checking. Polish them, and build your own library. It's a common thing in the programming world to build an entire library around an API, reducing API programming to a series of function calls, allowing you to build code that completely hides the various tasks of interfacing with the API. As you write more and more API code, you'll soon find that you can leverage your old code and solve problems more and more quickly. Once you've built a library to abstract the API, you might find that a simple application might only take a few dozen lines of new code, making it very quick and easy to automate security tasks.

Don't miss out on the non-security aspects of the product. Various pieces of very valuable business information are available via the Illumio API. You can extract an inventory of operating systems and their specific versions, IP addresses, system names, ports in use, software packages installed, physical locations, and user information all from the API. Once labelled, the Labels themselves are useful for calculating counts, locations, and uses of various computing resources within an

organization. This data can be used for a huge number of business purposes, and most of our customers soon find the Illumio infrastructure to be the source of record for accurate system data.

If you find a mistake, whether in the text or in the code, please feel free to file a bug with Illumio, and it will be corrected. A current copy of all code contained in this book will be maintained on the Illumio Support Portal for download.

Appendix: Source Code Samples

Chapter 2 Source Code

```
#!/usr/bin/env ruby

require 'json'
require 'rest-client'

login = "jeff.francis@illumio.com"
passwd = "SecretPassword"
pce = "illumio.mycompany.com"
port = "443"

response = RestClient::Request.new(
  :url =>
"https://"+pce+":"+port+"/api/v1/login_users/authenticate?pce_fqdn="+pce,
  :method => :post,
  :user => login,
  :password => passwd,
  :headers => {:accept => :json,
               :content_type => :json}).execute()

json = JSON.parse(response)

auth_token = json['auth_token']

response = RestClient::Request.new(
  :url => "https://"+pce+":"+port+"/api/v1/users/login",
  :method => :get,
  :user => login,
  :password => passwd,
  :headers => {:accept => :json,
               :content_type => :json,
               "Authorization" => "Token token="+auth_token+""}).execute()
```

```
json = JSON.parse(response)

org_id = json['orgs'][0]['org_href']

puts("Your org HREF is: #{org_id}")
```

Chapter 3 Source Code

```
#!/usr/bin/env ruby

require 'json'
require 'rest-client'

login = "api_1ea0799f8bd486c8e"
passwd =
"6c1dbf2971aa1ff1a45b142fa86fc6040eeb9bbbe1957d4a977225bdf02d400d"
pce = "illumio.company.com"
port = "443"
org_href = "/orgs/9"

response = RestClient::Request.new(
  :url => "https://"+pce+":"+port+"/api/v1"+org_href+"/labels",
  :method => :get,
  :user => login,
  :password => passwd,
  :headers => {:accept => :json,
               :content_type => :json}).execute()

json = JSON.parse(response)

json.each do |j|
  puts "href: #{j['href']}"
  puts "type: #{j['key']}"
  puts "name: #{j['value']}"
  puts ""
end
```

Chapter 4 Source Code

```
#!/usr/bin/env ruby

require 'json'
require 'rest-client'

login = "api_1ea0799f8bd486c8e"
passwd =
"6c1dbf2971aa1ff1a45b142fa86fc6040eeb9bbbe1957d4a977225bdf02d400d"
pce = "illumio.company.com"
port = "443"
org_href = "/orgs/9"

#RestClient.log = 'stderr'
response = RestClient::Request.new(
  :url => "https://"+pce+":"+port+"/api/v1"+org_href+"/labels",
  :payload => {"key" => "app", "value" => "web"}.to_json.to_str,
  :method => :post,
  :user => login,
  :password => passwd,
  :headers => {:accept => :json,
               :content_type => :json}).execute()

puts response

json = JSON.parse(response)

puts json
```

Chapter 5 Source Code

```
#!/usr/bin/env ruby


require 'json'
require 'rest-client'
require 'csv'


login = "api_1ea0799f8bd486c8e"
passwd =
"6c1dbf2971aa1ff1a45b142fa86fc6040eeb9bbbe1957d4a977225bdf02d400d"
pce = "illumio.company.com"
port = "443"
org_href = "/orgs/9"


File.open("labels.csv").each do |line|
  CSV.parse(line) do |stuff|
    json = JSON.parse(
      RestClient::Request.new(
        :url => "https://"+pce+":"+port+"/api/v1"+org_href+"/labels",
        :payload => {"key" => stuff[0], "value" =>
stuff[1]}.to_json.to_str,
        :method => :post,
        :user => login,
        :password => passwd,
        :headers => {:accept => :json,
                     :content_type => :json}).execute())
    puts json
  end
end
```

Chapter 6 Source Code

```
#!/usr/bin/env ruby

require 'json'
require 'rest-client'

login = "api_1ea0799f8bd486c8e"
passwd =
"6c1dbf2971aa1ff1a45b142fa86fc6040eeb9bbbe1957d4a977225bdf02d400d"
pce = "illumio.company.com"
port = "443"
org_href = "/orgs/9"

response = RestClient::Request.new(
  :url => "https://"+pce+":"+port+"/api/v1"+org_href+"/labels",
  :method => :get,
  :user => login,
  :password => passwd,
  :headers => {"Prefer" => "respond-async",
               :accept => :json,
               :content_type => :json}).execute()

sleep(response.headers[:retry_after].to_i)

status=""
location=response.headers[:location]
while (status!="done" and status!="failed")
  response = JSON.parse(RestClient::Request.new(
                          :url =>
"https://"+pce+":"+port+"/api/v1/"+location,
                          :method => :get,
                          :user => login,
                          :password => passwd,
                          :headers => {:accept => :json,
                                       :content_type => :json}).execute())
  status=response['status']
  puts status
  sleep 1 unless (status=="done" or status=="failed")
```

```
end

url=response["result"]["href"]

response = JSON.parse(RestClient::Request.new(
                        :url => "https://"+pce+":"+port+"/api/v1"+url,
                        :method => :get,
                        :user => login,
                        :password => passwd,
                        :headers => {:accept => :json,
                                     :content_type => :json}).execute())

response.each do |resp|
  puts "HREF: #{resp['href']}"
  puts "type: #{resp['key']}"
  puts "name: #{resp['value']}"
  puts ""
end
```

Chapter 7 Source Code

```
#!/usr/bin/env ruby

require 'json'
require 'rest-client'

login = "api_1ea0799f8bd486c8e"
passwd =
"6c1dbf2971aa1ff1a45b142fa86fc6040eeb9bbbe1957d4a977225bdf02d400d"
pce = "illumio.company.com"
port = "443"
org_href = "/orgs/9"

