CSV Agent Prompt Injection Remote Code Execution Vulnerability

Critical **igor-magun-wd** published **GHSA-3hjv-c53m-58jj** last week

Package

flowise (npm)

Affected versions	Patched versions
<= 3.0.13	3.1.0

flowise-components (npm)

<= 3.0.13	3.1.0

Description

Please find POC file here: https://trendmicro-my.sharepoint.com/:u:/p/kholoud_altookhy/IQD5eFPC1-orTptT5Bj7ix54AQTZU7MOHGS8XfC6dZ2H_aQ?e=Q29PiQ

ZDI-CAN-29411: FlowiseAI Flowise CSV Agent Prompt Injection Remote Code Execution Vulnerability

-- CVSS ----------

9.8: AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

-- ABSTRACT ----------

Trend Micro's Zero Day Initiative has identified a vulnerability affecting the following products:
Flowise - Flowise

-- VULNERABILITY DETAILS ----------

- Version tested: 3.0.13
- Installer file: hxxps://github.com/FlowiseAI/Flowise
- Platform tested: Ubuntu 25.10

Analysis

FlowiseAI Flowise CSV Agent pythonCode Prompt Injection Remote Code Execution Vulnerability

This vulnerability allows remote attackers to execute arbitrary code on affected installations of FlowiseAI Flowise. Authentication is not required to exploit this vulnerability.

The specific flaw exists within the run method of the CSV_Agents class. The issue results from the lack of proper sandboxing when evaluating an LLM generated python script. An attacker can leverage this vulnerability to execute code in the context of the user running the server.

Product information

FlowiseAI Flowise version 3.0.13 (https://github.com/FlowiseAI/Flowise)

Setup Instructions

npm install -g flowise@3.0.13
npx flowise start

Root Cause Analysis

FlowiseAI Flowise is an open source low-code tool for developers to build customized large language model (LLM) applications and AI agents. It supports integration with various LLMs, data sources, and tools in order to facilitate rapid development and deployment of AI solutions. Flowise offers a web interface with a drag-and-drop editor, as well as an API, through an Express web server accessible over HTTP on port 3000/TCP.

One such feature of Flowise is the ability to create chatflows. Chatflows use a drag and drop editor that allow a developer to place nodes which control how an interaction with a LLM will occur. One such node is the CSV Agent node that represents an Agent used to answer queries on a provided CSV file.

When a user makes a query against a chatflow using the CSV Agent node, the run method of the CSV_Agents class will be called. This method will first read the contents of the csv file passed to the node and convert it to a base64 string. It will then set up a pyodide environment and create a python script to be executed in this environment. This python script will use pandas to extract the column names and their types from the provided csv file. The method will then create a system prompt for an LLM using this data as follows:

```
You are working with a pandas dataframe in Python. The name of the dataframe is df

The columns and data types of a dataframe are given below as a Python dictionary with
keys showing column names and values showing the data types.
```

```
{dict}

I will ask question, and you will output the Python code using pandas dataframe to
answer my question. Do not provide any explanations. Do not respond with anything
except the output of the code.

Security: Output ONLY pandas/numpy operations on the dataframe (df). Do not use
import, exec, eval, open, os, subprocess, or any other system or file operations. The
code will be validated and rejected if it contains such constructs.

Question: {question}
Output Code:
```

Where {dict} is the extracted column names and {question} is the initial prompt provided by the user.

This system prompt will be sent to an LLM in order for it to generate a python script based on the user's prompt, and the LLM generated response will be stored in a variable name pythonCode. The method will then evaluate the pythonCode variable in a pyodide environment.

While the LLM-generated Python script is evaluated in a non-sandboxed environment, there is a list of forbidden patterns that are checked for before the script is executed on the server. The function validatePythonCodeForDataFrame() enumerates through a list, named FORBIDDEN_PATTERNS, which contains pairs of regex pattern and reasons. Each regex pattern is run against the Python script, and if the pattern is found in the script, the script is invalidated and is not run, responding to the request with a reason for rejection.

The input validation can be bypassed, which can still lead to running arbitrary OS commands on the server. An example of this is the pattern `/\bimport\s+(?!pandas|numpy\b)/g`, which intends to search for lines of code which import a module other than pandas or numpy. This can be bypassed by importing along with pandas or numpy. For example, consider the following lines of code:

```
import pandas as np, os as pandas
pandas.system("xcalc")
```

pandas is imported, but so is the os module, with pandas as its alias. OS commands can then be invoked with `pandas.system()`.

