

WYDZIAŁ  
**ELEKTROTECHNIKI  
I INFORMATYKI**  
POLITECHNIKI RZESZOWSKIEJ

**Jakub Skrzynecki**

Bezpieczeństwo aplikacji webowych

File upload

## **Spis treści**

<b>1. Wprowadzenie.....</b>	<b>4</b>
<b>1.1. Definicja ataków typu File Upload .....</b>	<b>4</b>
<b>1.2. Znaczenie bezpieczeństwa przesyłania plików w aplikacjach         webowych .....</b>	<b>4</b>
<b>1.3. Potencjalne konsekwencje ataków File Upload .....</b>	<b>4</b>
<b>2. Analiza przesyłania plików przy pomocy http .....</b>	<b>5</b>
<b>2.1. Kontekst Analizy:.....</b>	<b>5</b>
Etap 1: Ustanowienie Połączenia TCP (Trójstopniowe Uzgadnianie).....	5
Etap 2: Żądanie HTTP POST i Rozpoczęcie Uploadu.....	6
Etap 3: Przesyłanie Danych Pliku i Kompresja Obrazu (JPEG/JFIF).....	7
Etap 4: Odpowiedź Serwera HTTP .....	9
Etap 5: Zakończenie Połączenia TCP.....	9
<b>3. Metody uruchamiania złośliwych plików przesłanych na     serwer .....</b>	<b>10</b>
<b>3.1. Środowisko bez zabezpieczeń.....</b>	<b>10</b>
<b>3.2. Filtrowanie rozszerzeń.....</b>	<b>10</b>
<b>3.3. Ograniczenia MIME .....</b>	<b>10</b>
<b>3.4. Manipulacja konfiguracją przez .htaccess .....</b>	<b>11</b>
<b>3.5. Local File Inclusion (LFI).....</b>	<b>11</b>
<b>3.6. Remote File Inclusion (RFI).....</b>	<b>11</b>
<b>3.7. Wykorzystanie wrapperów PHP (php://input, php://filter) .....</b>	<b>12</b>
<b>3.8. Niebezpieczna deserializacja.....</b>	<b>12</b>
<b>3.9. Ucieczka z sandboxu i eskalacja w kontenerach.....</b>	<b>12</b>
<b>3.10. Dynamiczne wykonanie kodu w aplikacjach (eval, refleksja) .....</b>	<b>12</b>
<b>4. Analiza programu realizującego funkcję serwera plików....</b>	<b>13</b>

4.1. Analiza JavaScript w funkcji przycisku serwera.....	13
4.2. Jak otwiera się okno wyboru pliku.....	13
5. Ograniczenia przesyłania dużych plików w aplikacjach webowych.....	14
5.1. Walidacja po stronie klienta .....	14
5.1.1. Sprawdzanie rozmiaru pliku.....	14
5.1.2. Ograniczenie typów plików.....	15
5.2. Walidacja po stronie serwera.....	16
5.2.1. Node.js + Express + Multer.....	16
5.2.2. PHP.....	16
5.3. Konfiguracja serwera WWW .....	17
5.3.1. NGINX .....	17
5.3.2. Apache.....	17
6. Podsumowanie.....	18
7. Literatura .....	19

# **1. Wprowadzenie**

## **1.1. Definicja ataków typu File Upload**

Współczesne aplikacje webowe stanowią fundamentalny element cyfrowej infrastruktury, umożliwiając szeroki zakres interakcji i funkcjonalności. Jedną z powszechnie stosowanych i cenionych przez użytkowników funkcji jest możliwość przesyłania plików na serwer – czy to w postaci zdjęć profilowych, dokumentów, multimediów, czy innych danych. Chociaż ta funkcjonalność znacząco podnosi użyteczność aplikacji, wprowadza jednocześnie potencjalne wektory ataku, które mogą być wykorzystane przez złośliwe podmioty. W niniejszym rozdziale przyjrzymy się bliżej jednemu z kluczowych zagrożeń związanych z procesem przesyłania plików: atakom typu "file upload".

## **1.2. Znaczenie bezpieczeństwa przesyłania plików w aplikacjach webowych**

Bezpieczeństwo transferu plików w środowisku sieciowym ma kluczowe znaczenie zarówno z perspektywy ochrony integralności systemu, jak i zachowania poufności danych użytkowników. W kontekście rosnącej liczby ataków na infrastrukturę IT oraz coraz surowszych regulacji prawnych (w tym RODO/GDPR), każda luka w mechanizmach uploadu plików może prowadzić do poważnych konsekwencji – od utraty informacji poufnych, poprzez uszczerbek na reputacji podmiotu, aż po nałożenie wysokich kar finansowych.

W konsekwencji proces projektowania i wdrażania funkcjonalności uploadu powinien obejmować wielowarstwowe zabezpieczenia: precyzyjną walidację typu MIME i rozszerzeń, ograniczenie rozmiaru i wymiarów plików, skanowanie antywirusowe, a także izolację i bezpieczne składowanie przesłanych plików z wykorzystaniem dedykowanych repozytoriów (np. Object Storage z kontrolą dostępu).

## **1.3. Potencjalne konsekwencje ataków File Upload**

Skutki ataków File Upload mogą przybierać bardzo różnorodny charakter, zagrażając zarówno stabilności działania serwera, jak i bezpieczeństwu całej organizacji. Wgranie i wykonanie złośliwego skryptu umożliwia napastnikowi pełne przejęcie kontroli nad środowiskiem serwerowym, co w praktyce może skutkować kradzieżą danych wrażliwych, modyfikacją oprogramowania czy instalacją backdoorów.

