Research & Technique

n8n 임의 파일 읽기 취약점(CVE-2026-21858)

■ 서론

2026년 1월 7일, 오픈소스 워크플로우[1] 자동화 플랫폼인 n8n에서, 운영 환경에 따라 원격 코드 실행으로 이어질 수 있는 임의 파일 읽기 취약점(CVE-2026-21858)이 공개되었다. 해당 취약점은 'Ni8mare'라는 별칭으로 알려졌으며, n8n 의 Webhook(웹훅)[2] 및 Form(폼)[3] 요청 처리 과정에서 요청 데이터에 대한 검증이 충분히 이뤄지지 않은 데서 발생한다.

n8n 은 드래그 앤 드롭 방식으로 워크플로우를 구성할 수 있으며, 셀프 호스팅이 가능하다는 점에서 개인 사용자부터 기업 환경까지 폭넓게 사용되고 있다. 보안 검색 엔진 Censys 의 분석 결과, 2026년 2월 기준 전 세계적으로 약 113,052대의 n8n 인스턴스가 활성화되어 있으며, 그중 한국은 약 9,266대(8.22%)로 전 세계 4위의 높은 사용량을 기록하고 있다.

Host.location.country	Count of Hosts	%
United States	28,065	24.90%
Germany	17,928	15.90%
France	10,448	9.27%
South Korea	9,266	8.22%
Brazil	4,656	4.13%
Singapore	4,628	4.11%
India	3,949	3.50%
Netherlands	3,164	2.81%
Finland	3,121	2.77%
Vietnam	2,794	2.48%
China	2,779	2.47%

그림 1. 국가별 사용량 (Censys, 2026.02.11)

공격자는 폼 기반 워크플로우를 대상으로 조작된 요청을 전송해, 서버가 로컬 파일을 업로드 파일로 오인하도록 유도할 수 있다. 그 결과 서버 내 민감한 파일이 워크플로우 처리 과정에서 외부로 노출될 수 있다. 따라서 n8n 을 운영 중인 조직은 사용 중인 버전이 취약한 범위에 해당하는지 신속히 확인하고, 보안 패치를 적용하거나 추가적인 보호 조치를 검토할 필요가 있다.

[1] 워크플로우: 특정 업무를 자동화하기 위해 설계된 일련의 작업 흐름

[2] Webhook: 외부에서 서버로 특정 데이터를 실시간으로 보내주는 통로

[3] Form: 사용자가 텍스트나 파일 등을 직접 입력할 수 있는 데이터 입력 양식

■ 영향받는 소프트웨어 버전

CVE-2026-21858 에 취약한 소프트웨어는 다음과 같다.

S/W 구분	취약 버전
n8n	1.65.0 이상 1.121.0 미만

■ 공격 시나리오

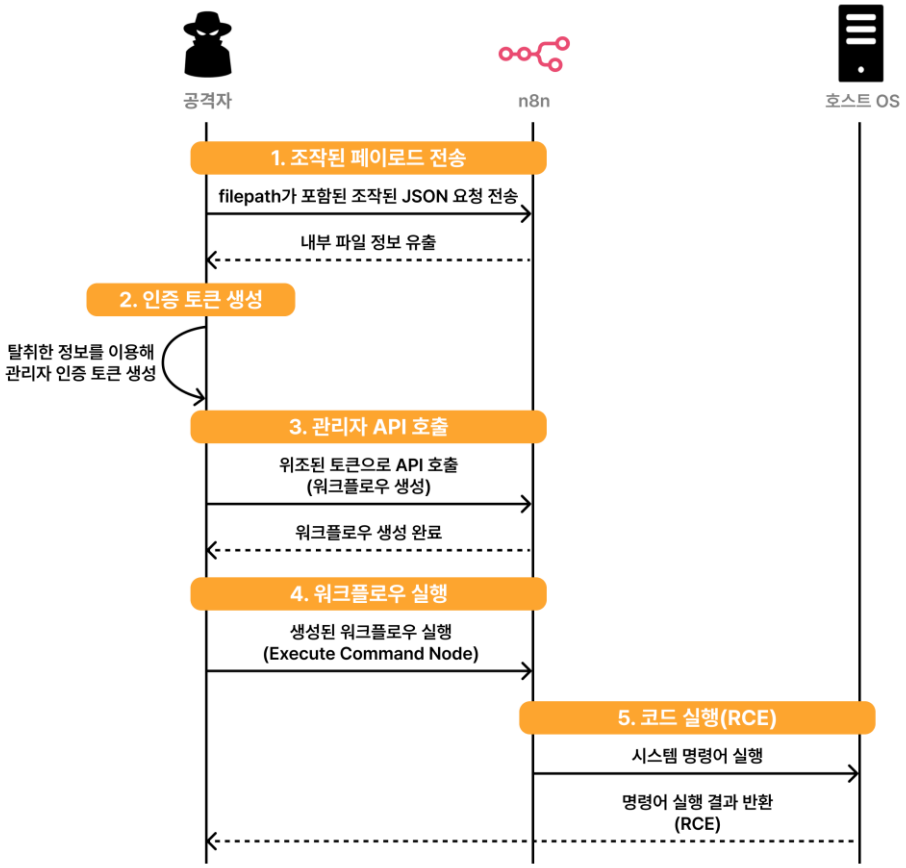

그림 2. 공격 시나리오

① 공격자는 Form Trigger[4](폼 트리거) 엔드포인트에 피해자 서버 파일 경로를 가리키도록 조작된 JSON 요청을 전송하여 내부 파일을 유출한다.
② 공격자는 탈취한 내부 파일에서 DB 정보 및 암호화 키로 관리자 JWT를 생성한다.
③ 공격자는 생성한 JWT로 REST API 요청을 사용하여 워크플로우와 시스템 명령 실행 노드를 생성한다.
④ 공격자는 시스템 명령 실행 노드에 명령어를 설정하고 워크플로우를 실행한다.
⑤ 시스템 명령이 실행되고, 공격자는 그 실행 결과를 확인한다.

