From Zero to Hero

How to create a custom Mythic agent

Workshop Instructors

- Cody Thomas (**@its_a_feature_**)
  - Created Mythic framework (https://github.com/its-a-feature/Mythic)
  - Focuses on macOS research and operations
  - Created the initial macOS ATT&CK page
  - Spoke at conferences:
    - Objective by the Sea (x2)
    - SANS Threat Hunting Summit
    - HOPE
    - Blackhat Arsenal
    - BSides Seattle
    - X33fcon
  - Love-hate relationship with JXA

Workshop Instructors

- Josiah Massari (**@Airzero24**)
  - Operator at SpecterOps
  - Wannabe C# and Swift dev
    - Atlas
    - Venator-Swift
    - https://github.com/airzero24
  - USAF tech guy
    - I fixed printers
  - Washed up bar band guitarist and indie record label owner

Workshop Goals

- Goal of the workshop is to illustrate how to create a quick, custom agent that works within the Mythic framework
  - You should walk away with the basic understanding of how to make your own agent
- Caveats:
  - We aren't covering every possible feature
  - We're not focusing on evasion
  - We're simply using PowerShell to demonstrate components for rapid development

Workshop Requirements

- Mythic Instance on Ubuntu 18.04+ (root access)
  - Installed and running
- Windows VM with PowerShell v5
  - Some form of code editor (ISE, Visual Studio Code, etc)
- Each VM with 20GB HDD, 3+GB RAM, 2vCPU
- Connectivity between Mythic and Windows VM
- Agent and Lab code from https://github.com/MythicAgents/hercules
- VMs can be local or hosted in the Cloud
  - AWS, GCP, Azure, etc