Ponadto zainfekowane pliki multimedialne bądź binaria z ukrytym malware mogą stać się wektorem szerzenia się infekcji w sieci wewnętrznej, umożliwiając dalsze ataki lateralne. Niewłaściwa segregacja uprawnień czy błędna konfiguracja ścieżek dostępu do plików sprzyjają eskalacji uprawnień, a w dłuższej perspektywie stanowią bazę dla zaawansowanych

i ukierunkowanych kampanii APT (Advanced Persistent Threat), w których napastnicy utrzymują ukryty, długotrwały dostęp do infrastruktury.

## 2. Analiza przesyłania plików przy pomocy http

Przeprowadzono analizę przesyłania pliku JPEG przy pomocy narzędzia Wireshark w celu przechwycenia komunikacji między klientem a serwerem. Zarejestrowane pakiety przedstawiono na Rys. 2.1.

(ip.src == 127.0.0.1 && tcp.dstport == 3000)    (ip.dst == 127.0.0.1 && tcp.srcport == 3000)					
No.	Time	Source	Destination	Protocol	Length Info
44	1.823757	127.0.0.1	127.0.0.1	TCP	56 59991 → 3000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
45	1.823819	127.0.0.1	127.0.0.1	TCP	56 3000 → 59991 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
46	1.823858	127.0.0.1	127.0.0.1	TCP	44 59991 → 3000 [ACK] Seq=1 Ack=1 Win=65280 Len=0
47	1.824182	127.0.0.1	127.0.0.1	TCP	750 59991 → 3000 [PSH, ACK] Seq=1 Ack=1 Win=65280 Len=706 [TCP PDU reassembled in 59]
48	1.824228	127.0.0.1	127.0.0.1	TCP	44 3000 → 59991 [ACK] Seq=1 Ack=707 Win=64768 Len=0
49	1.824359	127.0.0.1	127.0.0.1	TCP	16428 59991 → 3000 [PSH, ACK] Seq=707 Ack=1 Win=65280 Len=16384 [TCP PDU reassembled in 59]
50	1.824407	127.0.0.1	127.0.0.1	TCP	44 3000 → 59991 [ACK] Seq=1 Ack=17091 Win=48384 Len=0
51	1.824523	127.0.0.1	127.0.0.1	TCP	16428 59991 → 3000 [PSH, ACK] Seq=17091 Ack=1 Win=65280 Len=16384 [TCP PDU reassembled in 59]
52	1.824556	127.0.0.1	127.0.0.1	TCP	44 3000 → 59991 [ACK] Seq=1 Ack=33475 Win=65280 Len=0
53	1.824617	127.0.0.1	127.0.0.1	TCP	16428 59991 → 3000 [PSH, ACK] Seq=33475 Ack=1 Win=65280 Len=16384 [TCP PDU reassembled in 59]
54	1.824642	127.0.0.1	127.0.0.1	TCP	44 3000 → 59991 [ACK] Seq=1 Ack=49859 Win=48896 Len=0
55	1.824699	127.0.0.1	127.0.0.1	TCP	16428 59991 → 3000 [PSH, ACK] Seq=49859 Ack=1 Win=65280 Len=16384 [TCP PDU reassembled in 59]
56	1.824723	127.0.0.1	127.0.0.1	TCP	44 3000 → 59991 [ACK] Seq=1 Ack=66243 Win=65280 Len=0
57	1.824773	127.0.0.1	127.0.0.1	TCP	16428 59991 → 3000 [PSH, ACK] Seq=66243 Ack=1 Win=65280 Len=16384 [TCP PDU reassembled in 59]
58	1.824793	127.0.0.1	127.0.0.1	TCP	44 3000 → 59991 [ACK] Seq=1 Ack=82627 Win=48896 Len=0
59	1.824841	127.0.0.1	127.0.0.1	HTTP	9860 POST /upload/Doghi.jpg HTTP/1.1 (JPEG JFIF image)
60	1.824864	127.0.0.1	127.0.0.1	TCP	44 3000 → 59991 [ACK] Seq=1 Ack=92443 Win=39168 Len=0
61	1.825739	127.0.0.1	127.0.0.1	HTTP	226 HTTP/1.1 200 OK (text)
62	1.825772	127.0.0.1	127.0.0.1	TCP	44 59991 → 3000 [ACK] Seq=92443 Ack=183 Win=65280 Len=0
63	1.825907	127.0.0.1	127.0.0.1	HTTP	642 GET /list HTTP/1.1
64	1.825948	127.0.0.1	127.0.0.1	TCP	44 3000 → 59991 [ACK] Seq=183 Ack=93041 Win=38656 Len=0
65	1.830077	127.0.0.1	127.0.0.1	HTTP/JSON	299 HTTP/1.1 200 OK, JSON (application/json)
66	1.830119	127.0.0.1	127.0.0.1	TCP	44 59991 → 3000 [ACK] Seq=93041 Ack=438 Win=65024 Len=0
87	7.836816	127.0.0.1	127.0.0.1	TCP	44 3000 → 59991 [FIN, ACK] Seq=438 Ack=93041 Win=38656 Len=0
88	7.836859	127.0.0.1	127.0.0.1	TCP	44 59991 → 3000 [ACK] Seq=93041 Ack=439 Win=65024 Len=0

Rys. 2.1 Przechwycone pakiety podczas komunikacji klient-serwer

### 2.1. Kontekst Analizy:

Przechwycone pakiety pochodzą z komunikacji między klientem a serwerem działającymi na tej samej maszynie (localhost - 127.0.0.1), komunikującymi się przez tzw. pętlę zwrotną (Loopback). Klient używa dynamicznego portu (np. 59991), a serwer to aplikacja webowa nasłuchująca na porcie 3000.

Proces ten można podzielić na następujące etapy:

#### Etap 1: Ustanowienie Połączenia TCP (Trójstopniowe Uzgadnianie)

Zanim dane pliku zostaną przesłane, klient i serwer muszą nawiązać niezawodne połączenie przy użyciu protokołu TCP (Transmission Control Protocol). Ten etap nazywamy trójstopniowym uzgadnianiem (Three-Way Handshake), widocznym na Rys. 2.2.