[4] Form Trigger: n8n이 제공하는 웹 인터페이스를 통해 파일을 업로드할 수 있게 해주는 기능

■ 테스트 환경 구성 정보

테스트 환경을 구축하여 CVE-2026-21858 의 동작 과정을 살펴본다.

이름	정보
피해자	ubuntu:22.04 & n8n 1.120.4 (172.17.0.2)
공격자	Kali Linux (172.17.0.4)

■ 취약점 테스트

Step 1. 환경 구성

피해자 PC 에 n8n 취약 버전을 설치하여 취약 환경을 구성한다. CVE-2026-21858 취약점 테스트 구성을 위한 도커 이미지 및 취약점 테스트 파일은 아래 EQSTLab GitHub Repository 에서 확인할 수 있다.

- URL: https://github.com/EQSTLab/CVE-2026-21858

로컬 환경에서 피해자 PC 와 취약 환경을 구성한다. 다음 명령어로 도커 이미지를 빌드한 뒤 실행한다.

```
> git clone https://github.com/EQSTLab/CVE-2026-21858.git
> cd CVE-2026-21858
> docker build -t n8n-vuln:1.120.4 .
> run.bat
```

n8n 서버가 실행되면, 피해자 PC 로 http://localhost:9000/setup 에 접근하여 관리자 계정을 생성한다.

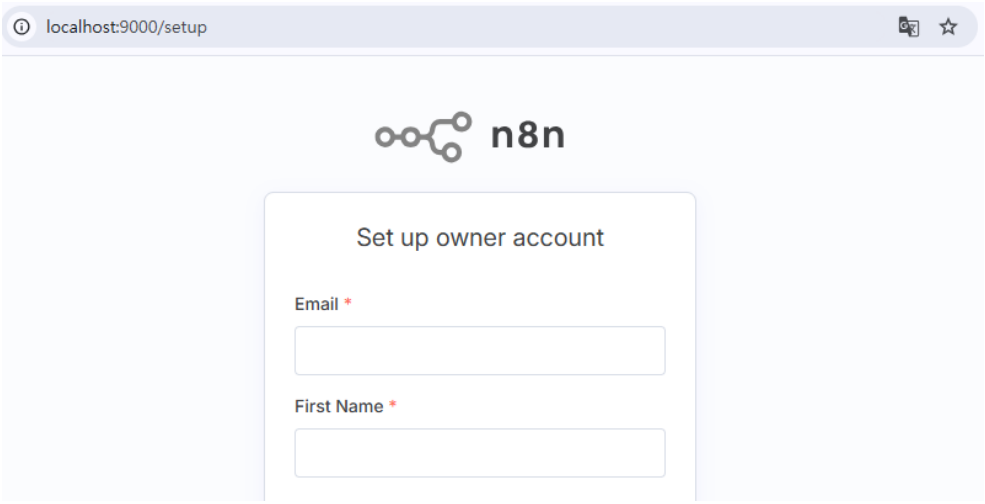

그림 3. 관리자 계정 생성

관리자 계정으로 워크플로우를 생성한다.

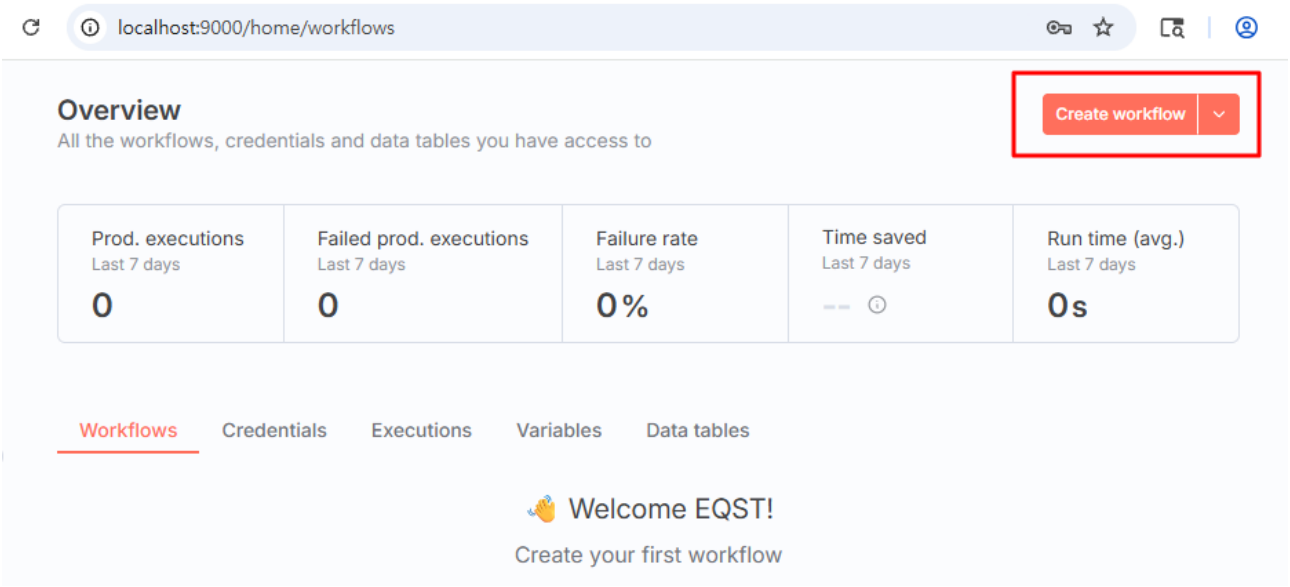

그림 4. 워크플로우 생성

workflow.txt 의 노드 JSON 을 워크플로우에 붙여 넣고, 워크플로우를 Active 상태로 전환한다. 해당 워크플로우는 폼 트리거를 통해 외부 사용자가 파일을 업로드하면, 이를 텍스트로 변환하여 응답하는 구조이다.

