Caldera:
Topics in Chain Mode

David Hunt, Alex Manners, & Andy Applebaum

MITRE

Intros

Alex Manners (@khyberspache)

- Lead Cyber Security Engineer
- Red Teamer

David Hunt (@privateducky)

- Principal Cyber Security Engineer
- Red Teamer

What to expect

- **Fast paced, development focused 4 hour sprint**
  - Limited amount of slides, maximum amount of hands-on-keyboard
- **Need some familiarity with:**
  - Python 3.7+ (development exercises)
  - Linux or Windows
- **New to red teaming and development?**
  - No problem! This is about learning - follow along with David
- **Our goal:**
  - Interest you enough that you decide to contribute new and interesting ideas to our open source project ☺

What we will cover today

- **Caldera Chain-Mode Quick Start**
- **Basic extensions for Caldera**
  - Building abilities
  - Building adversaries
- **Advanced extensions for Caldera**
  - Building your own plug-in
  - Writing a planner *Time permitting
  - Writing an agent *Time permitting

Tentative schedule

- **4 hours total (take your own breaks, we won't be stopping)**
  - 15 mins – Intro slides (this)
  - 15 mins – Lab setup
  - 30 mins – Caldera chain-mode quick start
  - 10 mins – Challenge 1
  - 65 mins – Basic extensions for Caldera
    - 20 mins – Building abilities
    - 10 mins – Challenge 2
    - 20 mins – Building adversaries
    - 15 mins – Challenge 3
  - 95 mins – Advanced extensions for Caldera
    - Building your own plugin (Fact Graph)
    - Writing a planner *Time permitting
    - Writing an agent *Time permitting
  - 10 mins – Questions and closing comments

Part 1: Lab Setup

Lab/Dev Environment

- **We will be working on the local machine**
  - Feel free to use virtual machines
- **Agent and server will run on same system**
  - Concepts remain the same when running agents on remote targets
- **System requirements:**
  - Windows 7/10, Linux (Debian or RedHat based), OS X
  - Python 3.7
  - Sqlite3

Caldera Setup

- **Initial Setup**
  - git clone https://github.com/mitre/caldera.git --recursive
  - python3 –m venv caldera
  - source ./caldera/bin/active
  - pip install -r requirements.txt
  - python3 server.py

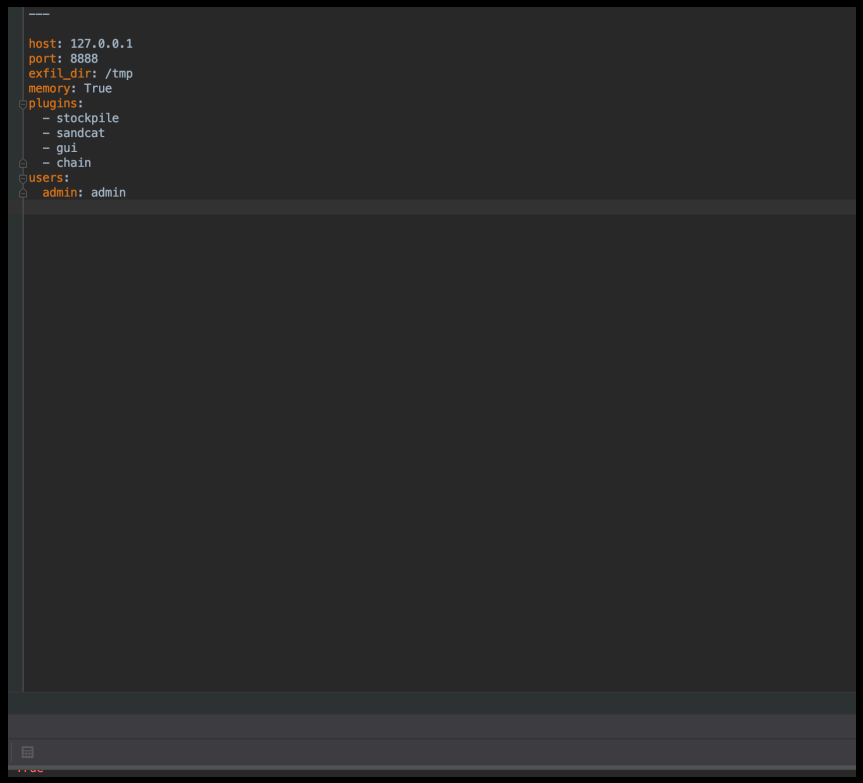
- **Launch your first agent**
  - https://github.com/mitre/caldera/wiki/Plugin:-sandcat