44	1.823757	127.0.0.1	127.0.0.1	TCP	56 59991 → 3000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
45	1.823819	127.0.0.1	127.0.0.1	TCP	56 3000 → 59991 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
46	1.823858	127.0.0.1	127.0.0.1	TCP	44 59991 → 3000 [ACK] Seq=1 Ack=1 Win=65280 Len=0

Rys. 2.2 Pakiety Three way handshake

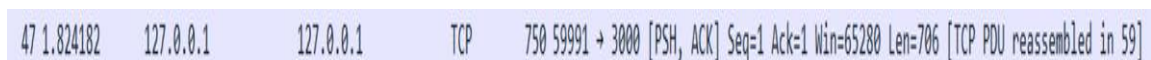
**Klient inicjuje połączenie (Pakiet 44):** Klient (127.0.0.1:59991) wysyła do serwera (127.0.0.1:3000) pakiet z ustawioną flagą SYN (Synchronize). Informacja [SYN] Seq=0 Win=65535 Len=0 oznacza, że klient prosi o synchronizację numerów sekwencyjnych, zaczyna swoją sekwencję od numeru 0 (względny), informuje o rozmiarze swojego bufora odbioru (okno - Win) i nie wysyła jeszcze żadnych danych (Len=0). Zawiera też opcje, takie jak maksymalny rozmiar segmentu (MSS).

**Serwer potwierdza i inicjuje swoje połączenie (Pakiet 45):** Serwer (127.0.0.1:3000) odpowiada klientowi pakietem z ustawionymi dwiema flagami: SYN i ACK (Acknowledge). Informacja [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 oznacza, że serwer również inicjuje swoją sekwencję numerów (od 0 - względny). Flaga ACK potwierdza otrzymanie pakietu SYN od klienta. Numer potwierdzenia Ack=1 oznacza, że serwer oczekuje następnego bajtu danych od klienta (numer sekwencyjny klienta + 1).

**Klient potwierdza (Pakiet 46):** Klient (127.0.0.1:59991) wysyła pakiet z ustawioną flagą ACK. Informacja [ACK] Seq=1 Ack=1 Win=65280 Len=0 oznacza, że klient potwierdza otrzymanie SYN-ACK od serwera. Po tym pakiecie połączenie TCP jest w pełni ustanowione i gotowe do przesyłania danych aplikacji (HTTP).

## **Etap 2: Żądanie HTTP POST i Rozpoczęcie Uploadu**

Po ustanowieniu połączenia TCP, klient może wysłać żądanie HTTP do serwera. Przesyłanie pliku zazwyczaj odbywa się metodą POST.



47 1.824182 127.0.0.1 127.0.0.1 TCP 750 59991 → 3000 [PSH, ACK] Seq=1 Ack=1 Win=65280 Len=706 [TCP PDU reassembled in 59]

Rys. 2.3 Pakiet żądania POST

**Klient wysyła żądanie POST (Pakiet 47):** Klient (127.0.0.1:59991) wysyła pakiet do serwera (127.0.0.1:3000) z ustawionymi flagami PSH (Push) i ACK. Informacja [PSH, ACK] Seq=1 Ack=1 Win=65280 Len=706 oznacza, że flaga PSH sugeruje serwerowi natychmiastowe przekazanie danych aplikacji do przetworzenia. Pakiet zawiera 706 bajtów danych.

### **Szczegóły żądania HTTP (widoczne w zrekonstruowanych danych w Pakiecie 59):**

- 1) POST /upload/Doghi.jpg HTTP/1.1: Metoda POST wskazuje, że chcemy wysłać dane na podany adres zasobu
- 2) Host: 127.0.0.1:3000: Określa docelowy host i port dla żądania

- 3) Content-Length: 91736: Informuje serwer, że ciało żądania (plik) będzie miało rozmiar 91736 bajtów
- 4) Content-Type: application/octet-stream: Określa, że przesyłane dane są binarnym strumieniem

### Etap 3: Przesyłanie Danych Pliku i Kompresja Obrazu (JPEG/JFIF)

Plik "Doghi.jpg" o rozmiarze 91736 bajtów jest zbyt duży, aby zmieścić się w pojedynczym segmencie TCP. Dlatego plik jest dzielony na wiele segmentów i wysyłany sekwencyjnie.

49	1.824359	127.0.0.1	127.0.0.1	TCP	16428 59991 → 3000 [PSH, ACK] Seq=707 Ack=1 Win=65280 Len=16384 [TCP PDU reassembled in 59]
50	1.824407	127.0.0.1	127.0.0.1	TCP	44 3000 → 59991 [ACK] Seq=1 Ack=17091 Win=48384 Len=0
51	1.824523	127.0.0.1	127.0.0.1	TCP	16428 59991 → 3000 [PSH, ACK] Seq=17091 Ack=1 Win=65280 Len=16384 [TCP PDU reassembled in 59]
52	1.824556	127.0.0.1	127.0.0.1	TCP	44 3000 → 59991 [ACK] Seq=1 Ack=33475 Win=65280 Len=0
53	1.824617	127.0.0.1	127.0.0.1	TCP	16428 59991 → 3000 [PSH, ACK] Seq=33475 Ack=1 Win=65280 Len=16384 [TCP PDU reassembled in 59]
54	1.824642	127.0.0.1	127.0.0.1	TCP	44 3000 → 59991 [ACK] Seq=1 Ack=49859 Win=48896 Len=0
55	1.824699	127.0.0.1	127.0.0.1	TCP	16428 59991 → 3000 [PSH, ACK] Seq=49859 Ack=1 Win=65280 Len=16384 [TCP PDU reassembled in 59]
56	1.824723	127.0.0.1	127.0.0.1	TCP	44 3000 → 59991 [ACK] Seq=1 Ack=66243 Win=65280 Len=0
57	1.824773	127.0.0.1	127.0.0.1	TCP	16428 59991 → 3000 [PSH, ACK] Seq=66243 Ack=1 Win=65280 Len=16384 [TCP PDU reassembled in 59]
58	1.824793	127.0.0.1	127.0.0.1	TCP	44 3000 → 59991 [ACK] Seq=1 Ack=82627 Win=48896 Len=0
59	1.824841	127.0.0.1	127.0.0.1	HTTP	9860 POST /upload/Doghi.jpg HTTP/1.1 (JPEG JFIF image)