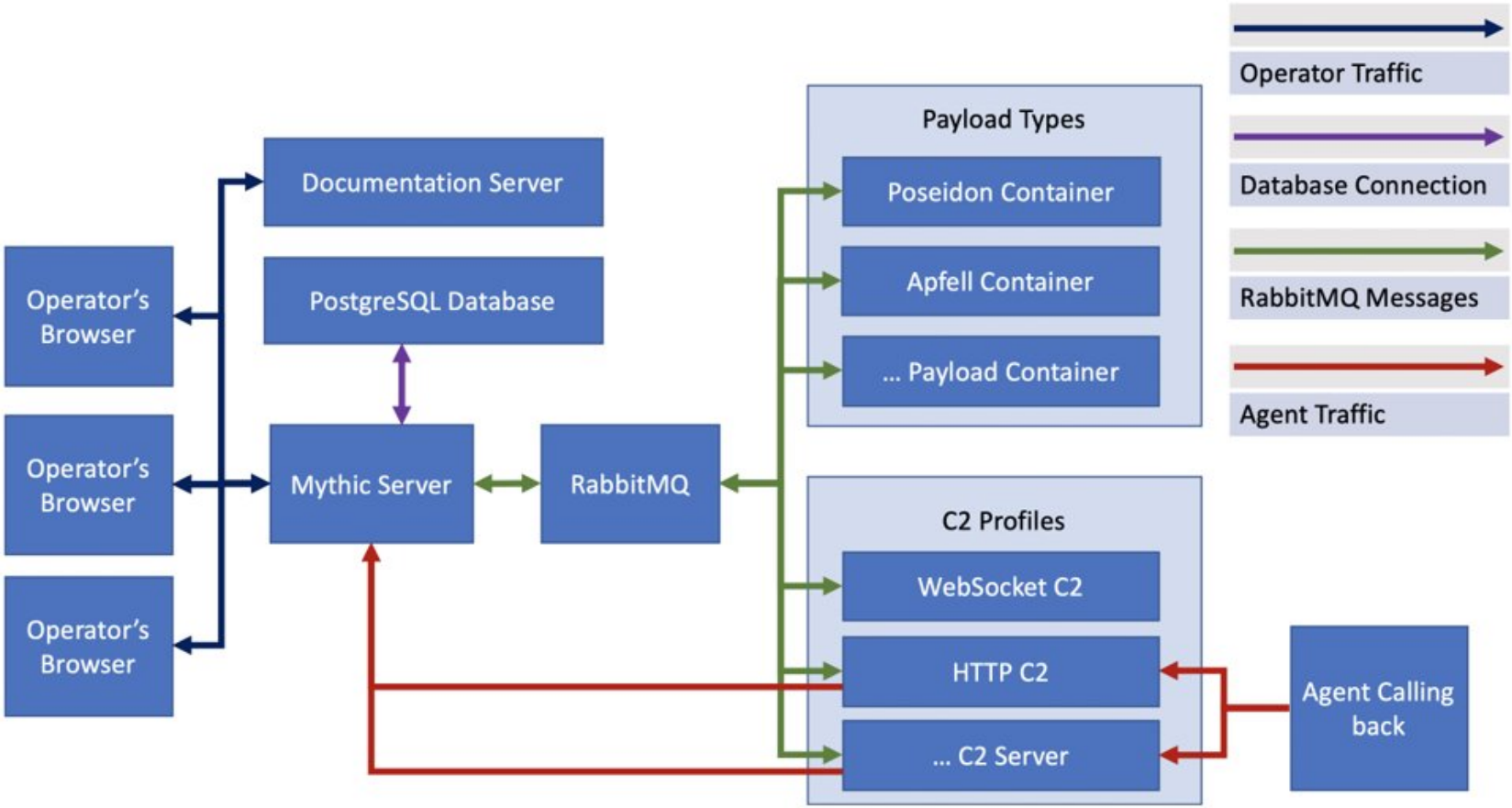
Overview

- What is Mythic?
- Agent requirements and design considerations
- Hercules overview
- Payload Container
- C2 Profiles
- Building Agent Commands
- Using Mythic hooking features

Mythic

- Open Source at https://github.com/its-a-feature/Mythic
- Documentation at https://docs.mythic-c2.net/
- YouTube series: https://www.youtube.com/playlist?list=PLHVFedjbv6sNLB1QqnGJxRBMukPRGYa-H
- Uses Docker to separate out all components
- Each Payload Type
- Each C2 Profile
- Database
- RabbitMQ
- Web Server
- Agent/C2 Specific Documentation
- Operators simply connect via a browser

Mythic Workflow



Why make a Mythic agent?

- Mythic is designed for modern red teams
  - Operators have their own login, preferences, roles
  - Operations create logical groupings of payloads/tasks
    - Lock down operations to specific operators
    - Block certain commands for certain operators in an operation
    - Spectator-views where you can't create/edit/delete anything
  - Techniques mapped to MITRE ATT&CK
  - Scriptable front-end UI components for tasking
  - Filter your UI to only see certain tasks

Why make a Mythic agent?

- Mythic is designed for modern red teams
  - Quality of life improvements like adding comments to tasks, searching across an operation, tracking stats
  - File Browser support
  - SOCKS support
  - Unified and server-side process-listing and file browser aggregation
  - Eventing system to chat with operators or send warnings that need to be addressed
  - Detailed Artifact tracking for deconflictions

Agent Requirements

- Payload Container
  - Local container managed through Mythic
  - OR external container/VM that connects back to Mythic
- Builder.py
  - Dual purpose as agent “information” file and script to build new agents
- Agent code
- Agent functions
  - These are the agent commands

Design Considerations

- What OS(s) will your agent support?
  - Windows, macOS, *nix, Android, etc.
- What languages are supported by this OS?
  - C/C++, C#, obj-C, GoLang, Swift, PowerShell, etc.
- What language will meet your operational needs?
  - Low level control – C/C++, obj-C, GoLang
  - Quick development – C#/Swift
  - “Fileless” – PowerShell, JXA

Design Considerations (cont'd)

- Does the language require complication?
  - If so, can this be accomplished in a docker container or will an external container/VM be required?
- What C2 profiles will the agent support?
  - Does the chosen language easily support Mythic C2 profiles? **JSON*
  - Or will it require a new custom C2 profile?
- Will the agent support dynamic command loading?
  - Will greatly depend on the language being used
- Will the agent support any additional hooking features?
  - Depends on the operational use case of the agent

Hercules Overview

- Focus on simplicity and ease of use
- OS Support: Windows
  - Targeting Windows 10 x64
- Language: PowerShell (v5)
- Operational Needs:
  - Lightweight and easy to modify
  - Ability to execute PowerShell commands
  - Capability to dynamically load and execute new commands

Hercules Overview (cont'd)

