ZOMBIE OPERATIONS MANAGEMENT PROTOCOL

Botnet Command and Control Interface

PROTOCOL SPECIFICATION

January 2024

prepared for

Computer Networks
CSC 382
Centre College

by

Michael Liao
Pierce Mason

Table of Contents

Introduction

We were asked to design a general-purpose protocol that would support messages between a command and control (C&C) system and a variable-size pool of machines (zombies or bots) in order to facilitate running Python scripts and collecting data based on the results of those scripts. The result of this design is the Zombie Operations Management Protocol (ZOMP).

Particulars

The C&C system must be able to:

- Ask a **specific** zombie to run a script.
- Ask a **specific** zombie to stop running a script.
- Ask a **specific** zombie to return the result of a script (a report).
- Ask **all** zombies to run a script.
- Ask **all** zombies to stop running a script.
- Ask **all** zombies to return a report.

The zombies must be able to:

- Run a script on command and log its results into a file (a report).
- Inform the C&C system about the currently-running status of a script on command.
- Return reports to the C&C system on command.
- Make clear any errors or exceptions that may interfere with a request from the C&C system.

In order to facilitate these messages, we also require that zombies be able to notify the C&C system of their existence at the beginning of interaction, so special care must be taken to the act of connecting a zombie to the C&C system, detailed in the Connecting Zombies to C&C section.

Assumptions

We are operating under the following assumptions:

- All zombies already know the IP address of the C&C system.
- All zombies are preloaded with all Python scripts to be run.

- All Python scripts are uniquely named.
- A zombie can run multiple scripts at the same time (via multithreading).
    - But it cannot run the *same* script at the same time.
    - **Multiple** different zombies can run the same script at the same time.
- Results from a given run of a script can be overwritten; i.e. each script will only have one report generated at a time.
    - So: when a script is run again, and a report is requested, the zombie will respond that no report is available.

Overview

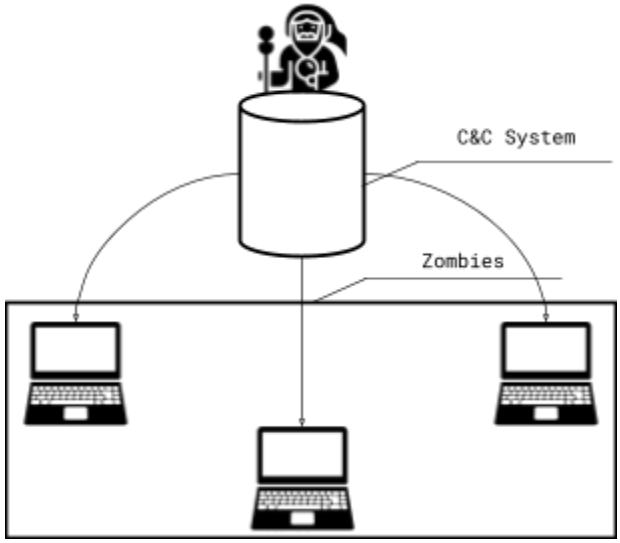

Fig. 1: Example of a botnet that ZOMP could be used on.

Because the C&C system will be a single device that will be always-on during the process, and zombies may join and leave at will, the C&C system will be functioning as a **server** to multiple zombie **clients**. In other words, zombies will initiate the interaction between themselves and the C&C system. The C&C system should begin running first, and then zombies may begin attempting to connect.

All messages will be sent through TCP, to ensure proper delivery between hosts.

If the C&C and zombies were on a single local network, broadcasting zombie requests to join the system might be viable. Because this is not necessarily the case for our situation, we assume that we may hardcode the C&C's IP address into zombies for them to initiate a connection.

C&C System State

The C&C will maintain information about a dynamic number of zombies that are currently connected to it. This information will be updated accordingly with new joins and departures.

Zombie State

When a zombie is connected to the C&C, it will maintain information about the scripts it is currently running and the output from those that have completed running.

Specifications

Both requests and responses are given numeric codes in order to make them more language-agnostic when it comes to implementation. Code messages (whether they be status messages or command words) are added for clarity, but when implemented in other languages, need not be exact.

We also leave room for further reporting in the “bands” or ranges of code numbers in order to give loose categorization.

Protocol version is also an included field for future proofing reasons: this document establishes ZOMP/1.0. Protocol version follows this format mask: **ZOMP/*<version number>***.

Connecting Zombies to C&C

Note that this process is also called registering a zombie/initiating a connection elsewhere in the document.

The C&C will begin listening at port **1932** for zombie requests to join its hivemind.

When a zombie begins running, it will send the following request to its (known) C&C server at port **1932**:

```
ZOMP/1.0 00 Ready to be registered\r\n
\r\n
```

Fig. 2: Request message to be registered by a zombie to C&C.