# This object holds credentials for a PCE.
class Creds
  attr_accessor :login, :passwd, :pce, :port, :org

  def initialize(login, passwd, pce, org, port=443)
    @login = login
    @passwd = passwd
    @pce = pce
    @port = port.to_s
    @org = org
  end

  def url_with_api(rest)
    if(rest[0] == "/")
      rest = rest[1..-1]
    end
    return("https://#{@pce}:#{@port}/api/v1/#{rest}")
  end

  def url_with_org(rest)
    if(rest[0] == "/")
      rest = rest[1..-1]
    end
    return("https://#{@pce}:#{@port}/api/v1#{@org}/#{rest}")
  end
```

```
end

# Makes a synchronous API call using a pre-populated Creds object, a
# type (:get, :post, :put, :delete), the specific call you're making
# to the API (such as "/labels"), whether you want the org specified
# as part of the full URL or not (this will be nil when supplying a
# full HREF for "resource"), and an optional payload (this will be
# ignored in cases it makes no sense).
def sync_api(creds,type,resource,org,payload = nil)
  api = ""
  if (org)
    api = creds.url_with_org(resource)
  else
    api = creds.url_with_api(resource)
  end

  response = RestClient::Request.new(
    :url => api,
    :method => type,
    :payload => payload,
    :user => creds.login,
    :password => creds.passwd,
    :verify_ssl => false,
    :headers => {:accept => :json,
                 :content_type => :json}).execute()

  if (response.to_s() != "")
    return(JSON.parse(response))
  else
    return "ok"
  end
end

# Makes an asynchronous API call using a pre-populated Creds object, a
# type (async calls can only be of type :get, but the field is still
# here for consistency with the sync api call, however this value will
# be ignored), the specific call you're making to the API (such as
# "/labels"), whether you want the org specified as part of the full
```

```
# URL or not (this will be nil when supplying a full HREF for
# "resource"), and an optional payload (payloads make no sense for async
# GET calls, so this field will be ignored).
def async_api(creds,type,resource,org,payload = nil)
  api = ""
  if (org)
    api = creds.url_with_org(resource)
  else
    api = creds.url_with_api(resource)
  end

  response = RestClient::Request.new(
    :url => api,
    :method => :get,
    :user => creds.login,
    :password => creds.passwd,
    :headers => {"Prefer" => "respond-async",
                 :accept => :json,
                 :content_type => :json}).execute()

  sleep(response.headers[:retry_after].to_i)

  monitor_url = response.headers[:location]
  status = ""
  while (status != "done" and status != "failed")
    response = sync_api(creds,:get,monitor_url,nil)
    status = response['status']
    sleep 1 unless (status == "done" or status == "failed")
  end

  return (sync_api(creds,:get,response["result"]["href"],nil))
end

# Create a Creds object holding everything we need to know about a PCE.
creds = Creds.new(login,passwd,pce,org_href,port)

puts "Getting labels via a sync API call:\n\n"
```

```
# Get the labels with a sync call...
response = sync_api(creds,:get,"labels",true)

# Show me the result of my call.
response.each do |resp|
  puts "HREF: #{resp['href']}"
  puts "type: #{resp['key']}"
  puts "name: #{resp['value']}"
  puts ""
end

puts "\n\nGetting labels via an async API call:\n\n"

# Now get the labels with an async call...
response = async_api(creds,:get,"labels",true)

# Show me the result of my call.
response.each do |resp|
  puts "HREF: #{resp['href']}"
  puts "type: #{resp['key']}"
  puts "name: #{resp['value']}"
  puts ""
end
```

Chapter 8 Source Code

```
#!/usr/bin/env ruby

require 'json'
require 'rest-client'
require 'csv'

login = "api_1ea0799f8bd486c8e"
passwd =
"6c1dbf2971aa1ff1a45b142fa86fc6040eeb9bbbe1957d4a977225bdf02d400d"
pce = "illumio.company.com"
port = "443"
org_href = "/orgs/9"