Rys. 2.4 Pakiety podziału oraz wysłania pliku

Tabl 2.1 Komunikacja podczas przesyłania pliku

```
POST /upload/Doghi.jpg HTTP/1.1
Host: 127.0.0.1:3000
Connection: keep-alive
Content-Length: 91736
sec-ch-ua-platform: "Windows"
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/133.0.0.0 Safari/537.36 OPR/118.0.0.0
sec-ch-ua: "Not(A:Brand";v="99", "Opera";v="118", "Chromium";v="133"
Content-Type: application/octet-stream
sec-ch-ua-mobile: ?0
Accept: */*
Origin: http://127.0.0.1:3000
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://127.0.0.1:3000/
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: pl-PL,pl;q=0.9,en-US;q=0.8,en;q=0.7
Cookie: csrftoken=2bhX5rzSDqh03r0xLmxRvIzD6J4KC91T

.....JFIF.....
.....
.....
.....".....
.....
.....!."1..2ABQ.Rab.qr.#...3....C.....$S...c...4s%...DT.....
.....).....!..1A"Q.2.BRabq.....?.....>q.I.....<F.x.....P
..#..$.v.
[Reszta danych]
q.....C.{.....
..a~.J..A8R3~d.hK.eA.....8S.X.<.Z.M4.9.b>..B..D.@.C.EZB..ct..SB..H.LIB...I...1..6.".....*1.$..
... (0-V].....`u).....tYH.]..u.....+...$..M..
HTTP/1.1 200 OK
Content-Type: text
Date: Mon, 28 Apr 2025 18:52:53 GMT
```

```

Connection: keep-alive
Keep-Alive: timeout=5
Transfer-Encoding: chunked

14
uploaded succesfully
0

GET /list HTTP/1.1
Host: 127.0.0.1:3000
Connection: keep-alive
sec-ch-ua-platform: "Windows"
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/133.0.0.0 Safari/537.36 OPR/118.0.0.0
sec-ch-ua: "Not(A:Brand";v="99", "Opera";v="118", "Chromium";v="133"
sec-ch-ua-mobile: ?0
Accept: */*
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://127.0.0.1:3000/
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: pl-PL,pl;q=0.9,en-US;q=0.8,en;q=0.7
Cookie: csrftoken=2bhX5rzSDqh03r0xLmxRvIzD6J4KC91T

HTTP/1.1 200 OK
Content-Type: application/json
Date: Mon, 28 Apr 2025 18:52:53 GMT
Connection: keep-alive

Keep-Alive: timeout=5

Transfer-Encoding: chunked

51

["Dog.jpeg","Doghi.jpg","index.php","info.txt","misery-tired.gif","vb%20(2).pdf"]

0

```

**Wielokrotne segmenty danych (Pakiety 49, 51, 53, 55, 57, 59):** Kolejne pakiety TCP wysyłane przez klienta zawierają dane pliku z flagami PSH, ACK i dużą długością segmentu (np. Len=16384). Każdy kolejny pakiet ma numer sekwencyjny zwiększony o liczbę bajtów przesłanych w poprzednich segmentach.

**Serwer potwierdza odbiór segmentów:** Serwer regularnie wysyła pakiety ACK do klienta. Numer potwierdzenia w tych pakietach zwiększa się, odzwierciedlając łączną liczbę bajtów danych pliku otrzymanych od klienta.

**Kompresja i kodowanie obrazu (widoczne w zrekonstruowanym Pakiecie 59):** W danych pliku JPEG można zidentyfikować specyficzne znaczniki JFIF (JPEG File Interchange Format):



- 1) **FF D8 (Start of Image):** Marker oznaczający początek pliku obrazu JPEG
- 2) **FF E0 (Application Segment 0 - JFIF):** Zawiera informacje o formacie pliku
- 3) **FF DB (Define Quantization Table):** Zawiera tablice kwantyzacji odpowiedzialne za kompresję stratną
- 4) **FF C0 (Start of Frame - Baseline DCT):** Definiuje parametry obrazu, takie jak wymiary (720x1280 pikseli)
- 5) **FF C4 (Define Huffman Table):** Zawiera tablice Huffmana do kodowania entropijnego
- 6) **FF DA (Start of Scan):** Marker poprzedzający faktyczny strumień skompresowanych danych obrazu
- 7) **FF D9 (End of Image):** Marker oznaczający koniec pliku obrazu JPEG

#### Etap 4: Odpowiedź Serwera HTTP

Po otrzymaniu i przetworzeniu całego żądania POST (czyli całego pliku) serwer wysyła odpowiedź HTTP.

61	1.825739	127.0.0.1	127.0.0.1	HTTP	226 HTTP/1.1 200 OK (text)
62	1.825772	127.0.0.1	127.0.0.1	TCP	44 59991 → 3000 [ACK] Seq=92443 Ack=183 Win=65280 Len=0

Rys. 2.5 Pakiety odpowiedzi serwera o poprawnym przesłaniu pliku

**Serwer wysyła odpowiedź (Pakiet 61):** Serwer wysyła odpowiedź HTTP z kodem statusu 200 OK, oznaczającym pomyślne zakończenie żądania. Odpowiedź zawiera informację "uploaded successfully" oraz listę plików w formacie JSON.