Part 2: Caldera Chain-Mode quickstart

Core Concepts

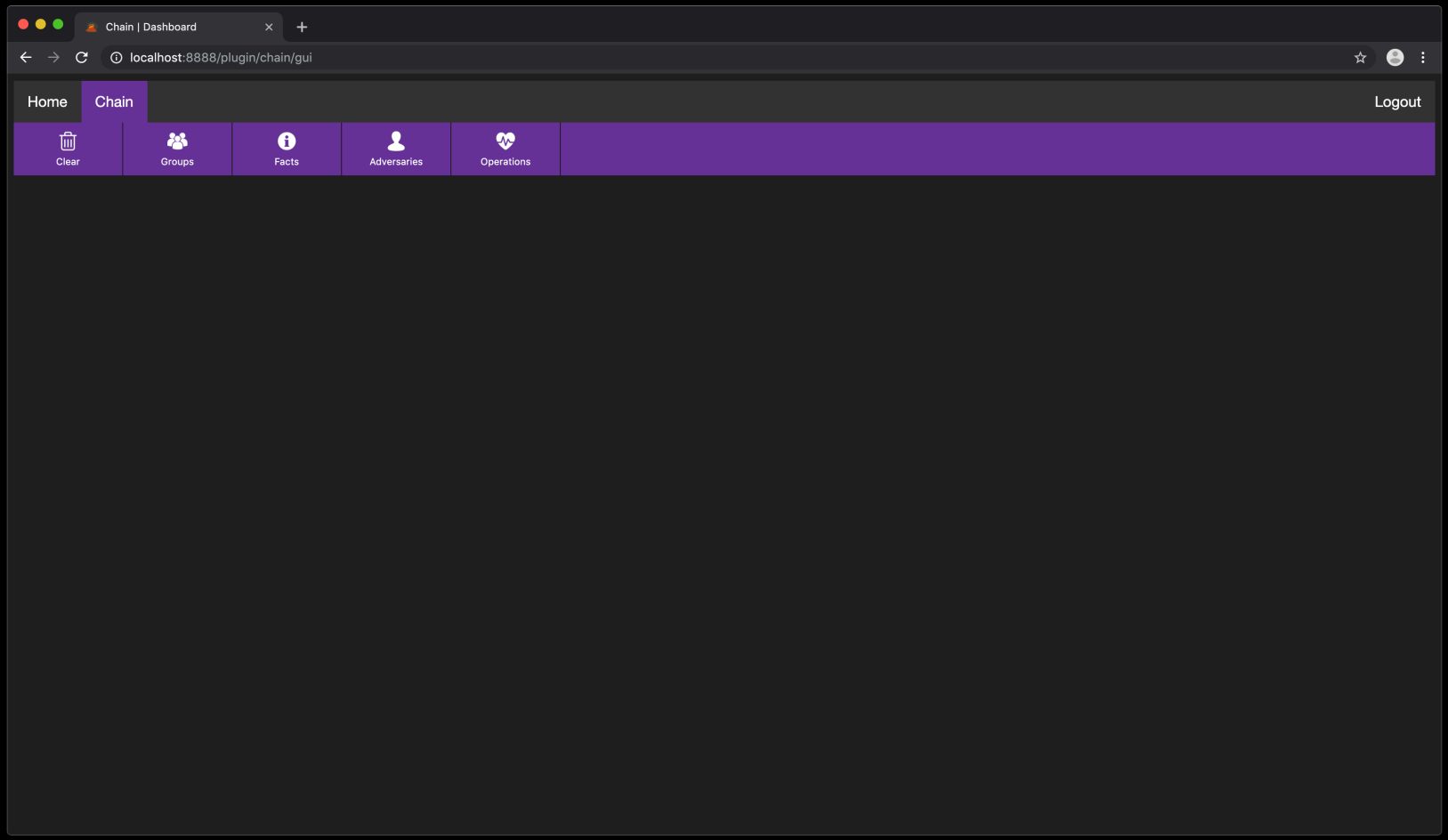
- **Groups**
  - User defined collections of targets that are running agents
- **Facts**
  - Knowledge about the target environment
- **Abilities**
  - Implementations of ATT&CK techniques
- **Adversaries**
  - Collections of abilities that emulate an adversary
- **Operations**
  - A target group + an adversary + facts (optional) = operation

Local.yml – the heart of the C2 server



- **Host**
  - Server address
- **Port**
  - Server port
- **Exfil_dir**
  - Local directory for exfil
- **Memory**
  - Non-persistent sql db
- **Plugins**
  - Selected plugins to run
- **Users**
  - Caldera accounts