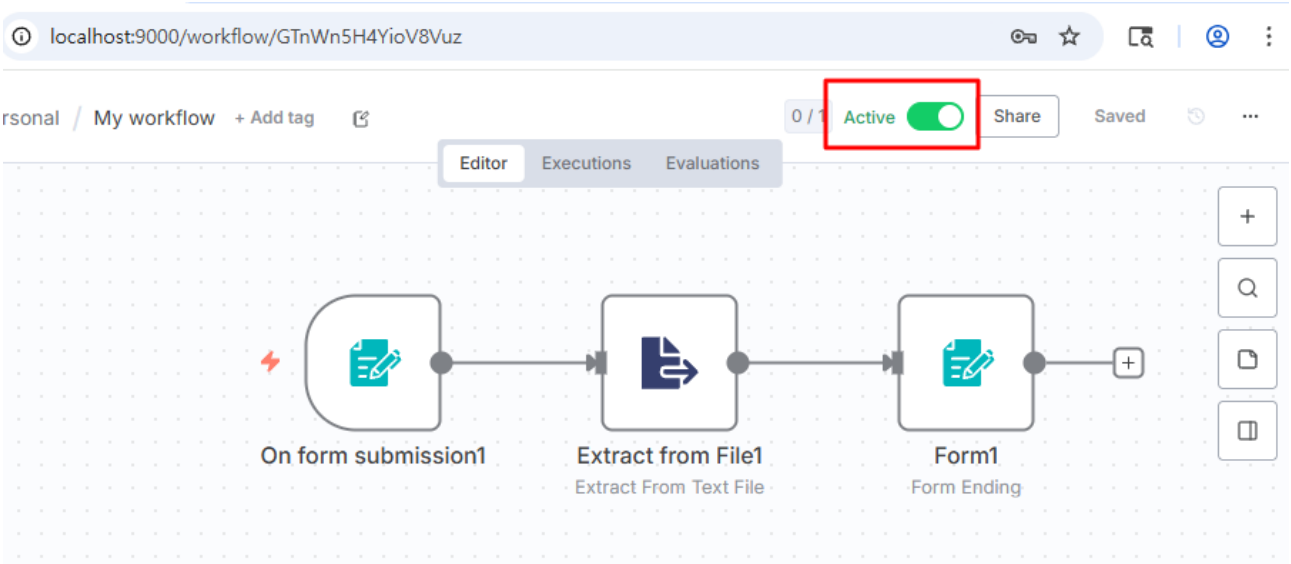

그림 5. 워크플로우 구성

On form submission1 노드를 더블클릭하여 Production URL 을 확인한다.

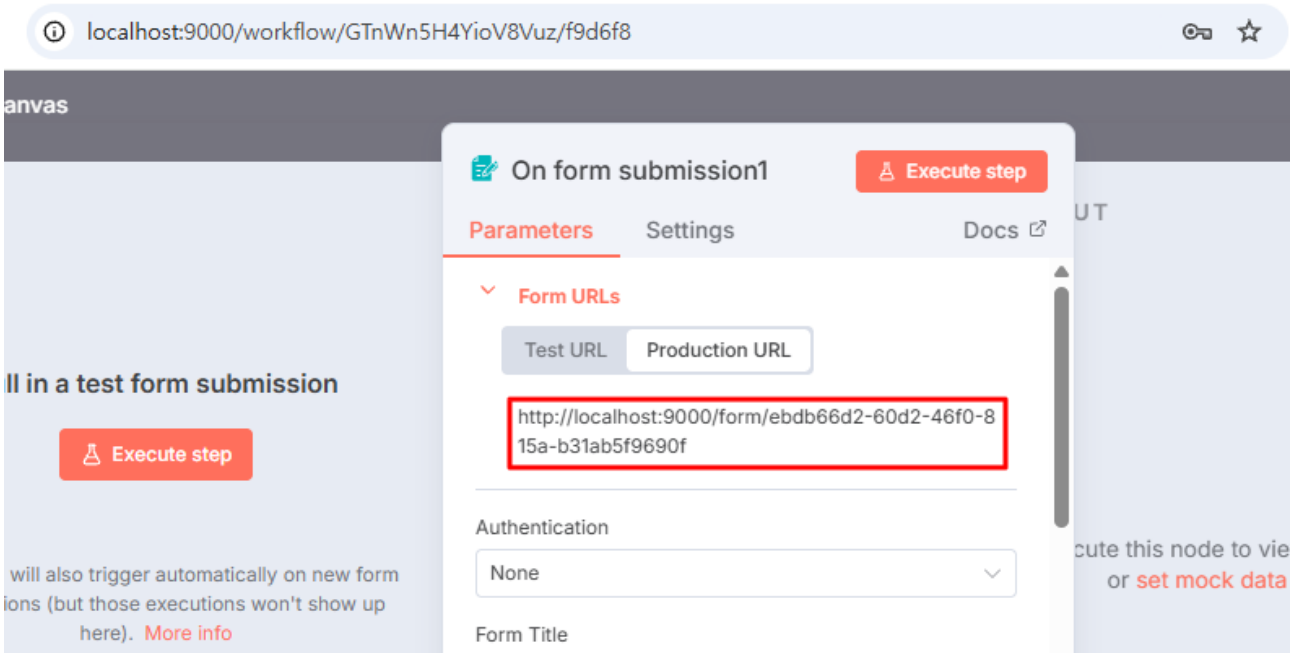

그림 6. 폼 트리거 페이지 URL

해당 URL 의 폼 트리거 페이지에서 파일을 업로드하는 것이 가능하다. 해당 페이지는 앞선 워크플로우에서 활성화할 경우, 외부 사용자들이 사용할 수 있도록 공개되는 엔드포인트이다.

그림 7. 파일 업로드 기능

Step 2. 취약점 테스트

공격자 PC 에서 취약점 엔트리 포인트[5] 인 n8n 폼 트리거 페이지를 확인한다.