#### Etap 5: Zakończenie Połączenia TCP

Po zakończeniu wymiany danych połączenie jest zamykane, zazwyczaj inicjowane przez serwer.

87	7.836816	127.0.0.1	127.0.0.1	TCP	44 3000 → 59991 [FIN, ACK] Seq=438 Ack=93041 Win=30656 Len=0
88	7.836859	127.0.0.1	127.0.0.1	TCP	44 59991 → 3000 [ACK] Seq=93041 Ack=439 Win=65024 Len=0

Rys. 2.6 Pakiet zakończenia komunikacji

Serwer inicjuje zamknięcie wysyłając pakiet z flagami FIN (Finish) i ACK, a klient potwierdza zakończenie pakietem ACK.

### **3. Metody uruchamiania złośliwych plików przesłanych na serwer**

Niniejszy rozdział prezentuje szczegółowe opisy technik, za pomocą których atakujący uruchamiają już wgrane na serwer złośliwe pliki. Każda metoda została przedstawiona w kontekście poziomu zabezpieczeń środowiska oraz typowych implementacji na serwerach Apache, Nginx, IIS czy w środowiskach skryptowych.

#### **3.1. Środowisko bez zabezpieczeń**

W najprostszych scenariuszach, gdy serwer WWW nie posiada żadnych mechanizmów ograniczających wykonanie kodu, złośliwy plik umieszczony w katalogu webroot staje się bezpośrednio wykonywany przez interpreter. Na przykład w aplikacji PHP każdy plik z rozszerzeniem .php jest automatycznie przekazywany do modułu interpretera (mod\_php lub PHP-FPM).

Atakujący przesyła do katalogu /uploads plik shell.php zawierający kod umożliwiający wykonywanie poleceń systemowych (np. funkcję system() lub exec()). Wywołując adres <https://oficjalna-strona.pl/uploads/shell.php?cmd=ls>, uzyskuje listę plików na serwerze. Analogicznie w aplikacjach opartych na ASP.NET czy Pythonie serwer HTTP kieruje żądanie do odpowiedniego runtime'u, który wykonuje zawartość pliku.

#### **3.2. Filtrowanie rozszerzeń**

Często mechanizmy uploadu ograniczają pliki do wybranych rozszerzeń (np. .jpg, .png, .pdf), jednak takie podejście bywa zawodne. Atakujący może wykorzystać techniki maskowania rozszerzeń, takie jak wielokrotne nazwy plików (shell.php.jpg) czy zmiana wielkości liter (shell.PHP).

Ponadto na serwerach Apache można wgrać plik .htaccess zawierający dyrektywę AddHandler application/x-httpd-php .jpg, co przekaże wszystkie pliki .jpg do interpretera PHP. W rezultacie nawet przy zaakceptowaniu ograniczeń rozszerzeń plik typu shell.jpg będzie wykonany jako skrypt dzięki zmianie reguł serwera.

#### **3.3. Ograniczenia MIME**

W uzupełnieniu do walidacji rozszerzeń część aplikacji weryfikuje również typ MIME przesyłanego pliku, analizując nagłówki Content-Type lub używając bibliotek do odczytu rzeczywistej zawartości. Atakujący może jednak spreparować nagłówek, wysyłając plik PHP z nagłówkiem Content-Type: image/jpeg.

Jeśli aplikacja nie przeprowadza głębokiej walidacji typu, plik zostaje zapisany jako obraz, mimo iż zawiera kod. W momencie wywołania pliku przez przeglądarkę serwer może przekierować go do interpretera PHP, o ile reguły MIME lub konfiguracja serwera nie blokują obsługi PHP w tym katalogu.

### **3.4. Manipulacja konfiguracją przez .htaccess**

Pliki konfiguracyjne .htaccess, dostępne na serwerach Apache w katalogach użytkowników, pozwalają na dynamiczne nadpisanie głównych ustawień serwera. Atakujący może wgrać plik .htaccess do katalogu uploadów z dyrektywami: AddType application/x-httpd-php .jpg .png oraz Options +ExecCGI. Dzięki temu obrazy i pliki multimedialne uzyskują możliwość interpretacji jako skrypty PHP.

Po wgraniu złośliwego pliku shell.jpg z ukrytym kodem PHP i zażądaniu jego wywołania kod zostanie uruchomiony przez interpreter. W środowiskach z restrykcyjnymi ustawieniami (AllowOverride None) metoda ta jest nieskuteczna, jednak w wielu dzielonych hostingach pliki .htaccess mają pełne prawa nadpisywania.

### **3.5. Local File Inclusion (LFI)**

Podatność LFI występuje, gdy aplikacja bez odpowiedniej walidacji wczytuje plik z systemu plików w odpowiedzi na żądanie użytkownika. Atakujący, po wgraniu pliku backdoor.php do katalogu uploadów, odnajduje parametr odpowiedzialny za wczytywanie plików (np. include(\$\_GET['page']));).

Poprzez manipulację ścieżką (?page=../../uploads/backdoor.php) zmusza serwer do wczytania i wykonania kodu PHP znajdującego się poza webrootem. W przypadku Nginx + PHP-FPM czy Apache + mod\_php proces jest identyczny: interpreter zostaje poinstruowany, aby obsłużył wczytany plik, co prowadzi do wykonania dowolnych instrukcji systemowych.

### **3.6. Remote File Inclusion (RFI)**

RFI to rozszerzenie techniki LFI na zasoby zdalne. Gdy w konfiguracji PHP jest włączona opcja allow\_url\_include, funkcje include() i require() mogą akceptować zewnętrzne adresy URL. Atakujący hostuje web shell na zewnętrznym serwerze (np. http://evil.com/shell.txt) i wstrzykuje do podatnego parametru URL tej funkcji.