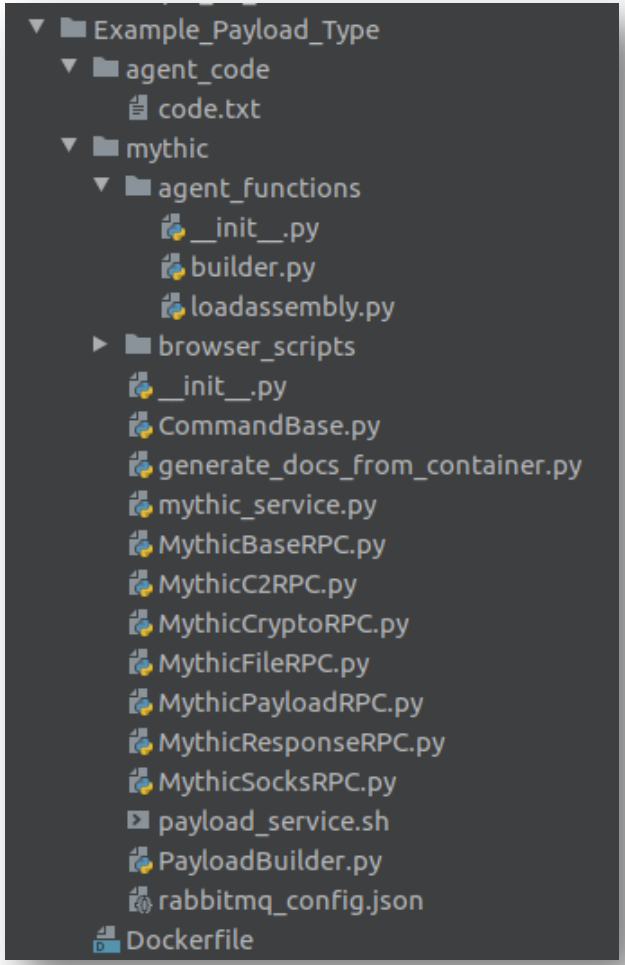
- Compilation not required
  - Can easily integrate into Mythic
- C2 Profiles: HTTP
  - Base C2 Profile for simplicity
  - More on this later
- Dynamic loading: Yes
  - Load commands in new PowerShell instance to execute
- Hooking: Limited
  - Due to time constraints (File browser, SOCKS, etc.)

Payload Type Containers

- Docker container for managing payload creation and agent commands
- Can be located on the Mythic server or an external container/VM if required
- Container manages all browser scripts for agent
  - Dynamically customize agent output
- Start with **Mythic/Example_Payload_Type**
  - Pre-configured basic Payload Type Container and file/folder structure
  - Always stay up-to-date with this folder

Payload Type Folder Structure

- AGENT_NAME
  - agent_code
    - All files used for building agent
  - mythic
    - agent_functions
      - Builder.py – required for every agent
      - Files for agent commands
    - browser_scripts
      - Any unique browser scripts for the agent to use
    - Various files for interacting with Mythic server
  - Dockerfile



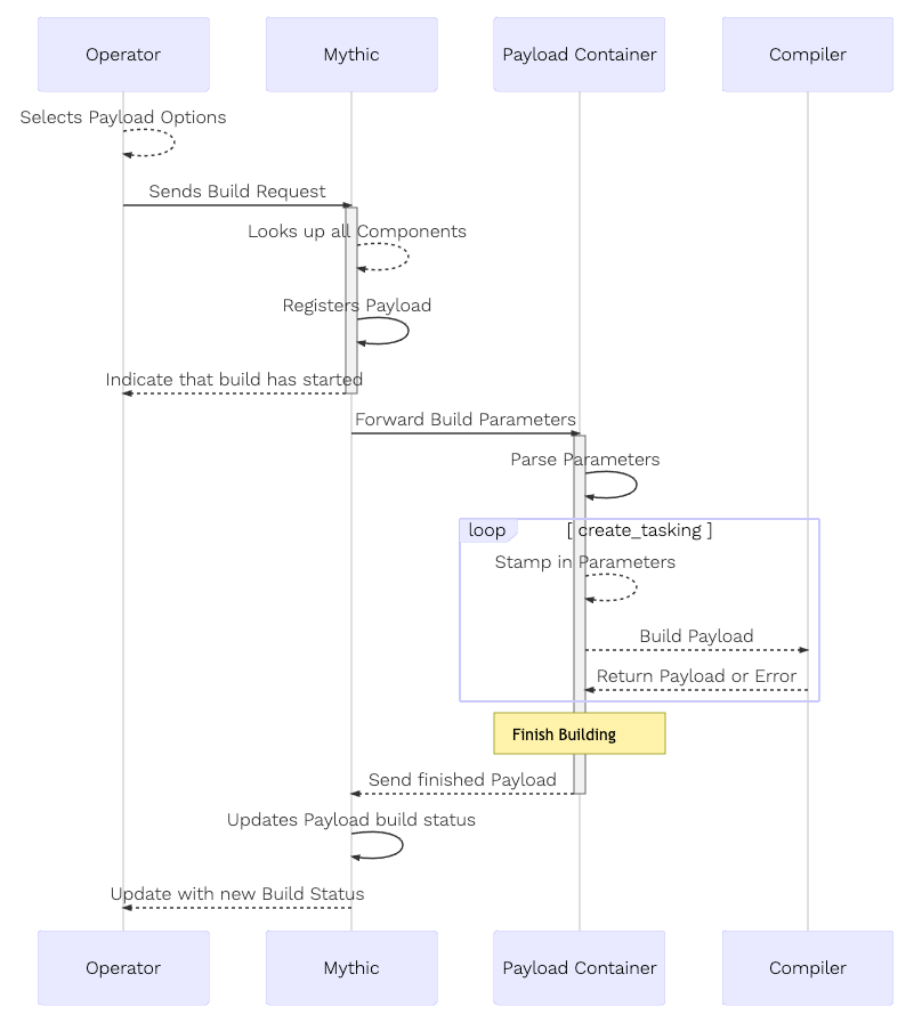
Payload Build File

```python
class MyNewAgent(PayloadType):

    name = "hercules"
    file_extension = "ps1"
    author = "@Airzero24"
    supported_os = [
        SupportedOS.Windows
    ]
    wrapper = False
    wrapped_payloads = []
    note = "This payload uses PowerShell to create a simple agent for demonstration purposes"
    supports_dynamic_loading = True
    build_parameters = {
        "output": BuildParameter(
            name="output",
            parameter_type=BuildParameterType.ChooseOne,
            description="Choose output format",
            choices=["ps1", "base64"],
        )
    }
    c2_profiles = ["HTTP"]

    async def build(self) -> BuildResponse:
        resp = BuildResponse(status=BuildStatus.Success)
        return resp
```