Demo Time



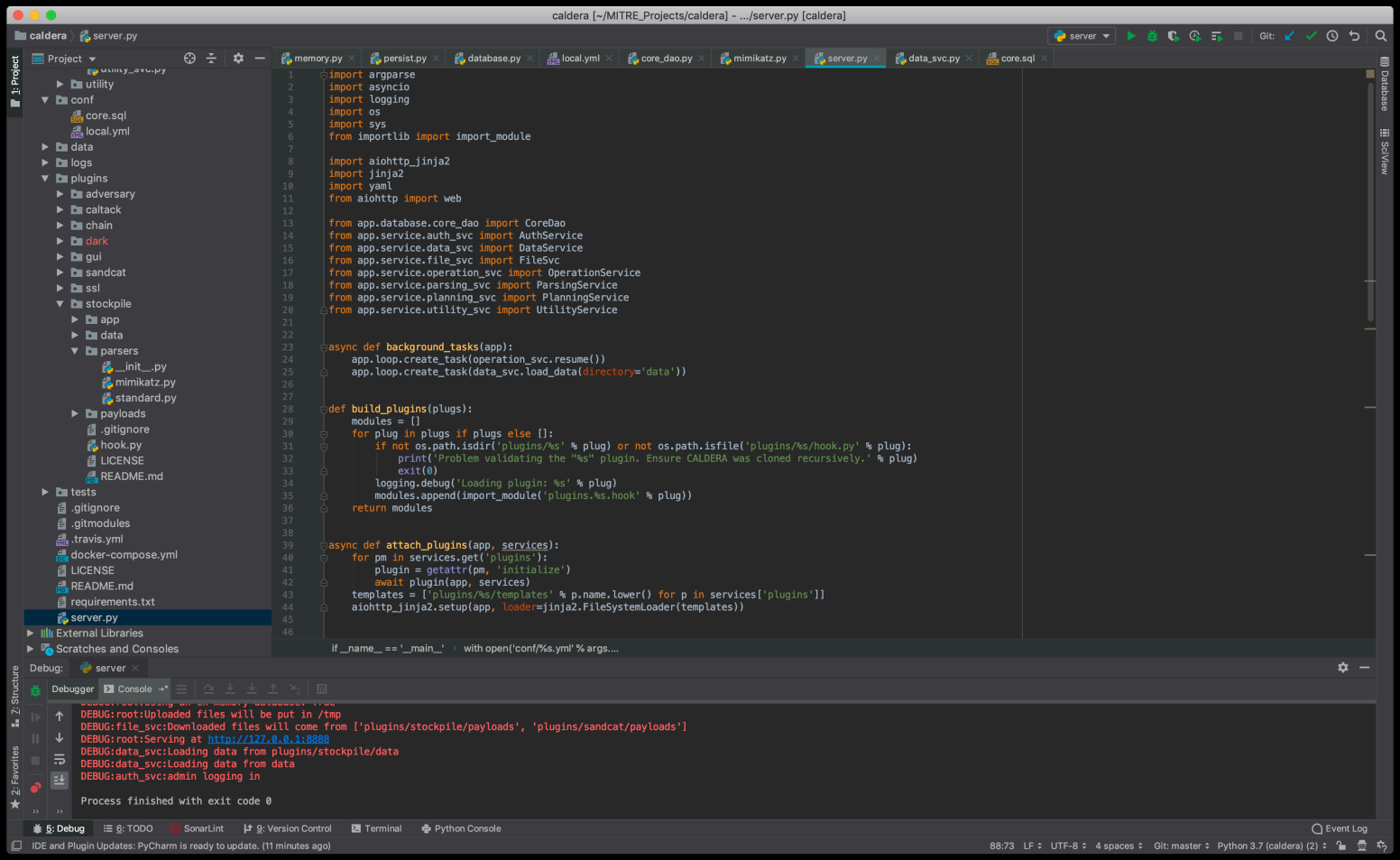
MITRE

Challenge 1

- Get an agent to beacon into our Caldera server (running on David's laptop) from your workstation/VM
- There is a prize!
- Time Limit: 10 minutes

MITRE

Code familiarization



MITRE

Part 3: Basic extensions for Caldera

In this section

- General info
- Building an ability
- Building an adversary

MITRE

General info

- **Most basic extensible components of caldera are yml**
  - New abilities and adversaries can be added with a simple yml file
- **Primarily working in the “stockpile” plugin**
  - Path: stockpile/data/
  - Primary repository for existing and new:
    - Abilities
    - Adversaries
    - Facts (not covered)
    - Planners (covered later)

Part 3-1: Building Abilities

https://github.com/mitre/caldera/wiki/Chain:-Ability

Ability yml

```
---

- id: 0360ede1-3c28-48d3-a6ef-6e98f562c5af
  name: GetComputers (Alice)
  description: Get a list of all computers in a domain
  tactic: discovery
  technique:
    attack_id: T1003
    name: discovery
  executors:
    windows:
      command: |
        Import-Module #{files}\PowerView.ps1 -Verbose -Force;
        Get-NetComputer;
      cleanup: ls
      payload: PowerView.ps1
      parser:
        name: regex
        property: host.host.fqdn
        script: '[\S]+'
```