Serwer pobiera zdalny plik i traktuje go jak lokalny skrypt, wykonując zawarte w nim instrukcje. Ta metoda omija większość lokalnych zabezpieczeń uploadu i umożliwia atakującemu natychmiastowe uruchomienie złośliwego kodu, niezależnie od ustawień katalogu uploadów.

### 3.7. Wykorzystanie wrapperów PHP (php://input, php://filter)

PHP udostępnia tzw. wrappery strumieni, które pozwalają na alternatywne sposoby odczytu i zapisu danych. W ataku polegającym na wykorzystaniu php://input atakujący kieruje żądanie POST do skryptu z parametrami, które nie są w pełni walidowane, a następnie w treści żądania umieszcza kod PHP.

W wywołaniu include('php://input') interpreter odczytuje dane bezpośrednio z wejścia surowego requestu, traktując je jako plik do wykonania. Wrapper php://filter może zostać użyty do odczytania zawartości przesłanego pliku w formie czystego tekstu przed jego wykonaniem.

### 3.8. Niebezpieczna deserializacja

Gdy aplikacja deserializuje obiekty otrzymywane w plikach uploadu, istnieje ryzyko wywołania metod magicznych. Atakujący tworzy obiekt z zaimplementowanymi metodami \_\_wakeup() lub \_\_destruct(), które zawierają kod wykonywalny. Serializuje obiekt do formatu PHP-serialized lub Java .ser i przesyła plik do aplikacji.

Podczas ładowania obiektu z pliku mechanizm deserializacji automatycznie wywołuje wspomniane metody, co prowadzi do wykonania złośliwych operacji. Atak ten jest szczególnie niebezpieczny w środowiskach wykorzystujących złożone frameworki, które automatycznie deserializują obiekty.

### 3.9. Ucieczka z sandboxu i eskalacja w kontenerach

Nowoczesne wdrożenia często wykorzystują konteneryzację (Docker, Kubernetes) lub mechanizmy sandbox. Choć izolują one procesy i ograniczają dostęp do plików systemowych, atakujący może umieścić złośliwy plik wewnątrz dozwolonego woluminu i następnie wykorzystać znane luki w demonach kontenera lub runtime.

Dzięki temu payload może zostać uruchomiony poza granicami kontenera, co oznacza osiągnięcie eskalacji uprawnień i dostęp do całego systemu operacyjnego hosta. W podobny sposób działa atak na słabą konfigurację role-based access control (RBAC) w Kubernetes.

### 3.10. Dynamiczne wykonanie kodu w aplikacjach (eval, refleksja)


W wielu językach dostępne są funkcje pozwalające na dynamiczne tłumaczenie i wykonanie ciągów kodu. Atakujący może stworzyć plik z treścią kodu w JavaScript, Pythonie czy Javie. Po uploadzie pliku aplikacja oferująca mechanizm pluginów lub skryptowania może odczytać jego zawartość i przekazać do eval(), exec() lub ClassLoader.

W Javie dynamiczne ładowanie klas z plików JAR pozwala na wstrzykiwanie nowych komponentów, natomiast w Pythonie funkcja `exec()` może wykonać kod w dowolnym kontekście aplikacji.

## 4. Analiza programu realizującego funkcję serwera plików

### 4.1. Analiza JavaScript w funkcji przycisku serwera

Przycisk przesyłania plików w interfejsie jest zaimplementowany przy użyciu kombinacji HTML i JavaScript. Element `<input type="file">` jest ukryty poprzez klasę CSS "hidden", natomiast stylizowany przycisk jest tworzony przez element `<label>`. Gdy użytkownik klika na etykietę "Upload", przeglądarka automatycznie aktywuje związany z nią element input o id "upload".



```
1 <label for="upload" class="btn btn-raised">Upload</label>
2 <input class="hidden" id="upload" type="file" onchange="uploadFile(event)">
```

Rys. 4.1 Kod realizujący funkcję przycisku po stronie frontendu

Kluczowy element interakcji stanowi atrybut `onchange="uploadFile(event)"`, który wywołuje funkcję JavaScript `uploadFile()` po wybraniu pliku przez użytkownika. Ta funkcja realizuje następujące operacje:

- 1) Pobiera wybrany plik z obiektu zdarzenia (`event.target.files[0]`)
- 2) Wykonuje żądanie HTTP do endpointu `/upload/{nazwa_pliku}`
- 3) Przesyła zawartość pliku jako dane binarne w ciele żądania

Na podstawie analizy kodu serwera funkcja `uploadFile()` prawdopodobnie używa API `Fetch` lub `XMLHttpRequest` do wykonania żądania HTTP typu POST/PUT do ścieżki `/upload/{nazwa_pliku}`.

### 4.2. Jak otwiera się okno wyboru pliku

Proces otwierania okna wyboru pliku przebiega następująco:

**Inicjacja:** Użytkownik klika na element `<label>` ze stylem przycisku, który jest wizualnie widoczny w interfejsie.

**Delegacja zdarzenia:** Przeglądarka wykrywa, że kliknięty element `<label>` jest powiązany z ukrytym elementem `<input type="file">` poprzez atrybut `for="upload"`.

**Aktywacja kontrolki systemowej:** Przeglądarka automatycznie aktywuje natywne okno dialogowe wyboru pliku, które jest kontrolowane przez system operacyjny, a nie przez aplikację internetową.

**Interakcja użytkownika:** Użytkownik wybiera plik w oknie dialogowym systemu operacyjnego.

**Zdarzenie zmiany:** Po zamknięciu okna dialogowego z wybranym plikiem przeglądarka generuje zdarzenie `change` dla elementu `input`, które wywołuje funkcję JavaScript `uploadFile(event)`.