The C&C system will then respond with one of two possible messages:

```
ZOMP/1.0 0 ACCEPT\r\n
\r\n
```

Fig. 3: ACCEPT response from C&C to a connecting zombie.

In this case, the C&C system now knows the zombie exists and is ready to begin sending proper C&C requests to it.

Or, it could send:

```
ZOMP/1.0 9 CLOSE\r\n
\r\n
```

Fig. 5: CLOSE response from C&C to a connecting zombie.

In this case, the C&C system may have some extraneous condition where it cannot establish the connection, and this message is followed by closing the connection. Note that this message may be sent at any time from the C&C to a particular zombie, not just during the initial connection.

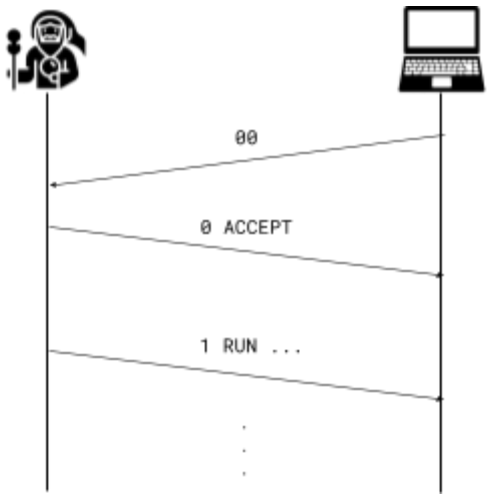

Fig. 4: Sequence diagram for an ACCEPTed zombie connection.

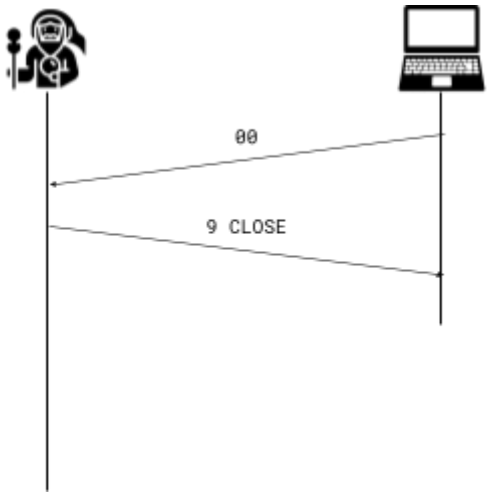

Fig. 6: Sequence diagram for a CLOSEd zombie connection.

C&C Requests

C&C Request Format

```
<protocol version> <request code> <command word>\r\n
<script name> <optional command-line arguments for script>\r\n
\r\n
```

Fig. 7: ZOMP C&C request format.

Command words are listed in the table below, and correspond to actions the C&C would like the zombies to perform accordingly.

Note that we do not have the zombie recipient explicitly called. It is expected that the C&C system will manage the zombie sockets, and choose to send this message only to the zombies that will run/report the script. This also means that, in order to command **all** zombies to run or report a certain script, it will just send this request to every active zombie socket.

Note that all script names should end with the **.py** file extension, and all optional command-line arguments are space-delimited. An example is provided below, where **clairvoyance.py** is a script that takes text filenames as input. The carriage return line feed characters (**/r/n**) have been invisibilized for readability, but exist as whitespace. More examples can be found in the Example Interactions section.

```
ZOMP/1.0 1 RUN
clairvoyance.py dungeon.txt castle.txt tavern.txt

```

Fig. 8: Example ZOMP request.

C&C Request Codes

Request Code	Command Word
0	ACCEPT
1	RUN

Request Code	Command Word
2	STOP
3	REPORT
5	NOT UNDERSTOOD
9	CLOSE

Fig. 9: C&C request codes.

Note that request code **0** is reserved for the initial registration of a zombie, request code **5** is for error reporting (in the case that a response comes back malformed), and that request code **9** can be sent at any point to discontinue connection to a zombie (primarily used to reject incoming connections during initial registration, but also a valid way to peacefully shut off a connection).

Zombie Responses

When a zombie is asked to run a script, it will either report that it doesn't have a script, that it is currently running, or create a thread to begin the script and affirm that the script has started running.
Upon receiving a request to stop running a particular script, the zombie will attempt to stop and give an affirmation, or respond that the script wasn't currently running.

When asked to report on a script, the zombie will respond with the status of it: either that it hasn't been run, that it is currently running, or with the most recent output from that script. In particular, if a zombie is currently running a script, it will not remember the request and must be messaged again when the script finishes.

In the event that a zombie already has a report of a script and is asked to rerun it (overwriting the report), it will respond with the report as well as an affirmation that the script is being rerun. This is an extra safety measure to ensure that the C&C system does not accidentally overwrite any previous run's data.