# This object holds credentials for a PCE.
class Creds
  attr_accessor :login, :passwd, :pce, :port, :org

  def initialize(login, passwd, pce, org, port=443)
    @login = login
    @passwd = passwd
    @pce = pce
    @port = port.to_s
    @org = org
  end

  def url_with_api(rest)
    if(rest[0] == "/")
      rest = rest[1..-1]
    end
    return("https://#{@pce}:#{@port}/api/v1/#{rest}")
  end

  def url_with_org(rest)
    if(rest[0] == "/")
      rest = rest[1..-1]
    end
    return("https://#{@pce}:#{@port}/api/v1#{@org}/#{rest}")
```

```
  end
end

# Makes a synchronous API call using a pre-populated Creds object, a
# type (:get, :post, :put, :delete), the specific call you're making
# to the API (such as "/labels"), whether you want the org specified
# as part of the full URL or not (this will be nil when supplying a
# full HREF for "resource"), and an optional payload (this will be
# ignored in cases it makes no sense).
def sync_api(creds,type,resource,org,payload = nil)
  api = ""
  if (org)
    api = creds.url_with_org(resource)
  else
    api = creds.url_with_api(resource)
  end

  response = RestClient::Request.new(
    :url => api,
    :method => type,
    :payload => payload,
    :user => creds.login,
    :password => creds.passwd,
    :verify_ssl => false,
    :headers => {:accept => :json,
                 :content_type => :json}).execute()

  if (response.to_s() != "")
    return(JSON.parse(response))
  else
    return "ok"
  end
end

# Makes an asynchronous API call using a pre-populated Creds object, a
# type (async calls can only be of type :get, but the field is still
# here for consistency with the sync api call, however this value will
# be ignored), the specific call you're making to the API (such as
```

```
# "/labels"), whether you want the org specified as part of the full
# URL or not (this will be nil when supplying a full HREF for
# "resource"), and an optional payload (payloads make no sense for async
# GET calls, so this field will be ignored).
def async_api(creds,type,resource,org,payload = nil)
  api = ""
  if (org)
    api = creds.url_with_org(resource)
  else
    api = creds.url_with_api(resource)
  end

  response = RestClient::Request.new(
    :url => api,
    :method => :get,
    :user => creds.login,
    :password => creds.passwd,
    :headers => {"Prefer" => "respond-async",
                 :accept => :json,
                 :content_type => :json}).execute()

  sleep(response.headers[:retry_after].to_i)

  monitor_url = response.headers[:location]
  status = ""
  while (status != "done" and status != "failed")
    response = sync_api(creds,:get,monitor_url,nil)
    status = response['status']
    sleep 1 unless (status == "done" or status == "failed")
  end

  return (sync_api(creds,:get,response["result"]["href"],nil))
end

# Return a hash of hashes containing all labels and their HREFs.
def get_all_labels(creds)
  response = sync_api(creds,:get,"/labels",true)
  if (response.length == 500)
```

```
    response = async_api(creds,:get,"/labels",true)
  end

  labels = Hash.new()
  labels['role'] = Hash.new()
  labels['app'] = Hash.new()
  labels['env'] = Hash.new()
  labels['loc'] = Hash.new()

  response.each do |resp|
    if (resp['key'] == "role")
      labels['role'][resp['value']] = resp['href']
    end

    if (resp['key'] == "app")
      labels['app'][resp['value']] = resp['href']
    end

    if (resp['key'] == "env")
      labels['env'][resp['value']] = resp['href']
    end

    if (resp['key'] == "loc")
      labels['loc'][resp['value']] = resp['href']
    end
  end

  return(labels)
end

# Create an unmanaged workload. "creds" is the PCE Creds object,
# "name" is the name of the workload to be displayed in the PCE,
# "hostname" is the full hostname of the workload (can be a null
# string - ""), "ip" is the ip address of the unmanaged workload as a
# string, and lab1, lab2, lab3, and lab4 are HREFs for the labels to
# be applied (all optional). Note that unlike labels, it's possible
# (and very confusing) to create duplicate unmanaged workloads, so be
# careful. Returns the JSON specifying the new unmanaged workload.
```

```
def create_umw(creds,name,hostname,ip,lab1=nil,lab2=nil,lab3=nil,lab4=nil)
  lab = Array.new()

  lab.push ({"href" => lab1}) if (lab1)
  lab.push ({"href" => lab2}) if (lab2)
  lab.push ({"href" => lab3}) if (lab3)
  lab.push ({"href" => lab4}) if (lab4)

  wl = {"name" => name,
        "hostname" => hostname,
        "public_ip" => ip,
        "interfaces" =>
        [{"name" => "eth0",
          "address" => ip,
          "cidr_block" => 32,
          "link_state" => "up"}],
        "online" => true,
        "labels" => lab }

  return(
    JSON.parse(
      sync_api(creds,:post,"/workloads",true,wl.to_json)))
end

# Create a label in the PCE. "type" must be one of "role", "app",
# "env", "loc" (note that this is not checked). "value" is the name
# given to the label. Does not check for duplicates before creating,
# and fails if a label of this name and type already exist. Returns
# the JSON specifying the new label.
def create_label(creds,type,name)
  return(
    JSON.parse(sync_api(creds,:post,"/labels",true,
                        {"key" => type, "value" =>
name}.to_json).to_json))
end

# Create a Creds object holding everything we need to know about a PCE.
creds = Creds.new(login,passwd,pce,org_href,port)
```

```
# Fetch all of the labels from the PCE.
labels = get_all_labels(creds)

File.open("workloads.csv").each do |line|
  CSV.parse(line) do |stuff|
    name = stuff[0]
    hostname = stuff[1]
    ip = stuff[2]
    role = stuff[3]
    app = stuff[4]
    env = stuff[5]
    loc = stuff[6]

    # Get the role HREF, or create it if it doesn't already exist.
    if (role != nil)
      if (labels['role'][role])
        puts "Role #{role} already exists."