Należy podkreślić, że wygląd i zachowanie okna wyboru pliku jest determinowane przez system operacyjny, a nie przez aplikację internetową, co zapewnia spójność z interfejsem systemu oraz bezpieczeństwo.

## **5. Ograniczenia przesyłania dużych plików w aplikacjach webowych**

### **5.1. Walidacja po stronie klienta**

#### **5.1.1. Sprawdzanie rozmiaru pliku**

Klient może szybko odmówić wysłania pliku, którego rozmiar przekracza ustalony limit (np. 5 MB), co oszczędza czas i pasmo łącza użytkownika.



```


1  <input type="file" id="fileInput" />
2  <script>
3      const MAX_SIZE = 5 * 1024 * 1024; // 5 MB
4      document.getElementById('fileInput')
5          .addEventListener('change', function(e) {
6          const file = e.target.files[0];
7          if (file.size > MAX_SIZE) {
8              alert('Plik jest zbyt duży (max 5 MB).');
9              e.target.value = ''; // wyczyść wybór
10         }
11     });
12 </script>

```

Rys. 5.1 Ograniczenie przesyłania plików przez definicję rozmiaru

### 5.1.2. Ograniczenie typów plików

Multer umożliwia narzucenie limitu rozmiaru (`limits.fileSize`) oraz whitelistę MIME-type'ów (`fileFilter`).



```

1  <input type="file" accept=".jpg,.png,.pdf" />
2

```

Rys. 5.2 Filtrowanie po rozszerzeniu pliku

## 5.2. Walidacja po stronie serwera

### 5.2.1. Node.js + Express + Multer

Multer umożliwia narzucenie limitu rozmiaru (`limits.fileSize`) oraz whitelistę MIME-type'ów (`fileFilter`).

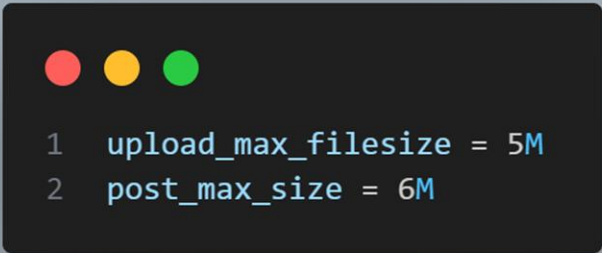


```
1  const multer = require('multer');
2  const upload = multer({
3    limits: { fileSize: 5 * 1024 * 1024 }, // 5 MB
4    fileFilter: (req, file, cb) => {
5      const allowed = ['image/jpeg', 'image/png', 'application/pdf'];
6      cb(null, allowed.includes(file.mimetype));
7    }
8  });
9  app.post('/upload', upload.single('file'), (req, res) => {
10    res.send('Plik zaakceptowany');
11  });
```

Rys. 5.3 Definicja rozmiaru i rodzaju pliku w node.js

### 5.2.2. PHP

W `php.ini` ustawiamy globalne limity:




```
1  upload_max_filesize = 5M
2  post_max_size = 6M
```

Rys. 5.4 Globalne limity w php

Dodatkowo można użyć limitów wewnątrz skryptu PHP:



A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It contains PHP code for file size and type validation.

```
1  if ($_FILES['file']['size'] > 5 * 1024 * 1024) {  
2      die('Przekroczono limit rozmiaru.');
```

```
3  }  
4  $allowed = ['image/jpeg', 'image/png', 'application/pdf'];  
5  if (!in_array($_FILES['file']['type'], $allowed)) {  
6      die('Nieobsługiwany typ pliku.');
```

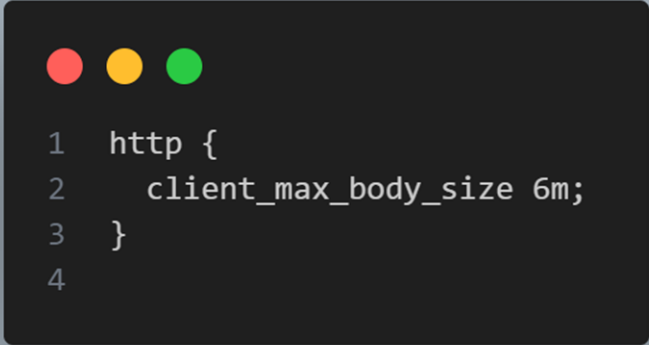
```
7  }  
8
```

Rys. 5.5 Limity wewnątrz skryptu php

### 5.3. Konfiguracja serwera WWW

#### 5.3.1. NGINX

Dyrektywa `client_max_body_size` określa maksymalny rozmiar ciała żądania; przekroczenie powoduje błąd 413.

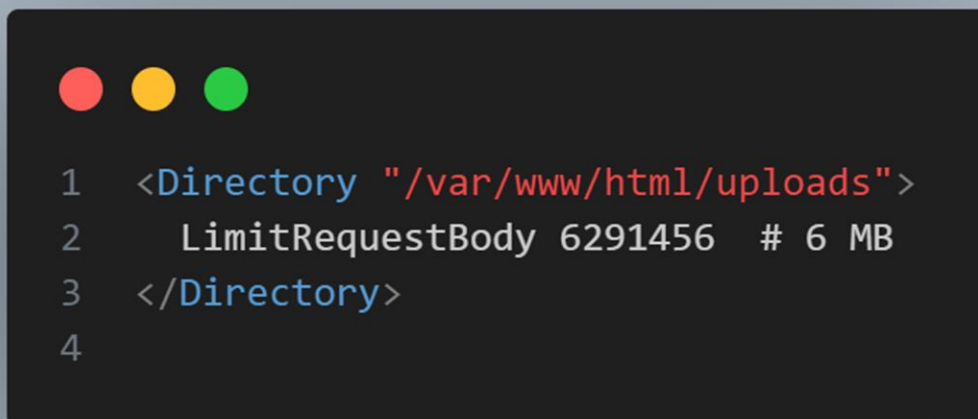
A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It contains Nginx configuration code for client\_max\_body\_size.

```
1  http {  
2      client_max_body_size 6m;  
3  }  
4
```

Rys. 5.6 Konfiguracja maksymalnego ciała żądania.