Payload Build Workflow



Demo 1
Containers

Agent Comms and C2 Profiles

- Mythic agents use different transports to govern communication with a Mythic server called C2 Profiles
- Profiles dictate the manner in which messages are received and sent to the server
- Types of transports available
  - HTTP/S
  - Websockets
- If you need to use another transport, create your own profile!

Mythic Message Structure

b446b886-ab97-49b2-b240-969a75393c06 eyJhY3Rpb24iOiAiZ2V0X3Rhc2tpbmciLCJ0YXNraW5nX3NpemUiOiAxfQ==

Payload/callback UUID

Base64 JSON Data Message

Requesting/Sending Messages

- Messages to Mythic are contained within the previously mentioned JSON data blob
- This data will have different values based on what action we are trying to perform
  - These are called Hooks withing Mythic (more later)
- For shuttling messages between our agent and the Mythic server, we need to use three hooks
  - Checkin
  - Get_tasking
  - Post_response

Checkin Data (Agent -> Mythic)

```
"action": "checkin", // required
"ip": "127.0.0.1", // internal ip address - required
"os": "Windows 10", // os version - required
"user": "airzero", // username of current user - required
"host": "WIN10-DEV", // hostname of the computer - required
"pid": 4444, // pid of the current process - required
"uuid": "payload uuid", //uuid of the payload - required

"architecture": "x64", // platform arch - optional
"domain": "shire.local", // domain of the host - optional
"integrity_level": 3, // integrity level of the process - optional
"external_ip": "8.8.8.8", // external ip if known - optional
"encryption_key": "base64 of key", // encryption key - optional
"decryption_key": "base64 of key", // decryption key - optional
```

Checkin Data (Mythic -> Agent)

```
{
    "action": "checkin",
    "id": "UUID", // new UUID for the agent to use
    "status": "success"
}
```

- At this point, a callback is registered in the UI on the "Active Callbacks" Page
- Every callback gets its own random UUID
  - This identifies a callback vs a payload

HTTP C2 Profile

- Mythic's "standard" HTTP/S profile
- Uses GET/POST requests to send messages between server and agent
- Mythic looks for data stored in first query parameter, cookie value, header value, or message body depending on request type
- Offers "some" level of customization

HTTP C2 Profile Within Hercules

- All request will use the POST method with data stored in the message body
  - This is for simplicity, could easily use GET or a combination of requests
- The Net.WebClient .NET class will be used to create web requests
  - Again for simplicity, there are many other options available
- Will use all build features of the C2 profile
  - Custom User-Agent, Host header, etc.

Demo 2
Base Agent

Mythic Tasking

- Agents specify how many tasks they want
- Can forward along "delegate" messages for agents linked via P2P protocols
- Always returns JSON with list of tasks
  - Every task contains:
    - Command
    - String of parameters
    - Timestamp
    - Task ID (random UUID)
  - Optionally get back delegate messages or SOCKS data

Get_tasking (Agent -> Mythic)

```
{
    "action": "get_tasking",
    "tasking_size": 1, //indicate the maximum number of tasks you want back
    //if passing on messages for other agents, include the following
    "delegates": [
        {"UUID1": agentMessage},
        {"UUID2": agentMessage}
    ]
}
```