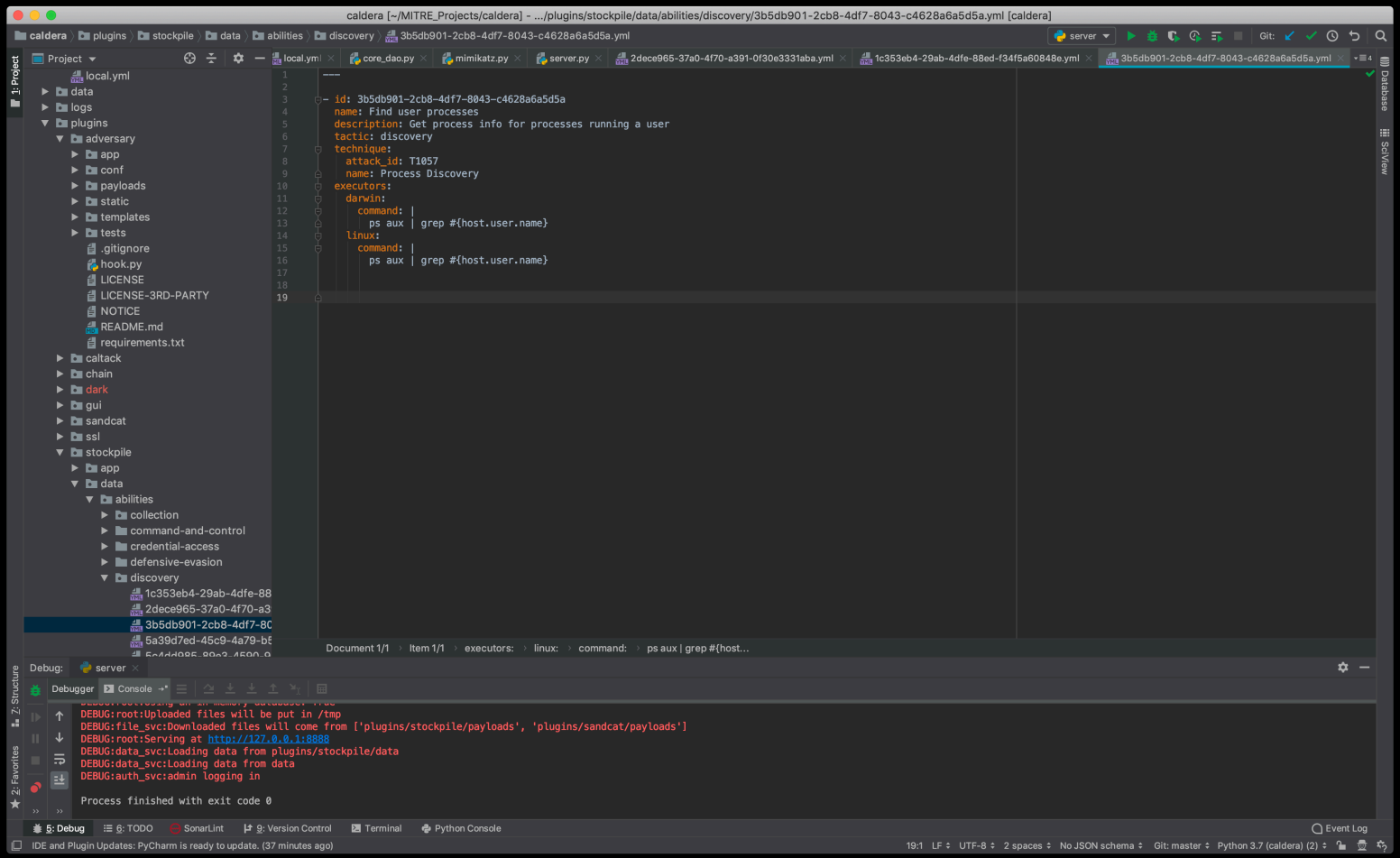
Field descriptors

entry	value
id	a randomly chosen uuid representing this ability
name	the given name for this ability
description	a description for this ability
tactic	the associated ATT&CK tactic for this ability
technique	the associated ATT&CK technique for this ability (both ATT&CK ID and name)
executors	the platforms this ability should run on (this requires both platform and command)

Executor field

entry	value
operating system	Darwin, linux, windows
command	Shell command to execute
cleanup	Shell command to execute during clean up phase (after operation runs)
payload	Various files stored on the caldera server that are required for the shell command to execute
parser	Parse the output of the shell command. Subfields are name, property, and script. Name – parser type (regex, custom, etc) Property – fact generated from the parser Script – regex string/script to execute

Code demo – Adding a new ability



MITRE

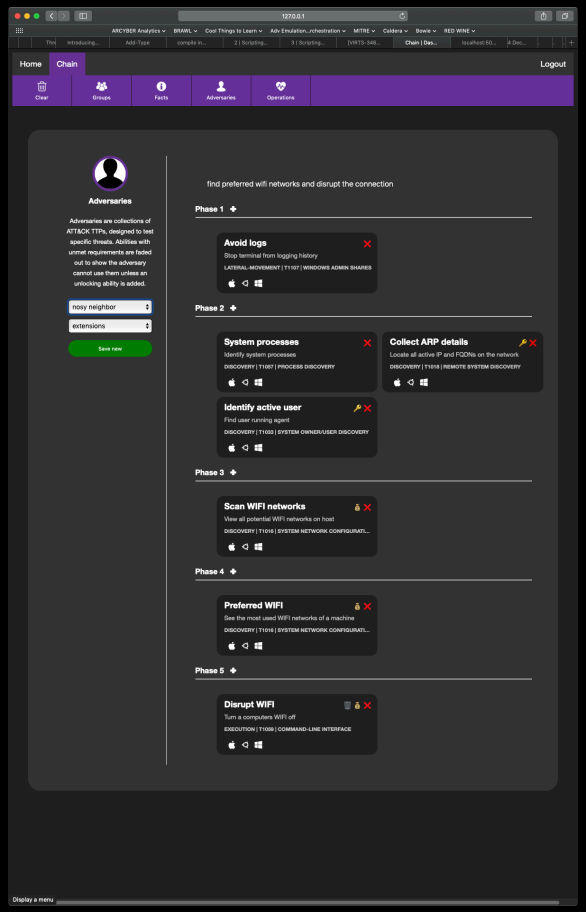
Challenge 2 – Write an ability

- **Your mission, should you choose to accept it:**
  - While planning your red team operation, you note that the target enterprise uses SMB shares for many of its critical business processes. In order for caldera to effectively emulate an adversary, you need to enable SMB share discovery. Write a multiple platform ability to identify SMB shares.
- **Hint: https://attack.mitre.org/techniques/T1135/**
- **No prize for this one**
- **Time limit: 10 mins**

Part 3-2: Building Adversaries

https://github.com/mitre/caldera/wiki/Chain:-Adversary

Adversaries



- **Consist of multiple abilities arranged in phases**
- **Adversaries are a core component of an operation**
- **Build either through the UI or through direct yml file edits**
  - We will focus on yml edits