#### 5.3.2. Apache

Dyrektywa `LimitRequestBody` pozwala ograniczyć rozmiar przesyłanego pliku w bajtach.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. It displays four lines of Apache configuration code in a monospaced font.

```
1 <Directory "/var/www/html/uploads">
2     LimitRequestBody 6291456 # 6 MB
3 </Directory>
4
```

Rys. 5.7 Definicja limitrequestbody w Apache

## 6. Podsumowanie

Niniejsza praca przedstawiła kompleksową analizę zagadnień bezpieczeństwa związanych z funkcjonalnością przesyłania plików w aplikacjach webowych. Głównym celem projektu było zbadanie teoretycznych i praktycznych aspektów ataków typu file upload oraz mechanizmów ich realizacji w środowisku sieciowym.

W ramach przeprowadzonych badań dokonano szczegółowej analizy komunikacji sieciowej podczas procesu przesyłania pliku między klientem a serwerem przy wykorzystaniu narzędzia Wireshark. Analiza ruchu sieciowego pozwoliła na identyfikację kluczowych etapów transmisji, począwszy od ustanowienia połączenia TCP poprzez trójstopniowe uzgadnianie, przez przesłanie żądania HTTP POST z danymi pliku, aż po otrzymanie odpowiedzi serwera i zamknięcie połączenia. Szczególną uwagę poświęcono strukturze przesyłanego pliku JPEG, w tym analizie markerów JFIF oraz mechanizmów kompresji obrazu, co pozwoliło na głębsze zrozumienie procesu transmisji danych binarnych w protokole HTTP.

Druga część pracy koncentrowała się na systematycznej klasyfikacji metod wykorzystywanych przez atakujących do uruchamiania złośliwych plików przesłanych na serwer. Przedstawiono dziesięć głównych technik ataków, począwszy od najprostszych scenariuszy w środowiskach pozbawionych zabezpieczeń, poprzez metody obejścia filtrowania rozszerzeń i ograniczeń MIME, aż po zaawansowane techniki wykorzystujące manipulację konfiguracji serwera, podatności typu Local File Inclusion oraz Remote File Inclusion.

Szczególną uwagę zwrócono na wykorzystanie wrapperów PHP, niebezpieczną deserializację obiektów oraz metody eskalacji uprawnień w środowiskach konteneryzowanych.

Analiza funkcjonalności po stronie klienta obejmowała badanie mechanizmów obsługi elementów HTML typu input file oraz JavaScript odpowiedzialnego za inicjację procesu przesyłania. Wyjaśniono sposób działania delegacji zdarzeń między elementami label a ukrytymi polami input, a także proces aktywacji natywnego okna dialogowego wyboru pliku przez system operacyjny.

Ostatni rozdział pracy przedstawił wielowarstwowe podejście do ograniczania przesyłania dużych plików, obejmujące walidację po stronie klienta, konfigurację serwerów aplikacyjnych oraz ustawienia serwerów [WWW](#). Zaprezentowano konkretne implementacje w różnych technologiach, w tym Node.js z frameworkiem Express i biblioteką Multer, PHP oraz serwerach Apache i Nginx.

Przeprowadzone badania pozwoliły na sformułowanie kompleksowego obrazu zagrożeń związanych z funkcjonalnością przesyłania plików oraz metod ich mitygacji. Praca wykazała, że skuteczne zabezpieczenie tej funkcjonalności wymaga implementacji mechanizmów kontrolnych na wszystkich poziomach architektury aplikacji webowej, od walidacji po stronie klienta, poprzez zabezpieczenia serwerowe, aż po odpowiednią konfigurację infrastruktury sieciowej. Szczególnie istotne okazało się zrozumienie, że żadna pojedyncza metoda zabezpieczenia nie jest wystarczająca, a jedynie wielowarstwowe podejście defense-in-depth może zapewnić odpowiedni poziom bezpieczeństwa przed atakami typu file upload.

## 7. Literatura

1. PortSwigger Ltd, File upload vulnerabilities, Dostęp: 3.06.2025 <https://portswigger.net/web-security/file-upload>
2. Soroush Dalili, Dirk Wetter, Landon Mayo, OWASP, Unrestricted File Upload, Dostęp 3.06.2025 [https://owasp.org/www-community/vulnerabilities/Unrestricted\\_File\\_Upload](https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload)
3. Rackspace Technology, Limit File Upload Size in NGINX, Dostęp: 3.06.2025 <https://docs.rackspace.com/docs/limit-file-upload-size-in-nginx>
4. OWASP Cheat Sheet Series, File Upload Cheat Sheet, Dostęp: 3.06.2025 [https://cheatsheetseries.owasp.org/cheatsheets/File\\_Upload\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html)
5. Apache Commons, Commons FileUpload, <https://commons.apache.org/proper/commons-fileupload/>, Dostęp: 3.06.2025

6. Richard Sharpe, Ed Warnicke, Ulf Lamping, Wireshark User's Guide, [https://www.wireshark.org/docs/wsug\\_html\\_chunked/](https://www.wireshark.org/docs/wsug_html_chunked/) , Dostęp: 3.06.2025
7. blackbird-eu, Insecure file uploads: A complete guide to finding advanced file upload vulnerabilities, <https://www.intigriti.com/researchers/blog/hacking-tools/insecure-file-uploads-a-complete-guide-to-finding-advanced-file-upload-vulnerabilities> , Dostęp: 3.06.2025