Using prompt injection techniques, an unauthenticated attacker with the ability to send prompts to a chatflow using the CSV Agent node may convince an LLM to respond with a malicious python script that executes attacker controlled commands on the Flowise server.

It is also possible for an authenticated attacker to exploit this vulnerability by specifying an attacker controlled server in a chatflow. This server would respond to prompts with an attacker controlled python script instead of an LLM generated response, which would then be evaluated on the server.

comments documenting the issue have been added to the following code snippet. Added comments are prepended with "!!!".

From packages/components/nodes/agents/CSVAgent/core.ts

```
import type { PyodideInterface } from 'pyodide'
import * as path from 'path'
import { getUserHome } from '../../../src/utils'

let pyodideInstance: PyodideInterface | undefined

export async function LoadPyodide(): Promise<PyodideInterface> {
    if (pyodideInstance === undefined) {
        const { loadPyodide } = await import('pyodide')
        const obj: any = { packageCacheDir: path.join(getUserHome(), '.flowise', 'pyodid
        pyodideInstance = await loadPyodide(obj)
        await pyodideInstance.loadPackage(['pandas', 'numpy'])
    }

    return pyodideInstance
}

export const systemPrompt = `You are working with a pandas dataframe in Python. The name

The columns and data types of a dataframe are given below as a Python dictionary with ke
{dict}

I will ask question, and you will output the Python code using pandas dataframe to answe

Security: Output ONLY pandas/numpy operations on the dataframe (df). Do not use import,

Question: {question}
Output Code:`

export const finalSystemPrompt = `You are given the question: {question}. You have an an
Standalone Answer:`
```