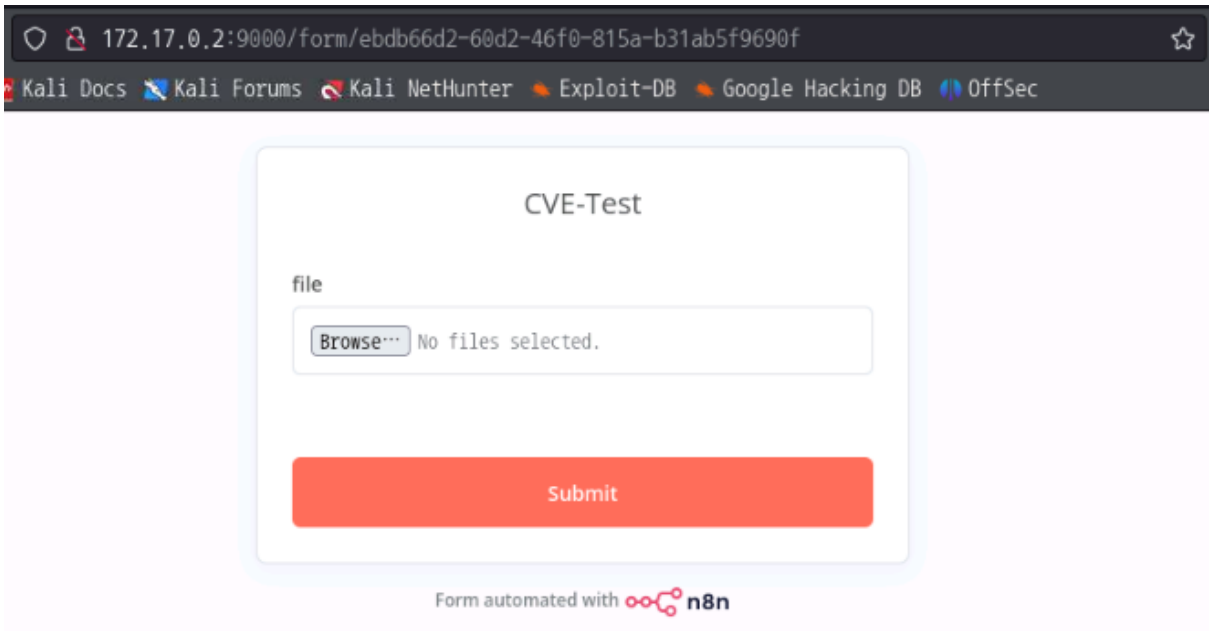

그림 8. 취약점 엔트리 포인트 확인

PoC 코드를 다운로드하고 실행한다.

```
> git clone https://github.com/EQSTLab/CVE-2026-21858.git
> cd CVE-2026-21858
> pip3 install -r requirements.txt
> python3 poc.py
```

앞서 확인한 폼 트리거 엔드포인트의 URL 을 입력하여 공격을 시도한다.

```
# python3 poc.py
=== n8n RCE Tool Setup ===
Enter Form URL (ex: http://localhost:9000/form/...): http://172.17.0.2:9000/form/ebdb66d2-60d
2-46f0-815a-b31ab5f9690f
```

그림 9. POC 실행

[5] 엔트리 포인트(Entry Point): 소프트웨어나 시스템에서 공격이 시작되는 진입점

관리자 JWT 를 생성하기 위해 필요한 정보들을 탈취한다. 서버가 디폴트 위치에 중요 정보를 보관하고 있을 경우, 공격자는 파일 경로를 예측해 이를 탈취할 수 있다. 탈취된 정보는 JWT 생성에 사용되며, 공격자는 n8n 을 관리자 권한으로 조작할 수 있게 된다.

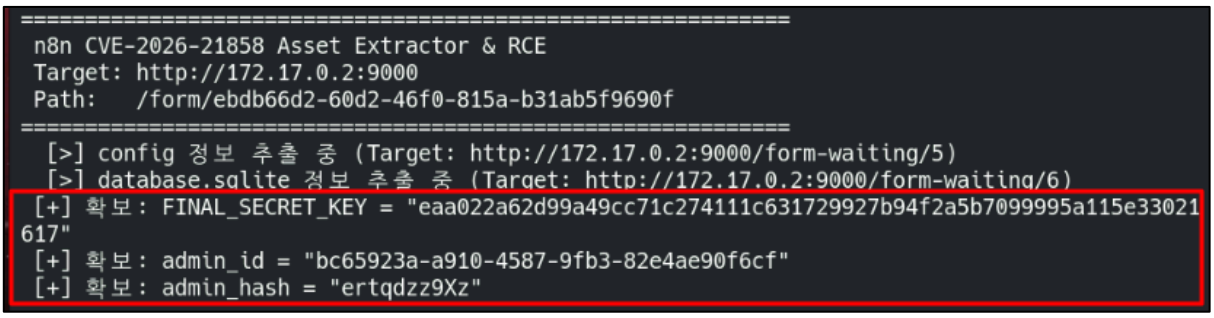

그림 10. 중요 정보 탈취

이후 공격자는 생성한 JWT 로 REST API 를 호출하고, 워크플로우 및 노드를 생성할 수 있다. 서버의 시스템 명령을 직접 실행할 수 있는 Execute Command 노드를 추가해 n8n 서버에 원격 명령을 수행할 수 있다.

그림 11. RCE

■ 취약점 상세 분석

취약점 상세 분석 장에서는 n8n의 웹훅 요청 처리 라이프사이클을 추적해 서버 내 파일을 읽는 공격 과정을 단계적으로 설명한다. [웹훅 개요]에서는 웹훅이 무엇인지 알아보고, [취약점 상세 분석] 에서는 관련 코드를 기반으로 취약점을 분석한다.

I. 웹훅 개요

웹훅은 외부 시스템과 n8n 을 실시간으로 연결하는 HTTP 기반의 자동 호출 메커니즘이다. 외부에서 웹훅 URL 로 요청을 전송하면, n8n 은 이를 신호로 받아 내부 워크플로우를 즉시 가동한다.

그림 12. 웹훅 vs. API 폴링 방식 차이

앞선 시나리오의 폼 트리거는 웹훅 메커니즘을 기반으로 동작하는 엔드포인트 중 하나이다. 일반적인 웹훅은 시스템 간 데이터 송수신에 최적화되어 있다. 이와 달리, 폼 트리거는 사용자 입력을 위한 HTML Form 인터페이스를 제공하며 내부적으로는 multipart/form-data 요청을 수신하여 워크플로우를 가동한다. 그러나 요청 처리 시 Content-Type 을 엄격히 검증하지 않는 취약점이 존재하여, 서버 내 파일 탈취 위험성이 제기되었다.