Zombie Response Format

```
<protocol version> <status code> <status message>\r\n
<script name>
Content-Length: <character count>\r\n
\r\n
<variable-length report body>
```

Fig. 10: ZOMP zombie response format.

The first line of a zombie's response will include the protocol version, followed by one of a selection of status codes to signify the response to a message.

The second line will name which Python script was referenced to help the server understand which request this is a response to. If our assumptions change in later versions, more identifying information may be appended to this line (such as the process ID).

If the message should have a report, then the Content-Length will have the length of such output. Otherwise, the Content-Length will be 0.

We use a signal of two **\r\n** characters for the end of the header, so that more headers may be added in the future for improved functionality. Finally, we add the report body, if applicable.

An example response (perhaps to the previous example in Fig. 4) follows. Again, whitespace characters are resolved for readability. To see more examples, please refer to the Example Interactions section.

```
ZOMP/1.0 10 OK, script running
clairvoyance.py
Content-Length: 0

```

Fig. 11: Example ZOMP response to a 1 RUN request.

Zombie Status Codes

Status Code	Status Message
00	Ready to be registered
Error Reporting (00-09)	
01	Bad request
02	Script not found
Responses to **RUN** (10-19)	
10	OK, script running
11	Ignore, script already running
12	OK, returning existing report
Responses to **STOP** (20-29)	
20	OK, stopping script
21	Ignore, script not currently running
22	Ignore, script completed running
Responses to **REPORT** (30-39)	
30	OK, reporting
31	No report, waiting on completion
32	No report, not running script

Fig. 12: Zombie response status codes.

Note that the **00** status code is reserved for the initial connection of a zombie to the C&C system. It is mentioned in more detail in the Connecting Zombies to C&C section.

Example Interactions

Suppose three zombies (Arthur, Bilbo, Conan) all have Python scripts **fireball.py** (which takes a command-line argument of an integer, and outputs the sum of 8 times that integer many rolls of six-sided dice), **bless.py** (which takes command-line arguments of three text files and improves them), and **scrying.py** (which, after ten minutes, returns a random pre-written description of a person).

Beginning and Closing the Connection

Suppose that the C&C server is controlled using a command line interface. The C&C server will first become online, and wait for more zombies. (It is still listening to the user's command line, it will just be unable to send anything).

When a zombie attempts to connect, it will establish a TCP connection at port **1932** with the C&C IP. It will then send the message:

```
ZOMP/1.0 00 Ready to be registered
```

Fig. 13: Zombie-to-C&C message trying to connect.

From this point forward, the C&C may send the closing message:

```
ZOMP/1.0 9 CLOSE
```

Fig. 14: C&C-to-zombie message refusing/closing connection.

that will end communication. The zombie will also close its connection if it realizes the TCP connection has closed.

For a general case, the C&C will send a message:

```
ZOMP/1.0 0 ACCEPT
```

Fig. 15: C&C-to-zombie message validating connection.

Which then prompts the zombie to listen for further commands (run, stop, and report requests).

Sample RUN Requests and Responses

C&C can issue a RUN **fireball.py** request to just Arthur by sending its message along the relevant socket it's linked to him:

```
ZOMP/1.0 1 RUN
fireball.py 5
```

Fig. 16: A run request for the fireball.py script.

Arthur can then respond in a variety of ways. Since this is the first command any of them have received, Arthur can go ahead and do that, and respond accordingly:

```
ZOMP/1.0 10 OK, script running
fireball.py
Content-Length: 0
```

Fig. 17: An affirmation for the run request in Fig. 16.

It won't take very long for the script to complete, and Arthur will store the output into a file locally and wait for C&C to request it.

However, in the split second before Arthur finishes the fireball.py, C&C has decided it wants to RUN **scrying.py** on all zombies. So it will issue the following request, one at a time, to all zombies:

```
ZOMP/1.0 1 RUN
scrying.py
```

Fig. 18: A run request for the scrying.py script.

Each zombie will respond with the following message, Arthur included, as each zombie can run multiple scripts concurrently.

```
ZOMP/1.0 10 OK, script running
scrying.py
Content-Length: 0
```

Fig. 19: An affirmation for the run request in Fig. 18.

If, however, C&C tries to do the same (have all zombies run this script) with **fireball.py**, Arthur will respond with a different message, as he is still running it from the first time it was requested:

```
ZOMP/1.0 11 Ignore, script already running
fireball.py
Content-Length: 0
```

Fig. 20: The notification that a command has been ignored since the requested script is still running.