Adversary yml

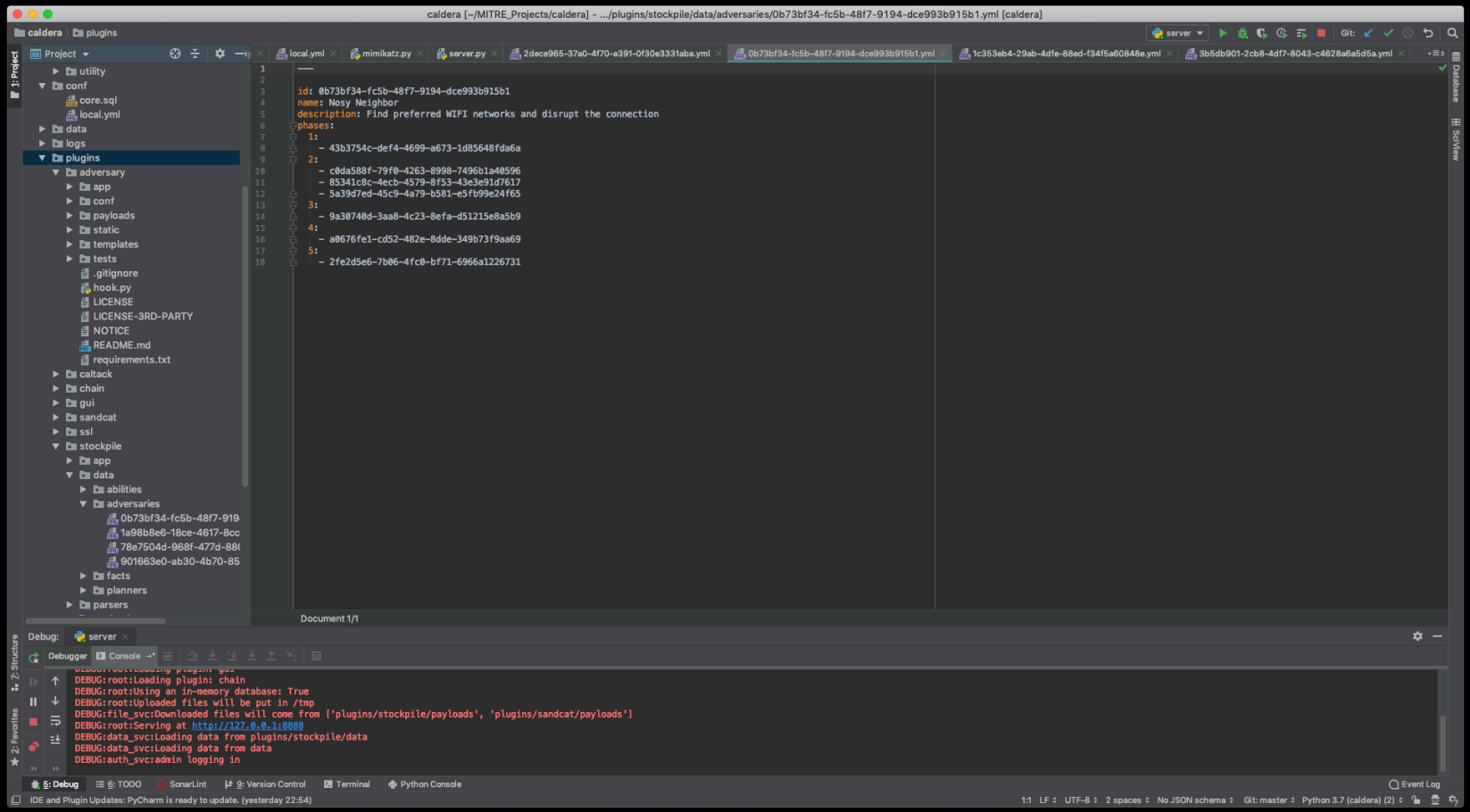
```
---

        id: 0b73bf34-fc5b-48f7-9194-dce993b915b1
        name: Nosy Neighbor
        description: Find preferred WIFI networks and disrupt the connection
        phases:
          1:
            - 43b3754c-def4-4699-a673-1d85648fda6a
          2:
            - c0da588f-79f0-4263-8998-7496b1a40596
            - 85341c8c-4ecb-4579-8f53-43e3e91d7617
            - 5a39d7ed-45c9-4a79-b581-e5fb99e24f65
          3:
            - 9a30740d-3aa8-4c23-8efa-d51215e8a5b9
          4:
            - a0676fe1-cd52-482e-8dde-349b73f9aa69
          5:
            - 2fe2d5e6-7b06-4fc0-bf71-6966a1226731
```

Field descriptors

entry	value
id	a randomly chosen uuid representing this ability
name	the given name for this adversary
description	a description for this adversary
phases	a break down of each ability to run for each operational phase

MITRE

Code demo – Adding a new adversary



MITRE

Challenge 3 – Write an adversary

- **Your mission, should you choose to accept it:**
  - David's Caldera server is also hosting another web service. Using the intelligence gained during challenge #1 (David's IP address), write abilities and an adversary to find that service and collect the name of the file.
  - There are many ways to do this with and without Caldera. Only solutions using Caldera can win a prize though!
- **Hint:**
  - How would you normally approach this problem in a pen test? Try doing it manually, then build the abilities and adversary.
- **There is a prize!**
- **Time Limit: 15 mins**

Part 4: Advanced extensions for Caldera

In this section

- General info
- Building a plugin
- Writing a planner
- Writing an agent

General info

- **Advanced extensibility is via python-based plugins**
  - New plugins can integrate essentially anything you want
  - Core services are passed as objects to plugins for ease of extension
- **Small additions singular probably don't require full plugins**
  - New abilities, new adversary, new planners, etc
- **Plugins are useful for large additions and new features**
  - Adding SSL via Haproxy
  - Multiple new abilities, adversaries, and planners
  - Fact graph – our demo!

Part 4-1: Building your own plugin

https://github.com/mitre/caldera/wiki/How-to-Build:-Plugins

Plugins

- Can be nearly anything, limited only by creativity:
  - New RAT/agent (like 54ndc47)
  - A new GUI
  - New collection of abilities that you want to keep in "closed-source”

- **Plugins are:**
  - Stored in the: “./caldera/plugins/” directory
  - Loaded by adding a ”-plugin_name” to the local.yml plugin section
  - Depends on the “hook.py” which hooks the plugin into Caldera during server load

Basic Structure - hook.py

```
from plugins.chain.app.chain_api import ChainApi

name = 'Chain'
description = 'Adds a REST API for chain mode, along with GUI configuration'
address = '/plugin/chain/gui'


async def initialize(app, services):
    chain_api = ChainApi(services)
    app.router.add_static('/chain', 'plugins/chain/static/', append_version=True)
    app.router.add_route('GET', '/plugin/chain/gui', chain_api.landing)
    app.router.add_route('*', '/plugin/chain/full', chain_api.rest_full)
    app.router.add_route('*', '/plugin/chain/rest', chain_api.rest_api)
```