II. 취약점 상세 분석

취약 버전인 n8n(1.120.4) 코드를 분석하여 어떻게 서버 내 파일이 노출될 수 있었는지 알아본다.

Step 1. Content-Type 에 따른 req.body.files 할당 방식 차이

n8n 은 외부 HTTP 요청을 수신하면, 요청의 Content-Type 헤더에 따라 서로 다른 파서로 본문을 처리한다. 이때 파일 업로드 정보로 사용되는 req.body.files 가 생성되는 방식이 Content-Type 에 따라 달라진다.

```
835  async function parseRequestBody(
863      const { contentType } = req;
864      if (contentType === 'multipart/form-data') {
865          req.body = await parseFormData(req);
866      } else {
867          if (nodeVersion > 1) {
868              if (
869                  contentType?.startsWith('application/json') ||
870                  contentType?.startsWith('text/plain') ||
871                  contentType?.startsWith('application/x-www-form-urlencoded') ||
872                  contentType?.endsWith('/xml') ||
873                  contentType?.endsWith('+xml')
874              ) {
875                  await parseBody(req);
876              }
877          } else {
878              await parseBody(req);
879          }
880      }
881  }
```

// multipart일 경우 parseFormData() 파서 호출

// 그 외의 타입은 parseBody()를 통해 req.body가 채워짐

그림 13. Content-Type 헤더 기반 파서 분기 로직

(1) 정상 요청 - multipart/form-data

폼 트리거에 정상적으로 폼 데이터를 전송하면 multipart/form-data 로 요청이 들어오며, parseFormData()가 호출된다. parseFormData()는 formidable 파서를 통해 업로드 파일을 서버 임시 경로(/tmp/<random-id>)에 저장하고, 그 경로를 req.body.files['field-0'].filepath 로 설정한다.

그림 14. 정상 요청 시 req.body.files 생성 흐름

즉 정상 흐름에서는 filepath 가 서버에서 생성한 임시 경로로 고정된다. 사용자가 임의의 로컬 파일 경로를 지정할 수 없다.

(2) 조작된 요청 - application/json

공격자가 폼 트리거를 통한 요청의 Content-Type 을 application/json 으로 조작하면, n8n 은 parseBody()를 호출하여 본문 JSON 을 그대로 req.body 로 사용한다. 이 과정에서 본문에 files 구조를 포함하면, req.body.files['field-0'].filepath 가 JSON 에 포함된 값 그대로 설정된다.

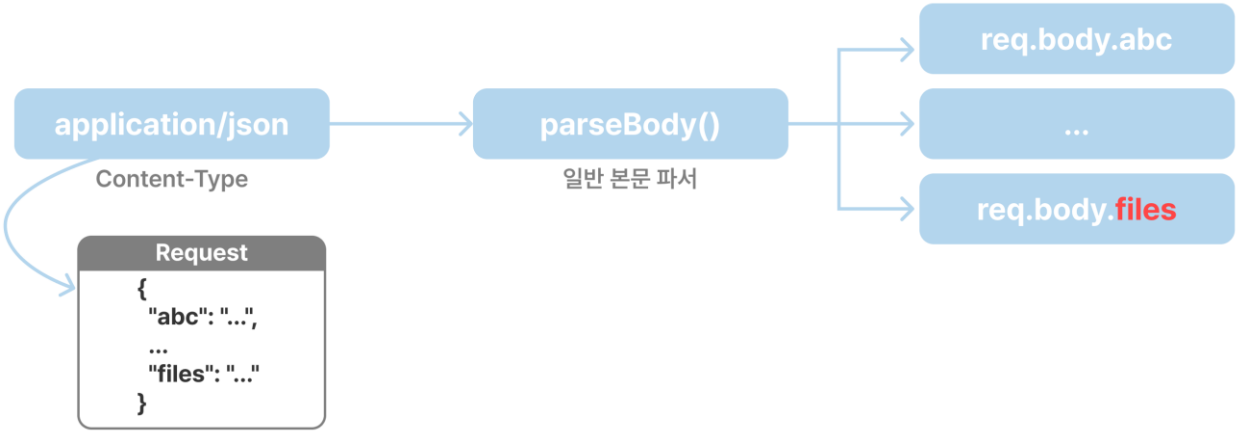

그림 15. 조작된 요청 시 req.body.files 생성 흐름

따라서 공격자는 아래 예시처럼 filepath 에 서버 로컬 파일 경로를 직접 주입할 수 있다.