Get_tasking (Mythic -> Agent)

```
{
    "action": "get_tasking",
    "tasks": [
        {
            "command": "command name",
            "parameters": "command param string",
            "timestamp": 1578706611.324671, //timestamp provided to help with ordering
            "id": "task uuid",
        }
    ],
    //if we were passing messages on behalf of other agents
    "delegates": [
        {"UUID1": agentMessage},
        {"UUID2": agentMessage}
    ]
}
```

Demo 3
Getting Tasking

Mythic Tasking - Responses

- Generic format for message responses in JSON
- Plug-n-play keys for hooking features
  - Displaying user output
  - Indicating task status (completed/error)
  - Reporting back file upload/downloads
  - Keylogging, artifacts, file listing, etc

Post_response (Agent -> Mythic)

```
{
    "action": "post_response",
    "responses": [
        {
            "task_id": "uuid of task",
            ... response message (see below)
        },
        {
            "task_id": "uuid of task",
            ... response message (see below)
        }
    ], //if we were passing messages on behalf of other agents
    "delegates": [
        {"UUID1": agentMessage},
        {"UUID2": agentMessage}
    ]
}
```

Hook Example Post Response

```
{
    "task_id": "task uuid here",
    "user_output": "some user output here",
    "artifacts": [
        {
            "base_artifact": "Process Create",
            "artifact": "sh -c whoami"
        },
        {
            "base_artifact": "File Write",
            "artifact": "/users/itsafeature/Desktop/notmalware.exe"
        }
    ]
}
```

Post_response (Mythic -> Agent)

```
{
    "action": "post_response",
    "responses": [
        {
            "task_id": UUID,
            "status": "success" or "error",
            "error": 'error message if it exists'
        }
    ],
    //if we were passing messages on behalf of other agents
    "delegates": [
        {"UUID1": agentMessage},
        {"UUID2": agentMessage}
    ]
}
```

Demo 4
Posting Responses

Processing Tasks (agent specific)

- Hercules will use custom PowerShell objects to easily track Mythic taskings
- Tasks executed asynchronously via separate PowerShell instances
  - Tasks are tracked and killable
  - This allows long-running tasks to not lock up the main agent loop

Demo 5
Process Tasking

Agent Commands

- Commands are defined in files with the Payload Type's **mythic/agent_functions** directory
- These are python files containing two classes:
  - Arguments
    - Defines a command's arguments, default values, parameter types, validation functions, etc
  - Command
    - Defines any additional processing steps to take on a tasking argument such as: registering a file with mythic, creating artifacts, building new payloads, encrypting messages, etc

Agent Commands (cont'd)

- Why have additional command processing?
  - Allows greater flexibility when customizing commands per payload type
  - Opportunities for integrating third party applications into agent workflows
  - Can hook into CI/CD pipelines
  - Can create custom DLLs/Binaries/ShellCode etc per task

Tasking File

```python
class SleepCommand(CommandBase):
    cmd = "sleep"
    needs_admin = False
    help_cmd = "sleep [interval] [jitter]"
    description = "Modify the time between callbacks in seconds."
    version = 1
    is_exit = False
    is_file_browse = False
    is_process_list = False
    is_download_file = False
    is_remove_file = False
    is_upload_file = False
    author = "@its_a_feature_"
    attackmapping = ["T1029"]
    argument_class = SleepArguments

    async def create_tasking(self, task: MythicTask) -> MythicTask:
        return task

    async def process_response(self, response: AgentResponse):
        pass
```

Tasking - RPC

- Tasking functions can call RPC endpoints
  - Register files, search for files
  - Build new payloads
  - Start/stop Socks
  - Create artifacts
  - Send user output
  - Getting payload details
  - Update callback information
  - And more...

Tasking – RPC Example

```python
async def create_tasking(self, task: MythicTask) -> MythicTask:
    gen_resp = await MythicPayloadRPC(task).build_payload_from_template(
        task.args.get_arg("template")
    )
    if gen_resp.status == MythicStatus.Success:
        # we know a payload is building, now we want it
        while True:
            resp = await MythicPayloadRPC(task).get_payload_by_uuid(gen_resp.uuid)
            if resp.status == MythicStatus.Success:
                if resp.build_phase == "success":
                    # it's done, so we can register a file for it
                    task.args.add_arg("template", resp.agent_file_id)
                    break
                elif resp.build_phase == "error":
                    raise Exception(
                        "Failed to build new payload: " + resp.error_message
                    )
                else:
                    await asyncio.sleep(1)
    return task
```