From packages/components/nodes/agents/CSVAgent/CSVAgent.ts

```
import { BaseLanguageModel } from '@langchain/core/language_models/base'
import { AgentExecutor } from 'langchain/agents'
import { LLMChain } from 'langchain/chains'
import { ConsoleCallbackHandler, CustomChainHandler, additionalCallbacks } from '../../.
import { ICommonObject, INode, INodeData, INodeParams, IServerSideEventStreamer, PromptT
import { getBaseClasses } from '../../../src/utils'
import { LoadPyodide, finalSystemPrompt, systemPrompt } from './core'
import { validatePythonCodeForDataFrame } from '../../../src/pythonCodeValidator'
import { checkInputs, Moderation } from '../../moderation/Moderation'
import { formatResponse } from '../../outputparsers/OutputParserHelpers'
import { getFileFromStorage } from '../../../src'

class CSV_Agents implements INode {
    label: string
    name: string
    version: number
    description: string
    type: string
    icon: string
    category: string
```

```
    baseClasses: string[]
    inputs: INodeParams[]

    // !!! [... Truncated for Readability ...]

    // !!! input variable holds prompt from user
    async run(nodeData: INodeData, input: string, options: ICommonObject): Promise<strin
        const csvFileBase64 = nodeData.inputs?.csvFile as string
        const model = nodeData.inputs?.model as BaseLanguageModel
        // !!! the node in the chatflow may specify a custom system prompt
        const systemMessagePrompt = nodeData.inputs?.systemMessagePrompt as string
        const moderations = nodeData.inputs?.inputModeration as Moderation[]
        const _customReadCSV = nodeData.inputs?.customReadCSV as string

        // !!! the chatflow may contain moderation nodes that search for prompt injectio
        if (moderations && moderations.length > 0) {
            try {
                // Use the output of the moderation chain as input for the CSV agent
                input = await checkInputs(moderations, input)
            } catch (e) {
                await new Promise((resolve) => setTimeout(resolve, 500))
                // if (options.shouldStreamResponse) {
                //     streamResponse(options.sseStreamer, options.chatId, e.message)
                // }
                return formatResponse(e.message)
            }
        }

        // !!! [... Truncated for Readability ...]

        // Use pandas to determine column names and data types of selected csv file
        const pyodide = await LoadPyodide()

        // First load the csv file and get the dataframe dictionary of column types
        // For example using titanic.csv: {'PassengerId': 'int64', 'Survived': 'int64',
        let dataframeColDict = ''
        let customReadCSVFunc = _customReadCSV ? _customReadCSV : 'read_csv(csv_data)'
        try {
            const code = `import pandas as pd
import base64
from io import StringIO
import json

base64_string = "${base64String}"

decoded_data = base64.b64decode(base64_string)

csv_data = StringIO(decoded_data.decode('utf-8'))

df = pd.${customReadCSVFunc}
my_dict = df.dtypes.astype(str).to_dict()
print(my_dict)
json.dumps(my_dict)`
            dataframeColDict = await pyodide.runPythonAsync(code)
        } catch (error) {
            throw new Error(error)
```

```
        }

        // !!! ask LLM to come up with python script...
        // Then tell GPT to come out with ONLY python code
        // For example: len(df), df[df['SibSp'] > 3]['PassengerId'].count()
        let pythonCode = ''
        if (dataframeColDict) {
            const chain = new LLMChain({
                llm: model,
                // !!! prompt passed to LLM
                prompt: PromptTemplate.fromTemplate(systemPrompt),
                verbose: process.env.DEBUG === 'true' ? true : false
            })
            const inputs = {
                dict: dataframeColDict,
                // !!! question, which is later subbed into the system prompt, is given
                question: input
            }
            const res = await chain.call(inputs, [loggerHandler, ...callbacks])
            // !!! the LLM's responce is assigned to the pythonCode variable
            pythonCode = res?.text
            // Regex to get rid of markdown code blocks syntax
            pythonCode = pythonCode.replace(/^```[a-z]+\n|\n```$/gm, '')
        }

        // Then run the code using Pyodide (only after validating to prevent RCE)
        let finalResult = ''
        if (pythonCode) {
            // !!! code is validated for malicious code patterns
            const validation = validatePythonCodeForDataFrame(pythonCode)
            if (!validation.valid) {
                throw new Error(
                    `Generated code was rejected for security reasons (${
                        validation.reason ?? 'unsafe construct'
                    }). Please rephrase your question to use only pandas DataFrame opera
                )
            }
            try {
                const code = `import pandas as pd\n${pythonCode}`
                // !!! The python code is evaluated in a non-sandboxed environment
                // TODO: get print console output
                finalResult = await pyodide.runPythonAsync(code)
            } catch (error) {
                throw new Error(`Sorry, I'm unable to find answer for question: "${input
            }
        }

        // Finally, return a complete answer
        if (finalResult) {
            const chain = new LLMChain({
                llm: model,
                prompt: PromptTemplate.fromTemplate(
                    systemMessagePrompt ? `${systemMessagePrompt}\n${finalSystemPrompt}`
                ),
                verbose: process.env.DEBUG === 'true' ? true : false
            })
```

```
            const inputs = {
                question: input,
                answer: finalResult
            }

            if (options.shouldStreamResponse) {
                const handler = new CustomChainHandler(shouldStreamResponse ? sseStreame
                const result = await chain.call(inputs, [loggerHandler, handler, ...call
                return result?.text
            } else {
                const result = await chain.call(inputs, [loggerHandler, ...callbacks])
                return result?.text
            }
        }

        return pythonCode
    }
}

module.exports = { nodeClass: CSV_Agents }
```