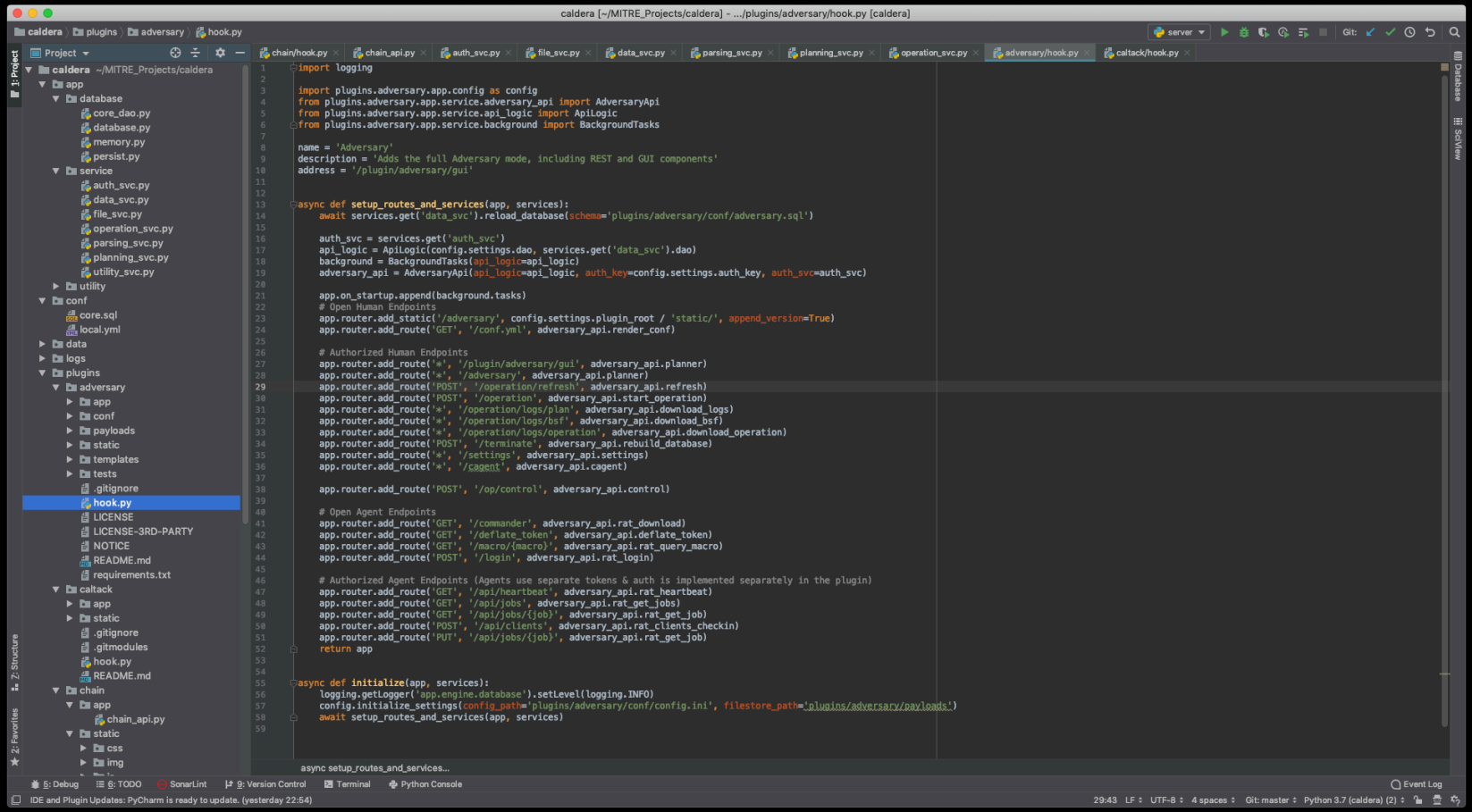
- name => single word for the plugin
- description => phrase explaining the plugin
- address => if NO GUI, None, else URI to browse to GUI

Initialize Function

- Initialize function gets hooked into CALDERA at boot
  - Accepts parameters
    - *app:* the AIOHTTP application object
    - *services:* core service objects Caldera creates at boot time

- **Core services:**
  - *data_svc:* DB service through the core_dao for CRUD operations
  - *utility_svc:* Generic, all-purpose functions (encoding, decoding, etc)
  - *operation_svc:* Logic to start, resume, and stop operations
  - *file_svc:* File-system ops on the Caldera server (upload, download)
  - *auth_svc:* authentication service for creating users, logging in, etc
  - *planning_svc:* Shared logic for planners
  - *parsing_svc:* Logic for parsing text blobs after abilities run
  - *plugins:* List of currently loaded plugins

Example – Chain Plugin



MITRE

Building a Plugin – Fact Graph

- **Your mission, should your choose to accept it:**
  - We want to be able to see the relationships between Caldera facts and abilities (what dependencies exist). Design a plugin in that will collect all of the available abilities and facts, then display it on a D3 relational graph.