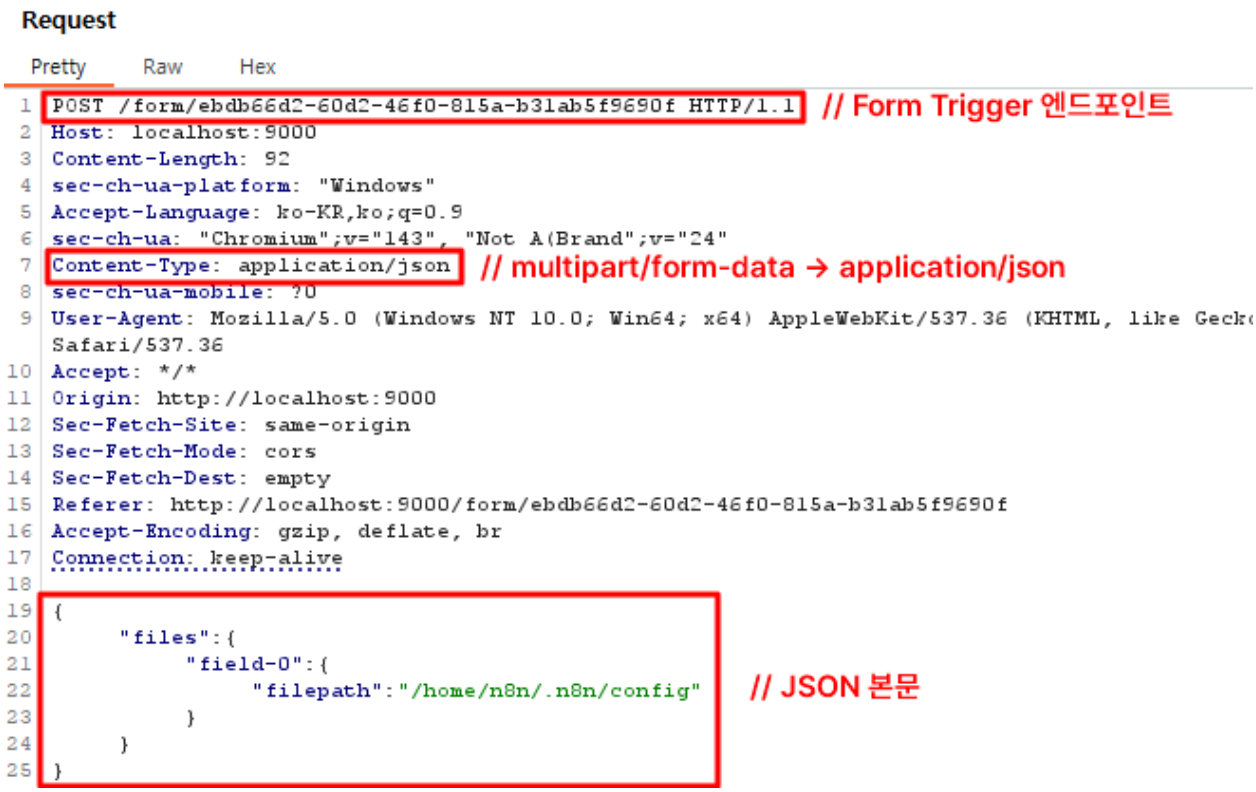

그림 16. application/json 요청을 통한 files 필드 조작 예시

즉, 조작된 요청 흐름에서는 공격자가 filepath 를 임의의 로컬 파일 경로로 지정할 수 있다.

Step 2. 트리거 식별 기준과 Content-Type 의 불일치

문제는 폼 트리거 요청이 들어오면 n8n 은 요청의 Content-Type 을 기준으로 이를 판별하지 않는다. 대신 요청 URL(/form/<uuid>)을 기준으로 처리 로직을 결정한다.

```
26   export class WebhookService {
341      async runWebhook(
360
361          const context = new WebhookContext(
362              workflow,
363              node,
364              additionalData,
365              mode,
366              webhookData,
367              [],
368              runExecutionData ?? null,
369          );
370
371          return nodeType instanceof Node
372              ? await nodeType.webhook(context)
373              : ((await nodeType.webhook.call(context)) as IWebhookResponseData);
374      }
375  }
```

// context에 현재 실행되고 있는
// 워크플로우, 노드, 실행 모드 등 값을 할당

// 현재 nodeType = FormTriggerV2
// 따라서 FormTriggerV2에 있는 webhook() 함수 호출

그림 17. Content-Type 과 무관한 FormTriggerV2 연결 과정

즉, 요청 본문 파싱은 Content-Type 에 따라 달라지지만, 처리 로직은 Content-Type 이 아니라 URL 기반 트리거 식별 결과로 결정된다. 이 구조 때문에 공격자가 application/json 으로 req.body.files['field-0'].filepath 를 주입해도, 서버는 폼 트리거 요청 처리를 수행한다.

```
207  export class FormTriggerV2 implements INodeType {
208      description: INodeTypeDescription;
209
210      constructor(baseDescription: INodeTypeBaseDescription) {
211          this.description = {
212              ...baseDescription,
213              ...descriptionV2,
214          };
215      }
216
217      async webhook(this: IWebhookFunctions) {
218          return await formWebhook(this);
219      }
220  }
```

// FormTriggerV2.webhook()은 formWebhook() 호출 값을 반환

그림 18. 폼 트리거 처리 로직 실행 (formWebhook)

Step 3. filepath 신뢰로 인한 로컬 파일 오인

폼 트리거 요청을 처리 시, prepareFormReturnItem() 함수로 요청 본문을 returnItem 에 저장하고, 이후 워크플로우에서 사용한다.

```
502  export async function formWebhook(
611
612      if (useWorkflowTimezone === undefined && node.typeVersion > 2) {
613          useWorkflowTimezone = true;
614      }
                                          // 현재 JSON 형식을 폼 형식의 JSON 형태로 변환
615
616      const returnItem = await prepareFormReturnItem(context, formFields, mode, useWorkflowTimezone);
617
618      return {
619          webhookResponse: { status: 200 },
                                          // 변환된 결과(JSON)와 상태코드 200을 반환
620          workflowData: [[returnItem]],
621      };
622  }
```

그림 19. 폼 입력값 및 파일 데이터 변환 처리 (prepareFormReturnItem)

prepareFormReturnItem()은 요청 본문에서 data 와 files 를 추출하여 폼 트리거 요청 처리 결과(returnItem)를 구성한다. 이 과정에서 files 가 multipart/form-data 파서가 생성한 값인지 여부는 확인하지 않는다. 단순히 "폼 트리거로 들어온 요청이라면 multipart/form-data 요청일 것" 이라는 전제 하에 처리한다.

```
349  export async function prepareFormReturnItem(
350      context: IWebhookFunctions,
351      formFields: FormFieldsParameter,
352      mode: 'test' | 'production',
353      useWorkflowTimezone: boolean = false,
354  ) {
355      const bodyData = (context.getBodyData().data as IDataObject) ?? {};
356      const files = (context.getBodyData().files as IDataObject) ?? {};
357
```

// data → 텍스트/필드 값

// files → 각 파일의 filepath, originalFilename 등 파일 메타 데이터

그림 20. files 객체 추출 및 할당

따라서 공격자가 application/json 요청으로 주입한 req.body.files['field-0'].filepath 도 정상 업로드 파일의 저장 경로로 간주되며, 이후 파일 복사/적재 로직에 그대로 전달될 수 있다.