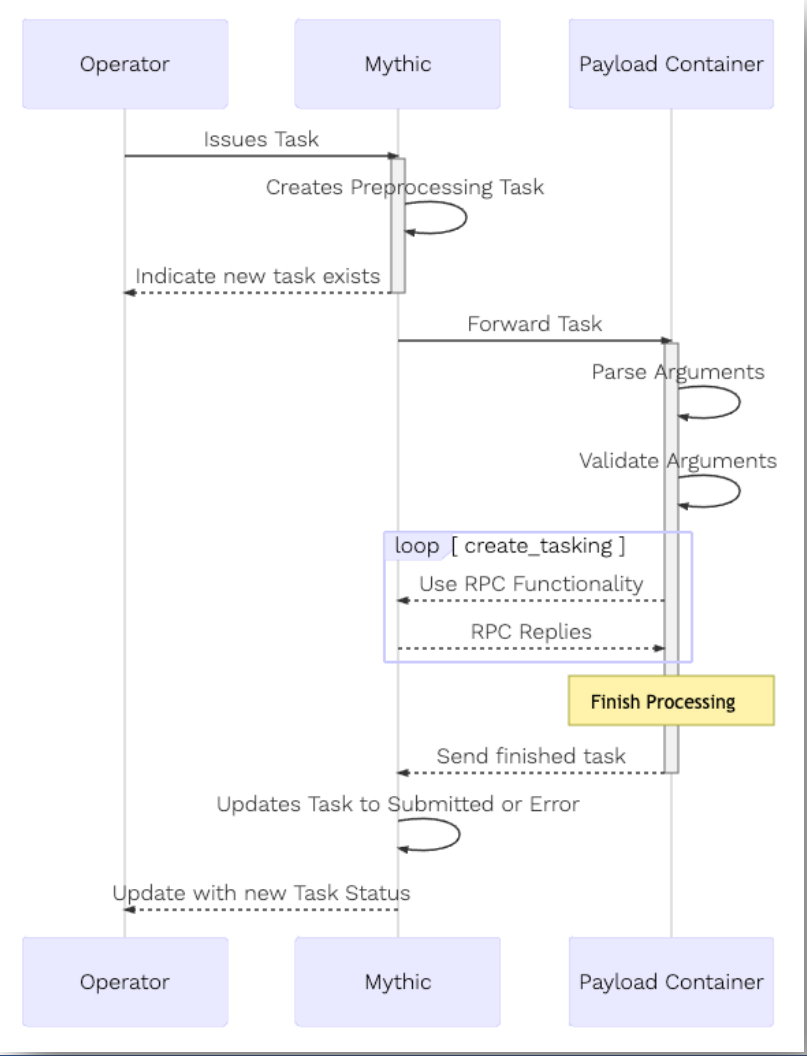
Tasking Arguments

```python
def positiveTime(val):
    if val < 0:
        raise ValueError("Value must be positive")


class SleepArguments(TaskArguments):
    def __init__(self, command_line):
        super().__init__(command_line)
        self.args = {
            "jitter": CommandParameter(
                name="jitter",
                type=ParameterType.Number,
                validation_func=positiveTime,
                required=False,
                description="Percentage of C2's interval to use as jitter",
            ),
            "interval": CommandParameter(
                name="interval",
                type=ParameterType.Number,
                required=False,
                validation_func=positiveTime,
                description="Number of seconds between checkins",
            ),
        }

    async def parse_arguments(self):
        if self.command_line[0] != "{":
            pieces = self.command_line.split(" ")
            if len(pieces) == 1:
                self.add_arg("interval", pieces[0])
            elif len(pieces) == 2:
                self.add_arg("interval", pieces[0])
                self.add_arg("jitter", pieces[1])
            else:
                raise Exception("Wrong number of parameters, should be 1 or 2")
        else:
            self.load_args_from_json_string(self.command_line)
```

Tasking Workflow



Demo 6
Building Full Hercules

Recap – What did we cover?

- The role of Payload Type containers and how to make them
- What it takes to build a payload and issue tasking
- How to transform user tasking to hook into Mythic features
- The role of C2 Profile containers

Additional Features

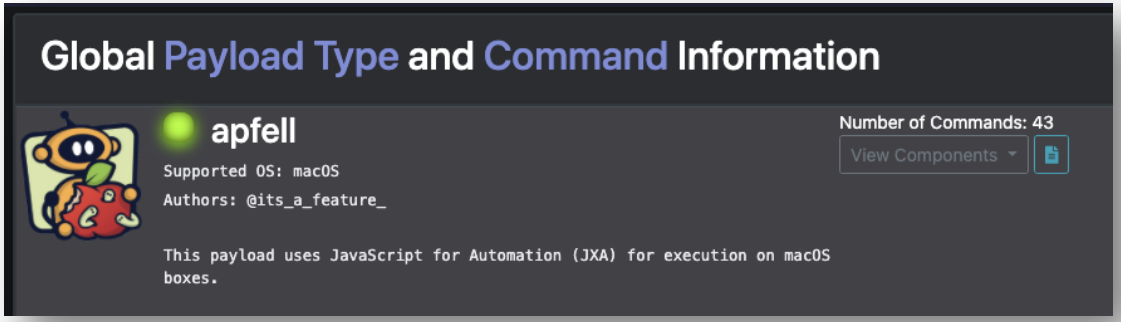
- We wanted to highlight some other major features for agents and Mythic
  - These aren't required to get an agent running, but are extremely helpful for quality of life and collaboration