- **We will be doing this together, but feel free to figure it out yourself!**
  - There is completed code available:
    - https://github.com/mitre/caldera/tree/bsides-factgraph

Getting started

- **Create a new folder in the plugins directory**
  - factgraph
- **Inside factgraph create the following folders:**
  - app
  - static
    - js
    - css
    - img
  - templates
- **Create new “hook.py” file in that folder**
- **Open “hook.py” in an IDE or editor of your choice**

Building the API Class and Hook file

- **Start with writing our name, description, and address**
- **Define the initialize function**
- **Need to add routes, while keeping code simple**
  - Add a “factgraph_api.py” file to your app/ folder
    - Logic and endpoint definitions will go here
    - Add 2 endpoints, a landing page and a REST endpoint
- **Back in Hook.py, add routes to those endpoints**

hook.py

```python
from plugins.factgraph.app.factgraph_api import FactGraphAPI

name = "FactGraph"
description = "D3 fact and ability relationship graph"
address = "plugins/factgraph/gui"


async def initialize(app, services):
    factgraph = FactGraphAPI(services)
    app.router.add_static('/factgraph', 'plugins/factgraph/static', append_version=True)
    app.router.add_route('GET', '/plugins/factgraph/gui', factgraph.landing)
    app.router.add_route('*', '/plugins/factgraph/rest', factgraph.rest_api)
```

factgraph_api.py

```python
from aiohttp import web
from aiohttp_jinja2 import template


class FactGraphAPI:

    def __init__(self, services):
        self.data_svc = services.get('data_svc')
        self.utility_svc = services.get('utility_svc')

    @template('factgraph.html')
    async def landing(self, request):
        pass

    async def rest_api(self, request):
        data = dict(await request.json())
        index = data.pop('index')
        options = dict(
            POST=dict(
                fact_graph=lambda d: self._build_graph_nodes_links()
            )
        )
        output = await options[request.method][index](data)
        return web.json_response(output)

    """ PRIVATE """

    async def _build_fact_preconditions(self, criteria=None):
        pass

    async def _build_graph_nodes_links(self):
        pass
```

Building a simple front end

- **Go to:**
  - https://github.com/mitre/caldera/tree/bsides-factgraph
  - Download the branch as a Zip file
  - Extract the Zip
    - Copy all of the files from the in ‘plugins/factgraph/static/’ folder of the Zip file to your working ‘factgraph/static/’ directory
    - Copy the template folder as well
    - This is HTML, CSS, JS, and image files that will render your graph on the front-end
- **We will step through this process and briefly look at the code to explain the components**
- **Note: I am not a GUI-guy despite what David might tell you**

Fill out the factgraph_api functions

- **We need to create the logic to render the nodes and links for the graph**
- **2 functions left to build:**
  - _build_fact_preconditions
  - _build_graph_nodes_links

- **Try to build this on your own! Pause here, next slide has answer.**

Completed factgraph_api.py

```python
import re

from aiohttp import web
from aiohttp_jinja2 import template


class FactGraphAPI:

    def __init__(self, services):
        self.data_svc = services.get('data_svc')
        self.utility_svc = services.get('utility_svc')

    @template('factgraph.html')
    async def landing(self, request):
        pass

    async def rest_api(self, request):
        data = dict(await request.json())
        index = data.pop('index')
        options = dict(
            POST=dict(
                fact_graph=lambda d: self._build_graph_nodes_links()
            )
        )
        output = await options[request.method][index](data)
        return web.json_response(output)

    """ PRIVATE """

    async def _build_fact_preconditions(self, criteria=None):
        abilities = await self.data_svc.explode_abilities(criteria=criteria)
        for ab in abilities:
            decoded_fact = self.utility_svc.decode_bytes(ab['test'])
            ab['requires'] = list(set(re.findall('#{([^}]+)}', decoded_fact)))
        return abilities

    async def _build_graph_nodes_links(self):
        abilities = await self._build_fact_preconditions()
        nodes, facts, links = [], [], []
        for ab in abilities:
            fact_found = False
            for p in ab['parser']:
                if not any(p['property'] == f['id'] for f in facts):
                    facts.append(dict(id=p['property'], type='fact', label=p['property']))
                links.append(dict(source=ab['ability_id']+ab['platform'], target=p['property'], label='unlocks'))
                fact_found = True
            for r in ab['requires']:
                if not any(r == f['id'] for f in facts):
                    facts.append(dict(id=r, type='fact', label=r))
                links.append(dict(source=r, target=ab['ability_id']+ab['platform'], label='required'))
                fact_found = True
            if fact_found and not any(ab['ability_id']+ab['platform'] == n['id'] for n in nodes):
                nodes.append(dict(id=ab['ability_id']+ab['platform'], type='ability', label=ab['name'],
                                  platform=ab['platform']))
        return dict(nodes=nodes+facts, links=links)
```


MITRE

Final step – update local.yml

- Lets update the local.yml to include the factgraph plugin and restart the server!
- All goes well, you should have a nice D3 graph of all of the available abilities and facts

Part 4-2: Writing a planner

https://github.com/mitre/caldera/wiki/How-to-Build:-Planners

Planners

- Planner translates the abilities associated with the chosen adversary for an operation onto the current environment in order to complete the operation
  - Planners decide what ability to execute and when

- **Key services:**
  - *planning_svc:* backbone planning service, which implements several planner helper functions
  - *data_svc:* database interaction service, and can be stored to interact with it during operations

- **Every planner requires two functions:**
  - __init__(self, data_svc, planning_svc)
  - execute(self, operation, phase)

Operation Object: execute(self, *operation*, phase)

Field	Value
name	The name of the operation.
host_group	The target host group data for the operation.
adversary	The chosen adversary data for the operation.
jitter	A representation of the variance for executing commands during this operation.
start	A timestamp marking when the operation was started.
finish	A timestamp marking when the operation was completed.
phase	A counter of the current phase of the operation.
cleanup	A boolean value denoting whether or not actions should be cleaned up during this operation.
stealth	A boolean value denoting whether or not stealth should be used during this operation.
planner	The planner this operation should utilize.