Step 4. 워크플로우 구성에 따른 파일 노출

위 단계에서 추출된 파일은 returnItem 에 복사된다. 이렇게 생성된 returnItem 은 이후 워크플로우 동작에서 사용된다.

```
349   export async function prepareFormReturnItem(
386
387           const entryIndex = Number(key.replace(/field-/g, ''));
388           const fieldLabel = isNaN(entryIndex) ? key : formFields[entryIndex].fieldLabel;
389
390           let fileCount = 0;
391           for (const file of processFiles) {
392               let binaryPropertyName = fieldLabel.replace(/\W/g, '_');
393
394               if (multiFile) {
395                   binaryPropertyName += `_${fileCount++}`;
396               }                                    // file.filepath에 있는 경로 그대로 파일을 읽고 복사함
397
398               returnItem.binary![binaryPropertyName] = await context.nodeHelpers.copyBinaryFile(
399                   file.filepath,
400                   file.originalFilename ?? file.newFilename,
401                   file.mimetype,
402               );
403           }
404       }
```

그림 21. 오염된 경로의 파일을 복사

만약 공격자가 아래와 같이 filepath 로 서버 내 파일을 가리키는 경우, n8n 서버는 이를 정상적인 업로드 파일로 오인하고 해당 파일을 복사하여 사용한다.

```
{
  "files":{
    "field-0":{
      "filepath":"/home/n8n/.n8n/config"
    }
  }
}
```

워크플로우에 업로드 파일을 볼 수 있는 기능이 존재할 경우, 공격자가 지정한 로컬 파일의 내용을 확인할 수 있게 된다. 아래는 조작된 filepath 에 의해 n8n 의 config 파일의 내용이 노출된 모습이다.

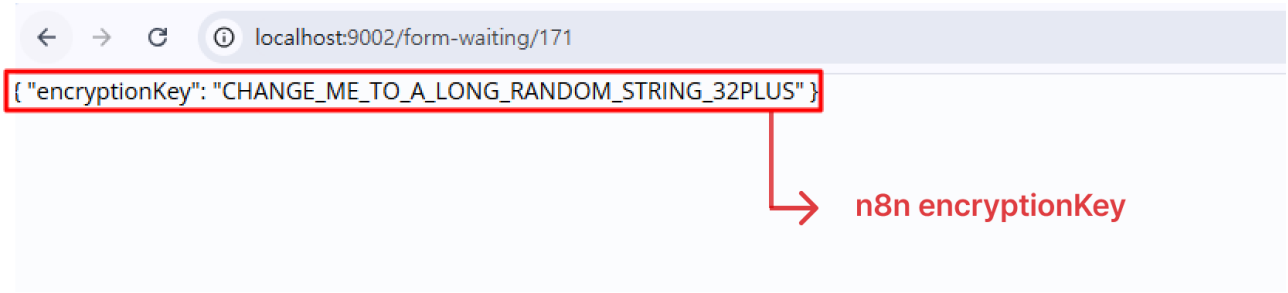

그림 22. 서버 내부 파일 내용이 응답 데이터로 반환된 결과 예시

■ 대응 방안

CVE-2026-21858 은 인증 없이 외부 요청만으로 서버 내부 파일에 접근할 수 있는 취약점으로, 실제 운영 환경에 미치는 영향이 매우 크다. 따라서 해당 취약점에 노출된 n8n 인스턴스는 즉각적인 패치 적용을 최우선으로 하되, 패치를 적용하기 어려운 경우를 대비한 추가적인 조치가 필요하다.

S/W 구분	패치 버전
n8n	1.121.0 이상

① 보안 패치 적용

2025 년 11 월 18 일 n8n 개발팀은 CVE-2026-21858 취약점에 대한 보안 패치를 공개했다. 해당 패치로 폼 데이터를 내부 실행 객체로 복사하기 전, 요청의 Content-Type 이 실제로 multipart/form-data 형식인지 여부를 명확히 확인하는 로직이 추가되었다.

packages/nodes-base/nodes/Form/utils/utils.ts 파일의 prepareFormReturnItem() 함수에서 이를 확인할 수 있다. 해당 패치가 적용된 환경에서는 조작된 데이터가 파일 경로 정보로 할당되는 과정 자체가 원천 봉쇄되므로, 본 취약점을 통한 공격은 근본적으로 차단된다.

```
s/nodes-base/nodes/Form/utils/utils.ts
    }

    export async function prepareFormReturnItem(
        context: IWebhookFunctions,
        formFields: FormFieldsParameter,
        mode: 'test' | 'production',
        useWorkflowTimezone: boolean = false,
    ) {
+       const req = context.getRequestObject() as MultiPartFormData.Request;
+       a.ok(req.contentType === 'multipart/form-data', 'Expected multipart/form-data');
        const bodyData = (context.getBodyData().data as IDataObject) ?? {};
        const files = (context.getBodyData().files as IDataObject) ?? {};
```

그림 23. CVE-2026-21858 보안 조치 내용

또한, 웹훅 및 테스트 코드 전반에 Content-Type 검증 로직을 확대 적용하여 유사한 취약점의 재발 가능성을 차단하였다.

```
s/nodes-base/nodes/Webhook/Webhook.node.ts
@@ -12,6 +12,7 @@ import type {
    INodeProperties,
} from 'n8n-workflow';
import { BINARY_ENCODING, NodeOperationError, Node } from 'n8n-workflow';
+ import * as a from 'node:assert';
import { pipeline } from 'stream/promises';
import { file as tmpFile } from 'tmp-promise';
import { v4 as uuid } from 'uuid';

@@ -316,6 +317,7 @@ export class Webhook extends Node {
        prepareOutput: (data: INodeExecutionData) => INodeExecutionData[][],
    ) {
        const req = context.getRequestObject() as MultiPartFormData.Request;
+       a.ok(req.contentType === 'multipart/form-data', 'Expected multipart/form-data');
        const options = context.getNodeParameter('options', {}) as IDataObject;
        const { data, files } = req.body;
```

그림 24. Webhook 노드 Content-Type 검증 추가

② 보안 패치 적용이 어려운 경우

운영 환경의 제약으로 즉각적인 패치가 어려운 경우, 공격 표면(Attack Surface)을 최소화하기 위해 다음과 같은 조치를 단계적으로 수행해야 한다.