Given that some time has passed, enough for Arthur to finish running **fireball.py**, C&C can issue a request for Arthur to RUN **fireball.py** once more (the same message as Fig. 16). In this instance, Arthur has a report saved, and will send that data before running the script again and potentially overwriting that report.

```
ZOMP/1.0 12 OK, returning existing report
fireball.py
Content-Length: 3
136
```

Fig. 21: A response to a request to run a script while previous results exist.

RUN with Multiple Command Line Arguments

In the case of asking Merlin to RUN **bless.py**, where multiple command line arguments will be supplied, the message request will be:

```
ZOMP/1.0 1 RUN
bless.py cleric.txt fighter.txt wizard.txt
```

Fig. 22: A run request with a script requiring multiple command line arguments.

Responses will not be changed; just the request.

Sample STOP Requests and Responses

Let's say that we don't actually want Conan to be running **scrying.py** (after all, he's a barbarian, not a wizard!). The C&C needs to send this message to achieve that:

```
ZOMP/1.0 2 STOP
scrying.py

```

Fig. 23: A stop request for the scrying.py script.

To which Conan will terminate the process and respond:

```
ZOMP/1.0 20 OK, stopping script
scrying.py
Content-Length: 0

```

Fig. 24: An affirmation that a script has been terminated.

If the C&C makes the mistake of repeating that same message, Conan, having already stopped the process, will instead respond with:

```
ZOMP/1.0 21 Ignore, script not currently running
scrying.py
Content-Length: 0

```

Fig. 25: A notification that a stop request has been ignored, as the script requested has already been stopped.

However, given enough time, Merlin will have finished **scrying.py** and logged its output. Sending a STOP request to him then will result in the following message, informing the C&C that there is data logged.

```
ZOMP/1.0 22 Ignore, script completed running
scrying.py
Content-Length: 101
Volo
```

```
A pompous blowhard of a man who spends his time aggrandizing and
making mountains of a molehill.
```

Fig. 26: A notification that a stop request has been ignored, as the script requested has already terminated and has results.

Sample REPORT Requests and Responses

Let's say Arthur has finished the second run of **fireball.py** and we want to see the results. We can request it with this message to Arthur:

```
ZOMP/1.0 3 REPORT
fireball.py

```

Fig. 27: A request for the result report after running the fireball.py script.

Arthur will send this back if all is well:

```
ZOMP/1.0 30 OK, reporting
fireball.py
Content-Length: 3
148
```

Fig. 28: A report response for the output of the fireball.py script.

For a more complex report, we can see what Merlin outputs after having run **bless.py** to completion:

```
ZOMP/1.0 30 OK, reporting
bless.py
Content-Length: 329
cleric.txt
Shadowheart, while certainly a cleric, is not your average
heal-dispenser.

fighter.txt
Lae'zel may bring a fierce alien outlook to every interaction, but
underlying all her words is the burning desire to keep her peers
safe.
```

```
wizard.txt
Gale is quicker with his tongue than his spells, but don’t
underestimate his Magic Missile.
```

Fig. 29: A report response for the output of the bless.py script.

There are no guarantees on how the output report is formatted. That is up to the script and C&C to determine how the final result will be parsed.

Now say we want to see the result of Merlin’s run of **scrying.py**. However, it hasn’t finished yet (ten minutes haven’t elapsed yet!), so Merlin will respond with:

```
ZOMP/1.0 31 No report, waiting on completion
fireball.py
Content-Length: 0
```

Fig. 30: A response indicating that no report is available since the requested script is still running.

And if we want to see how Conan is doing with the same script after we stopped him prematurely, he would respond with:

```
ZOMP/1.0 32 No report, not running script
fireball.py
Content-Length: 0
```

Fig. 31: A response indicating no report is available since the script is not running and no data exists.

Sample Error Handling

If a request is malformed, like below, the zombie will have to notify it with a response indicating such.

```
RUN
fireball.py
```

Fig. 32: A malformed RUN request. The version and code are missing.

```
ZOMP/1.0 01 Bad request

```

Fig. 33: A bad request response from a zombie to the C&C.

Another kind of malformed request might be for a zombie to run a script it does not have:

```
ZOMP/1.0 1 RUN
invisibility.py

```

Fig. 34: A request to run a script that does not exist.

To which the zombie should respond:

```
ZOMP/1.0 02 Script not found

```

Fig. 35: A file not found response from a zombie to the C&C.

Another potential error is that, if the zombie is misconfigured, it may send back a malformed response:

```
10 Script run
Content-Length: -1
```

Fig. 36: A malformed response from a zombie to the C&C. It is missing the version and has an invalid content length.

In which case the C&C system can respond by informing the zombie that the message is bad.

```
ZOMP/1.0 5 NOT UNDERSTOOD

```

Fig. 37: A response from the C&C to a zombie after a bad response.