Planners:
Produce Links in the core_chain table of DB

Field	Value
op_id	The operation's ID number, available in the operation database object.
host_id	The id of the target agent.
ability	The id of the target ability to execute, selected from the list linked in the operation database object.
jitter	A representation of how long to have the agent wait before checking back in after executing this command.
command	The command to execute (base64 encoded).
cleanup	Any cleanup commands to execute (base64 encoded).
score	The 'worth' of the chosen action.
decide	A timestamp of when this decision was made.
collect	A timestamp of when data from running this action retrieved. *(filled in by agents)*
finish	A timestamp of when the link was finalized. *(filled in by operation loop)*

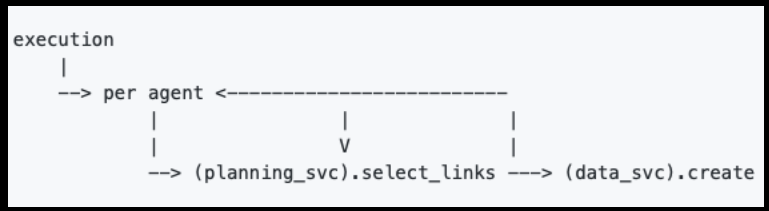
Example – Sequential (Logical) Planner

```
class LogicalPlanner:

    def __init__(self, data_svc, planning_svc):
        self.data_svc = data_svc
        self.planning_svc = planning_svc

    async def execute(self, operation, phase):
        for member in operation['host_group']:
            for l in await self.planning_svc.select_links(operation, member, phase):
                l.pop('rewards', [])
                await self.data_svc.create_link(l)
        await self.planning_svc.wait_for_phase(operation)
```


MITRE

Logical Planner Flow

- 'execution' method iterates through the total collection of available agents
- For each agent, the planning service (**planning_svc**) function **select_links** is called, which returns a list of valid links
- This list is broken down to be submitted to the **create** function within the data service (**data_svc**), which submits the action to the agents associated the operation
- Planners can have radically different flows, provided the starting point of 'execution' and terminating point of saving each link (**create**) exist



Planning Service

- **More complex than the actual planner**
- **Meat of it is the “select_links” function**
- **For a deep dive into this, refer to our GitHub wiki**
  - https://github.com/mitre/caldera/wiki/How-to-Build:-Planners

Writing a planner

- **Your mission, should your choose to accept it:**
  - The logical planner no longer fully meets your needs. Design and write new planner of your choosing.

- **Sky is the limit – work in groups to come up with potential ideas!**

Part 4-3: Writing an agent

https://github.com/mitre/caldera/wiki/How-to-Build:-Agents

Agents

- **Lynch-pin of Caldera**
  - Need agents to be on endpoints in order to execute operations!
- **Primary Agent: Sandcat Plugin (Gocat)**
  - Simple GOLang agent that executes commands passed from Caldera server in the hosts native shell (psh, bash, etc)
- **Writing agents:**
  - Plug into existing HTTP API endpoints
  - Create your own totally unique agent and C2 plugin
- **Sandcat plugin – Everything you need to BYoA**
  - https://github.com/mitre/sandcat/blob/master/app/sand_api.py
  - https://github.com/mitre/sandcat/blob/master/app/sand_svc.py

Agents - Basic knowledge (API & core_agent DB fields)

Endpoint	Use
/sand/beacon	This endpoint handles registration and check-ins (can return actions).
/sand/results	This endpoint handles submission of the results of completed actions.

Field	Value
id	Agent id. Automatically populated when an agent first registers.
paw	Representation of who this agent is. Default is Hostname$User.
checks	Count of how many times the agent has checked-in. It should be incremented at each beacon.
last_seen	Timestamp of the most recent beacon check-in.
platform	Platform the agent is executing on, such as windows or linux.
server	IP address used by the agent when contacting the server. This may be masked in some environments.
files	Location of a folder that the agent can safely drop files into, such as /tmp.

Agent - Registration

- **Agent beacons to /sand/beacon endpoint**
- **Beacon is base64 encoded JSON object**
- **X-PAW in HTTP POST header**

Field	Value
paw	Representation of who this agent is. The usual form for this is Hostname$User.
platform	Current platform/OS of the box the agent is operating on.
files	Location of a folder that the agent can safely drop files into, such as /tmp.

Agent - Liveliness

- **Same general structure as initial registration beacon**
- **Periodic beacons to C2 as liveliness check**

Field	Value
paw	Representation of who this agent is. The usual form for this is Hostname$User.
platform	Current platform/OS of the box the agent is operating on.

Agent - Actions

- **Agent beacons to /sand/beacon endpoint**
- **Beacon responds with JSON list object of actions to execute**

Field	Value
id	Represents specific action is being requested. This information is really only needed by the server, but is required as part of the response.
sleep	Field that specifies how long the agent should wait before checking back in.
command	Base64 encoded block containing the actual command(s) to execute.
cleanup	Field that could contain a cleanup command to run, encoded in base64. This field is optional.
payload	Name of any linked payloads for the command, should they be necessary.

Agent – Output/Response

- **If initial beacon has actions to execute, step through those actions**
- **Capture stdout/stderr**
- **POST output JSON object (Base64 encoded) up to /sand/results API endpoint**

Field	Value
link_id	Number referencing which specific action this is (used by the server for tracking purposes). This information can be acquired from the *id* value provided in the original request.
paw	Representation of who this agent is. It needs to match the one used during registration.
output	Base64 encoded text string containing the output of the executed action.
status	Field contains the closing status of the action executed (typically a 0 or a 1).

Writing an agent

- **Your mission, should your choose to accept it:**
  - The gocat agent binary is getting caught (Oh No!). Write a new agent in a language of your choice. Either make that agent interact with the Sandcat API endpoints, or design your own C2 channel!
- **Sky is the limit – work in groups to come up with potential ideas!**

Questions and closing comments