        role = labels['role'][role]
      else
        puts "Creating role #{role}..."
        href = create_label(creds,"role",role)['href']
        labels['role'][role] = href
        role = href
      end
    else
      puts "Role not specified."
      role = nil
    end

    # Get the app HREF, or create it if it doesn't already exist.
    if (app != nil)
      if (labels['app'][app])
        puts "App #{app} already exists."
        app = labels['app'][app]
      else
        puts "Creating app #{app}..."
        href = create_label(creds,"app",app)['href']
        labels['app'][app] = href
```

```
    app = href
  end
else
  puts "App not specified."
  app = nil
end

# Get the env HREF, or create it if it doesn't already exist.
if (env != nil)
  if (labels['env'][env])
    puts "Env #{env} already exists."
    env = labels['env'][env]
  else
    puts "Creating env #{env}..."
    href = create_label(creds,"env",env)['href']
    labels['env'][env] = href
    env = href
  end
else
  puts "Env not specified."
  env = nil
end

# Get the loc HREF, or create it if it doesn't already exist.
if (loc != nil)
  if (labels['loc'][loc])
    puts "Loc #{loc} already exists."
    loc = labels['loc'][loc]
  else
    puts "Creating loc #{loc}..."
    href = create_label(creds,"loc",loc)['href']
    labels['loc'][loc] = href
    loc = href
  end
else
  puts "Loc not specified."
  loc = nil
```

```
end
    puts "Creating unmanaged workload #{name}...\n\n"
    wl = create_umw(creds,name,hostname,ip,
                    role,app,env,loc);
  end
end
```

Chapter 9 Source Code

```
#!/usr/bin/env ruby

infected_ip = "10.10.10.10"

require 'json'
require 'rest-client'
require 'csv'

login = "api_1ea0799f8bd486c8e"
passwd =
"6c1dbf2971aa1ff1a45b142fa86fc6040eeb9bbbe1957d4a977225bdf02d400d"
pce = "illumio.company.com"
port = "443"
org_href = "/orgs/9"

qrole = "quarantine"
qapp = "quarantine"
qenv = "quarantine"
qloc = "quarantine"

# This object holds credentials for a PCE.
class Creds
  attr_accessor :login, :passwd, :pce, :port, :org

  def initialize(login, passwd, pce, org, port=443)
    @login = login
    @passwd = passwd
    @pce = pce
    @port = port.to_s
    @org = org
  end

  def url_with_api(rest)
    if(rest[0] == "/")
      rest = rest[1..-1]
    end
    return("https://#{@pce}:#{@port}/api/v1/#{rest}")
```

```
  end

  def url_with_org(rest)
    if(rest[0] == "/")
      rest = rest[1..-1]
    end
    return("https://#{@pce}:#{@port}/api/v1#{@org}/#{rest}")
  end
end

# Makes a synchronous API call using a pre-populated Creds object, a
# type (:get, :post, :put, :delete), the specific call you're making
# to the API (such as "/labels"), whether you want the org specified
# as part of the full URL or not (this will be nil when supplying a
# full HREF for "resource"), and an optional payload (this will be
# ignored in cases it makes no sense).
def sync_api(creds,type,resource,org,payload = nil)
  api = ""
  if (org)
    api = creds.url_with_org(resource)
  else
    api = creds.url_with_api(resource)
  end

  response = RestClient::Request.new(
    :url => api,
    :method => type,
    :payload => payload,
    :user => creds.login,
    :password => creds.passwd,
    :verify_ssl => false,
    :headers => {:accept => :json,
                 :content_type => :json}).execute()

  if (response.to_s() != "")
    return(JSON.parse(response))
  else
    return "ok"
```

```
  end
end

# Makes an asynchronous API call using a pre-populated Creds object, a
# type (async calls can only be of type :get, but the field is still
# here for consistency with the sync api call, however this value will
# be ignored), the specific call you're making to the API (such as
# "/labels"), whether you want the org specified as part of the full
# URL or not (this will be nil when supplying a full HREF for
# "resource"), and an optional payload (payloads make no sense for async
# GET calls, so this field will be ignored).
def async_api(creds,type,resource,org,payload = nil)
  api = ""
  if (org)
    api = creds.url_with_org(resource)
  else
    api = creds.url_with_api(resource)
  end

  response = RestClient::Request.new(
    :url => api,
    :method => :get,
    :user => creds.login,
    :password => creds.passwd,
    :headers => {"Prefer" => "respond-async",
                 :accept => :json,
                 :content_type => :json}).execute()

  sleep(response.headers[:retry_after].to_i)

  monitor_url = response.headers[:location]
  status = ""
  while (status != "done" and status != "failed")
    response = sync_api(creds,:get,monitor_url,nil)
    status = response['status']
    sleep 1 unless (status == "done" or status == "failed")
  end
```

```
  return (sync_api(creds,:get,response["result"]["href"],nil))
end