From packages/components/src/pythonCodeValidator.ts:

```
/**
 * Validates LLM-generated Python code before execution in Pyodide to prevent
 * remote code execution (RCE). Only allows code that is safe for pandas
 * DataFrame operations. Rejects imports, exec/eval, file/system access, and
 * other dangerous constructs that could escape the intended DataFrame context.
 */

export interface PythonCodeValidationResult {
    valid: boolean
    reason?: string
}

// !!! most of these patterns can be bypassed
/**
 * Forbidden patterns that indicate unsafe Python code.
 * Uses word boundaries and context to minimize false positives (e.g. df.astype is allow
 */
const FORBIDDEN_PATTERNS: Array<{ pattern: RegExp; reason: string }> = [
    // Imports (we already inject "import pandas as pd"; LLM code must not add modules)
    { pattern: /\bfrom\s+\S+\s+import\b/g, reason: 'import statement (from...import)' },
    { pattern: /\bimport\s+(?!pandas|numpy\b)/g, reason: 'import statement (only pandas/
    // Dangerous builtins
    { pattern: /\beval\s*\(/g, reason: 'eval()' },
    { pattern: /\bexec\s*\(/g, reason: 'exec()' },
    { pattern: /\bcompile\s*\(/g, reason: 'compile()' },
    { pattern: /\b__import__\s*\(/g, reason: '__import__()' },
    { pattern: /\bopen\s*\(/g, reason: 'open()' },
    { pattern: /\bbreakpoint\s*\(/g, reason: 'breakpoint()' },
    { pattern: /\binput\s*\(/g, reason: 'input()' },
    { pattern: /\braw_input\s*\(/g, reason: 'raw_input()' },
    { pattern: /\bglobals\s*\(/g, reason: 'globals()' },
    { pattern: /\blocals\s*\(/g, reason: 'locals()' },
```

```
    { pattern: /\bgetattr\s*\(/g, reason: 'getattr()' },
    { pattern: /\bsetattr\s*\(/g, reason: 'setattr()' },
    { pattern: /\bdelattr\s*\(/g, reason: 'delattr()' },
    { pattern: /\breload\s*\(/g, reason: 'reload()' },
    { pattern: /\bfile\s*\(/g, reason: 'file()' },
    { pattern: /\bexecfile\s*\(/g, reason: 'execfile()' },
    // Dangerous modules / attributes
    { pattern: /\bos\./g, reason: 'os module' },
    { pattern: /\bsubprocess\./g, reason: 'subprocess module' },
    { pattern: /\bsys\./g, reason: 'sys module' },
    { pattern: /\bsocket\./g, reason: 'socket module' },
    { pattern: /\burllib\./g, reason: 'urllib module' },
    { pattern: /\brequests\./g, reason: 'requests module' },
    { pattern: /\b__builtins__\b/g, reason: '__builtins__' },
    { pattern: /\b__loader__\b/g, reason: '__loader__' },
    { pattern: /\b__spec__\b/g, reason: '__spec__' },
    { pattern: /\b__class__\b/g, reason: '__class__ (reflection)' },
    { pattern: /\b__subclasses__\s*\(/g, reason: '__subclasses__()' },
    { pattern: /\b__bases__\b/g, reason: '__bases__' },
    { pattern: /\b__mro__\b/g, reason: '__mro__' },
    { pattern: /\b__globals__\b/g, reason: '__globals__' },
    { pattern: /\b__code__\b/g, reason: '__code__' },
    { pattern: /\b__closure__\b/g, reason: '__closure__' }
]

/**
 * Validates that the given Python code is safe to run in the pandas DataFrame context.
 * Call this before passing LLM-generated code to pyodide.runPythonAsync().
 */
export function validatePythonCodeForDataFrame(code: string): PythonCodeValidationResult
    for (const { pattern, reason } of FORBIDDEN_PATTERNS) {
        pattern.lastIndex = 0
        // !!! each pattern is tested against the code
        if (pattern.test(code)) {
            return { valid: false, reason: `Forbidden construct: ${reason}` }
        }
    }

    return { valid: true }
}
```

Proof of Concept

A proof of concept for this vulnerability is provided in ./poc.py. It expects the following syntax:

```
python3 poc.py --method [server OR chatflow OR prompt_injection] [--user <USER> --
passwd <password> --host <HOST> --r_host <R_HOST> --r_port <R_PORT> --l_port <L_PO
--port <PORT> --cmd <CMD> --chatflow_id <CHAT_ID>]
```

Where USER is a username of a user on the server, PASSWORD is the user's password, HOST is the ip address of the vulnerable flowise server, R_HOST is the ip address of a malicious server started by this poc, R_PORT is the port a malicious server started by this poc is listening on (default: 5000), L_PORT is the port a malicious server started by this poc should listening on (default: 5000), PORT is the port the vulnerable flowise server is listening on (default: 3000), CMD is the command to execute on the flowise server (default: xcalc), and CHAT_ID is the chatflow id of a chatflow using the CSV Agent node.