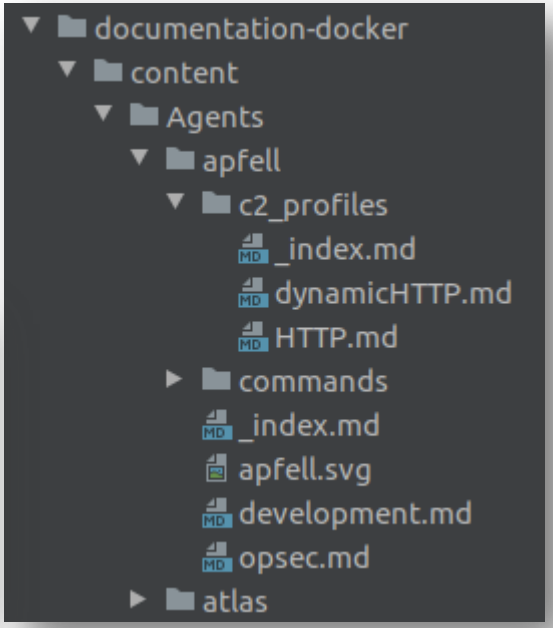
Agent Documentation

- An agent is only as good as the documentation supporting it
  - Poor documentation means nobody can use it
- Mythic has a Hugo documentation server
  - You can write OPSEC, command, C2, development, and general information in Markdown
  - This is auto rendered in the documentation website locally
- This allows you to bundle up agent-specific documentation **with** the agent
- **Example_Payload_Type** repo has a "generate_docs_from_container" python script to help with this

Agent Documentation

- All standard agent documentation lives in "Agents" folder
- Clicking blue "document" icon opens hugo documentation webserver





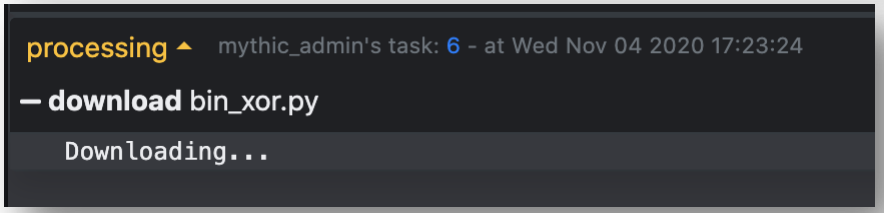
Browser Scripts

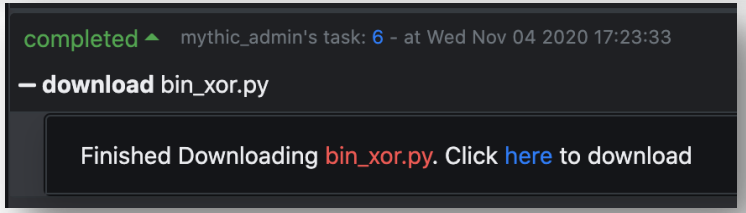
- Historically
  - Shove all data into a string and write regex or string parsers to try to find data that operators want
  - Error prone, not scalable, highly specific to one tool or command
- Now
  - Structured Input/Output allows for easy, precise, scalable processing
  - Script via JavaScript in the browser and output custom HTML
    - Very dynamic, lots of UI customizability options (DOM is your oyster)
  - You can have personal browser scripts or the admin for an operation can enforce theirs across everybody for consistency

Browser Scripts

- Scripts get full context of the task and all responses

```
function(task, responses){
  if(task.completed === true && task.status !== 'error'){
    try{
        let status = JSON.parse(responses[0]['response']);
        if(status.hasOwnProperty('agent_file_id')){
            let file_name = status['filename'];
            return "<div class='card'><div class='card-header border border-dark shadow'>Finished Downloading <span class='display'>"
                + escapeHTML(file_name) + "</span>. Click <a href='/api/v1.4/files/download/" + status['agent_file_id'] + "'>here</a>
                to download</div></div>";
        }
    }catch(error){
        return "<pre>Error: " + error.toString() + "\n" + JSON.stringify(responses, null, 2) + "</pre>";
    }
  }
  if(task.status === 'error'){
    return "<pre> Error: untoggle for error message(s) </pre>";
  }
  return "<pre> Downloading... </pre>";
}
```