# Return a hash of hashes containing all labels and their HREFs.
def get_all_labels(creds)
  response = sync_api(creds,:get,"/labels",true)
  if (response.length == 500)
    response = async_api(creds,:get,"/labels",true)
  end

  labels = Hash.new()
  labels['role'] = Hash.new()
  labels['app'] = Hash.new()
  labels['env'] = Hash.new()
  labels['loc'] = Hash.new()

  response.each do |resp|
    if (resp['key'] == "role")
      labels['role'][resp['value']] = resp['href']
    end

    if (resp['key'] == "app")
      labels['app'][resp['value']] = resp['href']
    end

    if (resp['key'] == "env")
      labels['env'][resp['value']] = resp['href']
    end

    if (resp['key'] == "loc")
      labels['loc'][resp['value']] = resp['href']
    end
  end

  return(labels)
end

# Return a hash of all workloads indexed by HREF.
```

```
def get_all_workloads(creds)
  workloads = Hash.new()

  response = sync_api(creds,:get,"/workloads",true)
  if (response.length == 500)
    response = async_api(creds,:get,"/workloads",true)
  end

  response.each do |wl|
    workloads[wl['href']] = wl
  end

  return(workloads)
end

# Given an IP address and the hash of all workloads, return an array
# of HREFs of workloads using this IP (in theory, it should always be
# zero or one result, but we allow for more for odd circumstances).
def find_ip_in_workloads(ip,workloads)
  matches = Array.new()

  workloads.keys.each do |wl|
    if(workloads[wl]['public_ip'] == ip)
      matches.push(wl)
    end

    workloads[wl]['interfaces'].each do |iface|
      if(iface['address'] == ip)
        matches.push(wl)
      end
    end
  end

  return(matches.uniq())
end

# Create a Creds object holding everything we need to know about a PCE.
creds = Creds.new(login,passwd,pce,org_href,port)
```

```
# Fetch all of the labels from the PCE.
labels = get_all_labels(creds)

# Grab the HREFs we need for the specified quarantine labels.
qrole_href = labels['role'][qrole]
qapp_href = labels['app'][qapp]
qenv_href = labels['env'][qenv]
qloc_href = labels['loc'][qloc]

# Make sure we found the specified labels in the PCE.
unless(qrole_href and qapp_href and qenv_href and qloc_href)
  puts "Unable to locate specified quarantine labels."
  exit(1)
end

# Fetch all workloads from the PCE.
workloads = get_all_workloads(creds)

# Find the specified workload(s).
infected = find_ip_in_workloads(infected_ip, workloads)

# Make sure we found the specified IP. If we did find it, change it's
# labels to the quarantine labels. If we didn't find it, create an
# unmanaged workload to represent it, then set the labels of that
# unmanaged workload to the quarantine labels.
lab = Array.new()

lab.push ({"href" => qrole_href})
lab.push ({"href" => qapp_href})
lab.push ({"href" => qenv_href})
lab.push ({"href" => qloc_href})

if(infected == [])

  puts "Creating new quarantined workload: infected_host_#{infected_ip}"

  wl = {"name" => "infected_host_" + infected_ip,
```

```
        "hostname" => "bad",
        "public_ip" => infected_ip,
        "interfaces" => [
          {"name" => "eth0",
           "address" => infected_ip,
           "cidr_block" => 32,
           "link_state" => "up"} ],
        "online" => true,
        "labels" => lab }

  sync_api(creds,:post,"/workloads",true,wl.to_json)

else

  puts "Quarantining existing workload: #{infected_ip}"

  wl = { "labels" => lab }
  sync_api(creds,:put,infected[0],nil,wl.to_json)

end

# Local Variables:
# mode: Ruby
# coding: utf-8
# End:
```

Chapter 10 Source Code

```
#!/usr/bin/env ruby
# coding: utf-8

require 'json'
require 'rest-client'
require 'csv'

login = "api_1ea0799f8bd486c8e"
passwd =
"6c1dbf2971aa1ff1a45b142fa86fc6040eeb9bbbe1957d4a977225bdf02d400d"
pce = "illumio.company.com"
port = "443"
org_href = "/orgs/9"

pprofile_name = "Pair Me"
pprofile_app = "Boondoggle"
pprofile_env = "Production"
pprofile_loc = "Addis Ababa"

# This object holds credentials for a PCE.
class Creds
  attr_accessor :login, :passwd, :pce, :port, :org

  def initialize(login, passwd, pce, org, port=443)
    @login = login
    @passwd = passwd
    @pce = pce
    @port = port.to_s
    @org = org
  end

  def url_with_api(rest)
    if(rest[0] == "/")
      rest = rest[1..-1]
    end
    return("https://#{@pce}:#{@port}/api/v1/#{rest}")
  end
```

```
  def url_with_org(rest)
    if(rest[0] == "/")
      rest = rest[1..-1]
    end
    return("https://#{@pce}:#{@port}/api/v1#{@org}/#{rest}")
  end
end