This poc has three modes of operation controlled by the method argument. The method argument may have any of the values "server" OR "chatflow" OR "prompt_injection".

method = "server"

By default the poc will start a malicious server listening on the port specified by the <L_PORT> value. This server will respond to requests made to the "/api/chat" endpoint with a JSON object containing an LLM response that contains a malicious python script. This python script will execute a command specified by the value.

method = "chatflow"

By default, the poc will first establish an authenticated session on the server using the and arguments. It will then send a POST request to the "/api/v1/chatflow" endpoint with a JSON body containing a crafted chatflow using a CSV Agent node and a ChatOllama node configured with a server specified by the <R_HOST> and <R_PORT> arguments. It will then send a POST request to the "/api/v1/internal-prediction/" endpoint in order to trigger a prediction using the chatflow.

It is intended that the server specified in the chatflow is a server started using the server method of this poc. When making a prediction against this chatflow, flowise will send a request to the specified server in order to generate an LLM response. The response recieved by flowise will be evaluated as a python script. Upon successful exploitation, The argument passed to the server method of this poc will be executed on the vulnerable flowise server.

method = "prompt_injection"

By default, the poc will send a POST request to the "/api/v1/prediction/*chat_id*" endpoint, where *chat_id* is a vulnerable chatflow id specified by the <CHAT_ID> parameter. The JSON body of this request will contain a question member whose value will be a prompt containing a prompt injection. Upon successful exploitation, The argument will be executed on the vulnerable flowise server.

Due to the nature of LLM responses, it may take multiple attempts to be successful or require a different prompt injection technique depending on the model used.

Testing Environment

The provided proof of concept was tested using FlowiseAI Flowise version 3.0.13 running on a Ubuntu 25.10 VM. The prompt injection method was tested using the Llama3.2 model running in Ollama.

Credits

Dre Cura (@dre_cura) and Nicholas Zubrisky (@NZubrisky) of TrendAI Research

-- CREDIT ----------
This vulnerability was discovered by:
Dre Cura (@dre_cura) and Nicholas Zubrisky (@NZubrisky) of TrendAI Research

-- FURTHER DETAILS ----------

Supporting files:

If supporting files were contained with this report they are provided within a password protected ZIP file. The password is the ZDI candidate number in the form: ZDI-CAN-XXXX where XXXX is the ID number.

Please confirm receipt of this report. We expect all vendors to remediate ZDI vulnerabilities within 120 days of the reported date. If you are ready to release a patch at any point leading up to the deadline, please coordinate with us so that we may release our advisory detailing the issue. If the 120-day deadline is reached and no patch has been made available we will release a limited public advisory with our own mitigations, so that the public can protect themselves in the absence of a patch. Please keep us updated regarding the status of this issue and feel free to contact us at any time:

Zero Day Initiative
zdi-disclosures@trendmicro.com

The PGP key used for all ZDI vendor communications is available from:

http://www.zerodayinitiative.com/documents/disclosures-pgp-key.asc

-- INFORMATION ABOUT THE ZDI ----------
Established by TippingPoint and acquired by Trend Micro, the Zero Day Initiative (ZDI) neither re-sells vulnerability details nor exploit code. Instead, upon notifying the affected product vendor, the ZDI provides its Trend Micro TippingPoint customers with zero day protection through its intrusion prevention technology. Explicit details regarding the specifics of the vulnerability are not exposed to any parties until an official vendor patch is publicly available.

Please contact us for further details or refer to:

http://www.zerodayinitiative.com

-- DISCLOSURE POLICY ----------

Our vulnerability disclosure policy is available online at:

http://www.zerodayinitiative.com/advisories/disclosure_policy/

How This Differs from CVE-2026-41137

This advisory demonstrates a bypass for the input validation that was introduced to address CVE-2026-41137, by modifying the way packages are imported. We addressed this advisory by disallowing all imports in the CSV Agent.

Severity

Critical 9.2 / 10

CVSS v4 base metrics

Exploitability Metrics	
Attack Vector	Network
Attack Complexity	High
Attack Requirements	Present
Privileges Required	None
User interaction	None
Vulnerable System Impact Metrics	
Confidentiality	High
Integrity	High
Availability	High
Subsequent System Impact Metrics	
Confidentiality	None
Integrity	None
Availability	None

Learn more about base metrics

CVSS:4.0/AV:N/AC:H/AT:P/PR:N/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N

CVE ID

CVE-2026-41264

Weaknesses

No CWEs

Credits

zdi-disclosures Reporter