Browser Scripts

- You can toggle a script on/off per task or globally

completed ▲ mythic_admin's task: 205 - at Fri Nov 20 2020 10:30:12

— ps

PID	ARCH	NAME	USER	BIN_PATH
5756	x64	ApplicationFrameHost	WIN10\win10_local_admin	C:\WINDOWS\system32\ApplicationFrameHost.exe
7500	x64	audiodg	NT AUTHORITY\LOCAL SERVICE	C:\WINDOWS\system32\AUDIODG.EXE
1764	x64	backgroundTaskHost	WIN10\win10_local_admin	C:\WINDOWS\system32\backgroundTaskHost.exe
2312	x64	backgroundTaskHost	WIN10\win10_local_admin	C:\WINDOWS\system32\backgroundTaskHost.exe
5616	x64	backgroundTaskHost	WIN10\win10_local_admin	C:\WINDOWS\system32\backgroundTaskHost.exe
5632	x64	backgroundTaskHost	WIN10\win10_local_admin	C:\WINDOWS\system32\backgroundTaskHost.exe
5964	x64	browser_broker	WIN10\win10_local_admin	C:\WINDOWS\system32\browser_broker.exe
5368	x64	conhost	WIN10\win10_local_admin	C:\WINDOWS\system32\conhost.exe
8560	x64	conhost	WIN10\win10_local_admin	C:\WINDOWS\system32\conhost.exe

completed ▲ mythic_admin's task: 205 - at Fri Nov 20 2020 10:30:12

— ps


```
[
    {
        "process_id":  5756,
        "architecture":  "x64",
        "name":  "ApplicationFrameHost",
        "user":  "WIN10\\win10_local_admin",
        "bin_path":  "C:\\WINDOWS\\system32\\ApplicationFrameHost.exe",
        "parent_process_id":  ""
    },
    {
        "process_id":  7500,
        "architecture":  "x64",
        "name":  "audiodg",
        "user":  "NT AUTHORITY\\LOCAL SERVICE",
        "bin_path":  "C:\\WINDOWS\\system32\\AUDIODG.EXE",
        "parent_process_id":  ""
    },
```

Browser Scripts

- Scripts can be either tied to a specific command or be a "helper" script that all scripts for your agent can access
  - Ex: creating a table is a helper script
  - Ex: processing hercules' ps output is command specific
- Scripts live as **.js** files in your **hercules/mythic/browser_scripts** folder
- If you add or edit one of these in your container, restart the container to pull in updates

Browser Scripts

```python
class PsCommand(CommandBase):
    cmd = "ps"
    needs_admin = False
    help_cmd = "ps"
    description = "This uses Get-Process to return a formatted version of the running processes"
    version = 1
    is_exit = False
    is_file_browse = False
    is_process_list = True
    is_download_file = False
    is_remove_file = False
    is_upload_file = False
    author = "@its_a_feature_"
    attackmapping = ["T1057"]
    browser_script = BrowserScript(script_name="ps", author="@its_a_feature_")

    async def create_tasking(self, task: MythicTask) -> MythicTask:
        return task
```

<--- Command specific example. Include **browser_script** entry where the script name is the name of the .js file

Shared script in the builder.py file ---> Include an array of **support_browser_scripts** with the same format

```python
build_parameters = {
    "output": BuildParameter(
        name="output",
        parameter_type=BuildParameterType.ChooseOne,
        description="Choose output format",
        choices=["ps1", "base64"],
    )
}
c2_profiles = ["HTTP"]
support_browser_scripts = [
    BrowserScript(script_name="create_table", author="@its_a_feature_")
]
```

Mythic Scripting

- At its core, Mythic is a webserver with RESTful endpoints
  - If we authenticate properly, we can hit those same RESTful endpoints programmatically vs the UI
- Mythic now has a PyPI package for scripting: **mythic**
- The scripting page for Mythic shows many examples and explains the kinds of functions available
  - https://docs.mythic-c2.net/scripting

Mythic Scripting

- This is super useful for agent development
  - Automatically issue tasks and process output for new callbacks
  - Automatically create payloads and deploy them
- Scripting also uses WebSockets for notifications from Mythic about files, tasks, responses, payloads, etc

External Agents

- Mythic can "install" agents from GitHub, GitLab, BitBucket, etc
  - These external agents follow a specific folder format
  - Hooks into Payloads, C2, Documentation, Wrappers
  - https://github.com/its-a-feature/Mythic_External_Agent
- An organization exists on GitHub to aggregate these external agents for easy discovery:
  - https://github.com/MythicAgents
  - Reach out via Twitter (@its_a_feature_) or Bloodhound Slack for an invite and you can push your own repo

www.specterops.io
@specterops
info@specterops.io