1) 워크플로우 및 노드 관리

불필요한 폼 트리거 노드가 포함된 워크플로우를 즉시 비활성화 해야 한다. 특히 테스트 목적으로 생성된 워크플로우가 운영 환경에 그대로 남아 공격 진입점으로 악용되지 않도록 사전에 비활성화 하는 것이 바람직하다.

2) 노드별 인증 메커니즘 강화

트리거 노드에 Basic Auth 또는 Header 기반 인증을 적용해야 한다. 유효한 인증 정보가 포함되지 않은 외부 HTTP 요청만으로는 내부 워크플로우가 실행되지 않게 대처할 수 있다.

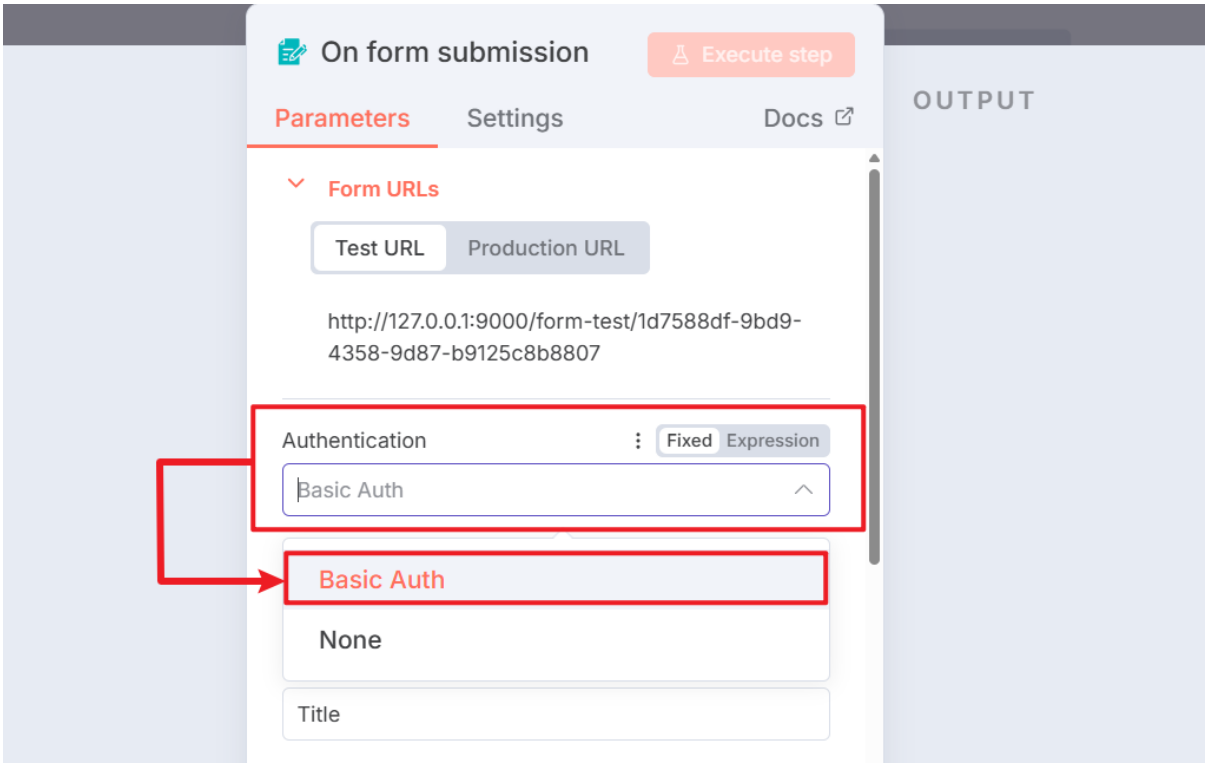

그림 25. 기본 인증 방식 추가

③ 진단 스크립트를 활용한 취약 여부 진단

운영 중인 n8n 인스턴스의 취약 여부를 신속하게 판별하기 위해 자동화된 점검 스크립트를 활용할 수 있다. 본 스크립트는 실제 공격을 수행하는 대신, Content-Type 설정에 따른 서버의 응답 패턴을 분석하여 안전하게 취약점을 진단한다.

1) 진단 원리

n8n 의 /form/ 엔드포인트에 application/json 형식의 요청을 보내고, 본문에 filepath 키워드를 포함하여 전송한다. 이때 서버가 요청을 정상적으로 수용할 경우, Content-Type 검증이 적용되지 않은 취약한 상태로 판단한다. 반대로 요청을 거부한다면 패치가 적용된 안전한 상태로 판단한다.

2) 진단 스크립트

```
import requests

def check_n8n_vulnerability():
    url = input("Enter n8n Form URL: ").strip()

    if "/form/" not in url:
        print("[NOTICE] Only '/form/' URLs are supported.")
        return

    try:
        res = requests.post(
            url,
            json={"filepath": "/etc/passwd", "fileName": "test"},
            headers={'Content-Type': 'application/json'},
            timeout=5
        )

        if res.status_code == 200:
            print(f"\n[VULNERABLE] {url} (Status: 200)")
        else:
            print(f"\n[SAFE] {url} (Status: {res.status_code})")

    except Exception as e:
        print(f"\n[ERROR] {url}: {e}")

if __name__ == "__main__":
    check_n8n_vulnerability()
```

출력되는 상태 메시지에 따라 취약 여부를 판별한다.

메시지	설명
[VULNERABLE]	해당 인스턴스는 취약하므로 즉시 패치 또는 보완 조치가 필요함
[SAFE]	보안 패치가 적용되었거나 요청이 정상적으로 차단됨
[NOTICE]	잘못된 URL 입력

■ 참고 사이트

• NVD: https://nvd.nist.gov/vuln/detail/CVE-2026-21858
• n8n GitHub Security: https://github.com/n8n-io/n8n/security
• n8n Docs: https://docs.n8n.io/integrations/builtin/core-nodes/n8n-nodes-base.form/
• The Hacker News: https://thehackernews.com/2026/01/critical-n8n-vulnerability-cvss-100.html
• CYERA: https://www.cyera.com/research-labs/ni8mare-unauthenticated-remote-code-execution-in-n8n-cve-2026-21858
• Chocapikk GitHub: https://github.com/Chocapikk/CVE-2026-21858