# Makes a synchronous API call using a pre-populated Creds object, a
# type (:get, :post, :put, :delete), the specific call you're making
# to the API (such as "/labels"), whether you want the org specified
# as part of the full URL or not (this will be nil when supplying a
# full HREF for "resource"), and an optional payload (this will be
# ignored in cases it makes no sense).
def sync_api(creds, type, resource, org, payload = nil)
  api = ""
  if (org)
    api = creds.url_with_org(resource)
  else
    api = creds.url_with_api(resource)
  end

  response = RestClient::Request.new(
    :url => api,
    :method => type,
    :payload => payload,
    :user => creds.login,
    :password => creds.passwd,
    :verify_ssl => false,
    :headers => {:accept => :json,
                 :content_type => :json}).execute()

  if (response.to_s() != "")
    return(JSON.parse(response))
  else
    return "ok"
  end
```

```
end

# Makes an asynchronous API call using a pre-populated Creds object, a
# type (async calls can only be of type :get, but the field is still
# here for consistency with the sync api call, however this value will
# be ignored), the specific call you're making to the API (such as
# "/labels"), whether you want the org specified as part of the full
# URL or not (this will be nil when supplying a full HREF for
# "resource"), and an optional payload (payloads make no sense for async
# GET calls, so this field will be ignored).
def async_api(creds, type, resource, org, payload = nil)
  api = ""
  if (org)
    api = creds.url_with_org(resource)
  else
    api = creds.url_with_api(resource)
  end

  response = RestClient::Request.new(
    :url => api,
    :method => :get,
    :user => creds.login,
    :password => creds.passwd,
    :headers => {"Prefer" => "respond-async",
                 :accept => :json,
                 :content_type => :json}).execute()

  sleep(response.headers[:retry_after].to_i)

  monitor_url = response.headers[:location]
  status = ""
  while (status != "done" and status != "failed")
    response = sync_api(creds, :get, monitor_url, nil)
    status = response['status']
    sleep 1 unless (status == "done" or status == "failed")
  end

  return (sync_api(creds, :get, response["result"]["href"], nil))
```

```
end

# Given a type ("role", "app", "env", or "loc"), a name, and a creds
# object, create a label with the specified name. Does not check to
# see if a label of the specified type and name already exist, and
# fails if this is the case.
def create_label(type, name, creds)
  return(sync_api(creds, :post, "/labels", true, {"key" => type, "value"
=> name}.to_json))
end

# Return a hash of hashes containing all labels and their HREFs.
def get_all_labels(creds)
  response = sync_api(creds, :get, "/labels", true)

  labels = Hash.new()
  labels['role'] = Hash.new()
  labels['app'] = Hash.new()
  labels['env'] = Hash.new()
  labels['loc'] = Hash.new()

  response.each do |resp|
    if (resp['key'] == "role")
      labels['role'][resp['value']] = resp['href']
    end

    if (resp['key'] == "app")
      labels['app'][resp['value']] = resp['href']
    end

    if (resp['key'] == "env")
      labels['env'][resp['value']] = resp['href']
    end

    if (resp['key'] == "loc")
      labels['loc'][resp['value']] = resp['href']
    end
  end
```

```
  return(labels)
end

# Create a Creds object holding everything we need to know about a PCE.
creds = Creds.new(login, passwd, pce, org_href,port)

# Fetch all of the labels from the PCE.
labels = get_all_labels(creds)

# Make the new labels for our McMurdo service (if needed). Also, store
# the data for the new services we create in the global 'labels' var
# for future reference.
unless labels["app"][pprofile_app]
  labels["app"][pprofile_app]=create_label("app", pprofile_app,
creds)["href"]
end
unless labels["env"][pprofile_env]
  labels["env"][pprofile_env]=create_label("env", pprofile_env,
creds)["href"]
end
unless labels["loc"][pprofile_loc]
  labels["loc"][pprofile_loc]=create_label("loc", pprofile_loc,
creds)["href"]
end


pprofile = Hash.new()
pprofile["name"] = pprofile_name
pprofile["enabled"] = true
pprofile["mode"] = "illuminated"
pprofile["key_lifespan"] = 86400
pprofile["allowed_uses_per_key"] = "unlimited"
pprofile["role_label_lock"] = false
pprofile["app_label_lock"] = true
pprofile["env_label_lock"] = true
pprofile["loc_label_lock"] = true
pprofile["mode_lock"] = false
pprofile["log_traffic"] = true
pprofile["log_traffic_lock"] = true
pprofile["visibility_level"] = "flow_summary"
```

```
pprofile["visibility_level_lock"] = true

lab = Array.new()
lab.push({"href"=>labels["app"][pprofile_app]})
lab.push({"href"=>labels["loc"][pprofile_loc]})
lab.push({"href"=>labels["env"][pprofile_env]})
pprofile["labels"] = lab

# Now push the pairing profile to the PCE. If it already exists (or if
# there is any other error), this fails.
result = sync_api(creds, :post, "/pairing_profiles",
                  true, pprofile.to_json)["href"]
href = result["href"]

# Fetch and print the activation code.
result = sync_api(creds, :post, href + "/pairing_key",
                  false, {}.to_json)
activation_code = result["activation_code"]

puts "Activation code: #{activation_code}"
```

Chapter 11 Source Code

```
#!/usr/bin/env ruby
# coding: utf-8

require 'json'
require 'rest-client'
require 'csv'

login = "api_1ea0799f8bd486c8e"
passwd =
"6c1dbf2971aa1ff1a45b142fa86fc6040eeb9bbbe1957d4a977225bdf02d400d"
pce = "illumio.company.com"
port = "443"
org_href = "/orgs/9"

ring_app = "Refrigeration"
ring_env = "Production"
ring_loc = "McMurdo"
ring_role = "thing"
ring_ruleset = "fridge"
ring_port = "42"
ring_proto = "17"
ring_service = "fridgeomatic"
ring_allowed_net = "10.246.0.1"
ring_allowed_netmask = "32"
ring_allowed_name = "fridge_command_central"

# This object holds credentials for a PCE.
class Creds
  attr_accessor :login, :passwd, :pce, :port, :org

  def initialize(login, passwd, pce, org, port=443)
    @login = login
    @passwd = passwd
    @pce = pce
    @port = port.to_s
    @org = org
  end
```

```
  def url_with_api(rest)
    if(rest[0] == "/")
      rest = rest[1..-1]
    end
    return("https://#{@pce}:#{@port}/api/v1/#{rest}")
  end

  def url_with_org(rest)
    if(rest[0] == "/")
      rest = rest[1..-1]
    end
    return("https://#{@pce}:#{@port}/api/v1#{@org}/#{rest}")
  end
end

# Makes a synchronous API call using a pre-populated Creds object, a
# type (:get, :post, :put, :delete), the specific call you're making
# to the API (such as "/labels"), whether you want the org specified
# as part of the full URL or not (this will be nil when supplying a
# full HREF for "resource"), and an optional payload (this will be
# ignored in cases it makes no sense).
def sync_api(creds, type, resource, org, payload = nil)
  api = ""
  if (org)
    api = creds.url_with_org(resource)
  else
    api = creds.url_with_api(resource)
  end

  response = RestClient::Request.new(
    :url => api,
    :method => type,
    :payload => payload,
    :user => creds.login,
    :password => creds.passwd,
    :verify_ssl => false,
    :headers => {:accept => :json,
```

```
                  :content_type => :json}).execute()

  if (response.to_s() != "")
    return(JSON.parse(response))
  else
    return "ok"
  end
end

# Makes an asynchronous API call using a pre-populated Creds object, a
# type (async calls can only be of type :get, but the field is still
# here for consistency with the sync api call, however this value will
# be ignored), the specific call you're making to the API (such as
# "/labels"), whether you want the org specified as part of the full
# URL or not (this will be nil when supplying a full HREF for
# "resource"), and an optional payload (payloads make no sense for async
# GET calls, so this field will be ignored).
def async_api(creds, type, resource, org, payload = nil)
  api = ""
  if (org)
    api = creds.url_with_org(resource)
  else
    api = creds.url_with_api(resource)
  end

  response = RestClient::Request.new(
    :url => api,
    :method => :get,
    :user => creds.login,
    :password => creds.passwd,
    :headers => {"Prefer" => "respond-async",
                 :accept => :json,
                 :content_type => :json}).execute()

  sleep(response.headers[:retry_after].to_i)

  monitor_url = response.headers[:location]
  status = ""
```

```
  while (status != "done" and status != "failed")
    response = sync_api(creds, :get, monitor_url, nil)
    status = response['status']
    sleep 1 unless (status == "done" or status == "failed")
  end

  return (sync_api(creds, :get, response["result"]["href"], nil))
end

# A wrapper around sync_api and async_api (see one of those calls
# above for parameter info). First makes a sync API call, and if it
# was a :get call and the result contains 500 (or more, in case API
# limits are changed) values, makes the same call with the async
# API. Best to call the "right" call yourself from an efficiency of
# execution standpoint, but makes code easier to write and understand
# if you don't mind the (potential) extra load on your PCE.
def call_api(creds, type, resource, org, payload = nil)
  response = sync_api(creds, type, resource, org, payload)
  if (response.length >= 500 and type == :get)
    response = async_api(creds, type, resource, org, payload)
  end
  return(response)
end

# Given a type ("role", "app", "env", or "loc"), a name, and a creds
# object, create a label with the specified name. Does not check to
# see if a label of the specified type and name already exist, and
# fails if this is the case.
def create_label(type, name, creds)
  return(call_api(creds, :post, "/labels", true, {"key" => type, "value"
=> name}.to_json))
end

# Return a hash of hashes containing all labels and their HREFs.
def get_all_labels(creds)
  response = call_api(creds, :get, "/labels", true)

  labels = Hash.new()
  labels['role'] = Hash.new()
```

```
  labels['app'] = Hash.new()
  labels['env'] = Hash.new()
  labels['loc'] = Hash.new()

  response.each do |resp|
    if (resp['key'] == "role")
      labels['role'][resp['value']] = resp['href']
    end

    if (resp['key'] == "app")
      labels['app'][resp['value']] = resp['href']
    end

    if (resp['key'] == "env")
      labels['env'][resp['value']] = resp['href']
    end

    if (resp['key'] == "loc")
      labels['loc'][resp['value']] = resp['href']
    end
  end

  return(labels)
end

# Given the name of a variable, a desired value, and a an array of
# things to search through, return a list of array entries that
# match. Useful for searching through services, IP lists, labels,
# etc. for a desired match. If HREF is true, return a list of HREFs
# instead of a list of full matches. Most commonly used to search for
# matches to "name". Returns an array, as it's legal to have multiple
# items with the same name (unfortunately).
def find_var(var, value, stuff, href = false)
  if href
    return((stuff.select {|thing| thing[var] == value}).map {|thing|
thing["href"]})
  else
    return(stuff.select {|thing| thing[var] == value})
  end
```

```
end

# Create a simple service. Simple, meaning only one port and
# protocol. Services can actually have an arbitrary number of ports
# and protocols (as well as ranges), and can even specify process
# names, but we're keeping this example simple. Note that proto is the
# protocol number, not the name. TCP==6 and UDP==17.
def create_simple_service(name, port, proto, creds)
  return(call_api(creds, :post, "/sec_policy/draft/services", true,
                  {"name" => name,
                   "service_ports" =>
                   [{"port" => port.to_i,
                     "protocol" => proto.to_i}]}.to_json))
end

# Create a simple IP List. Simple, meaning only one address or subnet.
# IP Lists can actually have an arbitrary number of addresses and
# subnets, including exclusions, but we're keeping it simple.
def create_simple_iplist(name, address, subnet, creds)
  addr = address + "/" + subnet
  return(call_api(creds, :post, "/sec_policy/draft/ip_lists", true,
                  {"name" => name,
                   "ip_ranges" =>
                   [{"from_ip" => addr,
                     "exclusion" => false}]}.to_json))
end

# Create a Creds object holding everything we need to know about a PCE.
creds = Creds.new(login, passwd, pce, org_href,port)

# Fetch all of the labels from the PCE.
labels = get_all_labels(creds)

# Fetch all of the services from the PCE.
services = call_api(creds, :get, "/sec_policy/draft/services", true)

# Fetch all of the IP Lists from the PCE.
iplists = call_api(creds, :get, "/sec_policy/draft/ip_lists", true)
```

```
# Make the new labels for our McMurdo service (if needed). Also, store
# the data for the new services we create in the global 'labels' var
# for future reference.
unless labels["app"][ring_app]
  labels["app"][ring_app]=create_label("app", ring_app, creds)["href"]
end
unless labels["env"][ring_env]
  labels["env"][ring_env]=create_label("env", ring_env, creds)["href"]
end
unless labels["loc"][ring_loc]
  labels["loc"][ring_loc]=create_label("loc", ring_loc, creds)["href"]
end

# Create the service if it doesn't already exist. Note that this is
# somewhat dumb, as it will find *any* service with the specified
# name, which isn't necessarily the one you want. Somebody could have
# created another service with the same name.
if (find_var("name", ring_service, services, true).length == 0)
  services.push(create_simple_service(ring_service, ring_port, ring_proto,
creds))
end

# Create the IP List if it doesn't already exist. Note that like the
# service creation above, this is somewhat dumb, as it will find *any*
# IP List with the specified name, which isn't necessarily the one you
# want. Somebody could have created another IP List with the same
# name.
if (find_var("name", ring_allowed_name, iplists, true).length == 0)
  iplists.push(create_simple_iplist(ring_allowed_name,
                                    ring_allowed_net,
                                    ring_allowed_netmask,
                                    creds))
end

# The ruleset hash will hold the new ruleset until we push it to the
# PCE.
ruleset = Hash.new()
```

```
# Set a few basic things in the ruleset.
ruleset["name"] = ring_ruleset
ruleset["enabled"] = true

# Set the scope(s) for the ruleset. Just one scope for this example,
# though there can be more than one.
scope = Array.new()
scope.push({"label"=>{"href"=>labels["app"][ring_app]}})
scope.push({"label"=>{"href"=>labels["loc"][ring_loc]}})
scope.push({"label"=>{"href"=>labels["env"][ring_env]}})
ruleset["scopes"] = [scope]

# The rules array will hold new rules while they're built. Each rule
# is added to the ruleset object as each rule is finished.
rules = Array.new()

# Build up the first rule, then add it to the rules array.
rule = Hash.new()
rule["enabled"] = true
rule["sec_connect"] = true
rule["unscoped_consumers"] = false
rule["providers"] = [{"actors" => "ams"}]
rule["consumers"] = [{"actors" => "ams"}]
rule["service"] = {"href" => find_var("name", "All Services", services,
true)[0]}
rule["ub_service"] = {"href" => find_var("name", "All Services", services,
true)[0]}
rules.push(rule)

# Now build up the second rule, then add it to the rules array, also.
rule = Hash.new()
rule["enabled"] = true
rule["sec_connect"] = false
rule["unscoped_consumers"] = false
rule["providers"] = [{"label" => {"href" => labels['role'][ring_role]}}]
rule["consumers"] = [{"ip_list" => {"href" => find_var("name",
ring_allowed_name, iplists, true)[0]}}]
rule["service"] = {"href" => find_var("name", ring_service, services,
true)[0]}
```

```
rule["ub_service"] = {"href" => find_var("name", "All Services", services,
true)[0]}
rules.push(rule)


# Now add the rules to the ruleset.
ruleset["rules"] = rules


# Now push the whole ruleset to the PCE. If it already exists (or if
# there is any other error), this fails.
response = call_api(creds, :post, "/sec_policy/draft/rule_sets", true,
ruleset.to_json)


# Last, but not least, we need to provision the changes we've
# made. Note that the following will provision *ALL* outstanding
# changes. It's certainly possible that this means provisioning
# changes made by other users who may be logged in at the same time we
# are. Solving this is left as an exercise for the user.
response = call_api(creds, :post, "/sec_policy", true, {"commit_message"
=> "penguins rule"}.to_json)
```