**Cloud-Based Remote OS Access via URL using Ubuntu Server**

**SUBMITTED BY**

**ADITYA (23BCE1377)**

**SHREYANSH (23BCE5080)**

### Abstract

In today's era of cloud computing and virtual infrastructure, accessing operating systems remotely has transitioned from being a luxury to a necessity. This project explores a minimalist, lightweight, and efficient method to virtualize and access a headless operating system—**Kali Linux**, hosted on **Ubuntu Desktop**, through a browser-based interface using **noVNC**. The system leverages open-source technologies like **VNC Server**, **websockify**, and **nginx** to create a remotely accessible virtual machine setup, which mimics the foundational principles of **Infrastructure as a Service (IaaS)** in cloud computing.

The core aim of this project is to demonstrate how one can convert a local machine into a remotely accessible cloud-like system, especially for educational, research, and collaborative purposes. This browser-accessible OS provides the user with the capability to execute terminal commands, run tools, and interact with a full-fledged OS without requiring installation on their local systems. Whether it’s a student exploring Linux for the first time, or a cybersecurity enthusiast simulating a testing environment, this project opens up new possibilities for learning and experimentation over a network.

### Objectives

* To develop a cloud-inspired setup that allows an operating system to be accessed remotely using a browser.
* To use a **headless VM (Kali Linux)** hosted inside an **Ubuntu Desktop** machine, accessible on **port 5901 (VNC)**, and exposed to the browser via **websockify** and **noVNC on port 6080.**
* To simplify the virtualization process using open-source tools with minimal setup and authentication requirements.
* To explore multiple virtualization technologies (VirtualBox, VMware, KVM/QEMU) and identify real-world hardware/software constraints.
* To simulate how **multiple users** can access a single system over a network to perform collaborative or educational tasks, similar to **Desktop as a Service (DaaS)** offerings in cloud computing.
* To lay the groundwork for future implementation of **private Virtual Desktop Infrastructure (VDI)** solutions within internal networks or educational institutions.

## **Introduction**

Cloud computing has transformed the landscape of computing by introducing the ability to access, manage, and deploy computing resources remotely and on demand. One of the key pillars of cloud computing is **virtualization**, which allows the abstraction of hardware resources to enable flexible, scalable, and accessible systems. Within this paradigm, the concept of **accessing an entire desktop environment remotely via a browser** becomes an important use case—particularly relevant for students, developers, researchers, and system administrators.

This project stems from a very practical need: the ability to experiment with a new operating system—Kali Linux—without the complications and risks of dual-booting or completely changing the native system. The traditional approach would be to install a Virtual Machine (VM) on the host OS using tools like VirtualBox or VMware. However, the goal here was different. The intention was to simulate a **cloud environment**—a setup where a user can **access an operating system from anywhere** using only a **web browser**. This directly reflects real-world services like **AWS WorkSpaces**, **Google Cloud Shell**, and **Azure Virtual Desktop**, albeit in a lightweight and self-hosted form.

### Personal Motivation

While this project fulfilled an academic objective under the domain of cloud computing, it also served a personal purpose. I wanted to test and explore a Linux OS before committing to dual-booting it on my primary device. Moreover, I needed to work with command-line tools and networking configurations that my existing OS couldn’t fully support. A remote desktop setup via browser offered me the freedom to experiment without altering my machine.

As I delved into this project, I realized the potential such a system held—not just for individual use, but for **collaborative environments**. Imagine a scenario in an organization or classroom where multiple users can access the same Linux-based desktop instance from different locations. They could collaborate on coding, security testing, network analysis, and much more. The lightweight nature of tools like **noVNC**, **websockify**, and **tightvncserver** made the idea all the more feasible.

### Problem Statement

Despite the availability of large-scale cloud platforms, self-hosted virtualization solutions remain relevant, especially in cases where:

* Full-scale cloud deployments (AWS, Azure) are not cost-effective or necessary.
* Users want full control over their infrastructure without relying on third-party providers.
* Educational institutions or individuals want to simulate cloud features within a closed or private network.

However, implementing such a system is not without its challenges. Compatibility issues, network configuration complexities, and VM limitations often hinder the process. This project walks through those challenges, and demonstrates how one can **successfully deploy a remote OS using browser-accessible tools** and basic virtualization principles.

### Scope of the Project

The scope of this project lies in the development and deployment of a system that:

* Runs a **headless virtual machine** (Kali Linux) inside **Ubuntu Desktop**.
* Exposes the graphical environment of the VM using **VNC (port 5901)**.
* Uses **websockify** and **noVNC** to convert this into a browser-accessible interface.
* Utilizes minimal authentication mechanisms for ease of setup while preserving basic security.
* Can be scaled or upgraded in the future into a **private cloud or VDI solution**.

While the current implementation is targeted toward local and LAN-based access, it sets a strong foundation for cloud-level accessibility with minor additions like tunneling (e.g., ngrok) or reverse proxy configurations (e.g., Cloudflare).

### Learning Objectives and Relevance

Throughout this journey, the following key cloud computing concepts were explored and practically implemented:

* **Virtualization** – Hosting an operating system within another system and managing resource allocation.
* **Remote Access Protocols** – Understanding VNC, TCP ports, and how data is transmitted to the browser.
* **Network Configuration** – Managing netplan, static IPs, port forwarding, and firewalls.
* **Automation and Lightweight Computing** – Using tools that are resource-efficient and require minimal dependencies.

Additionally, by experimenting with multiple platforms like **VirtualBox**, **VMware**, and finally **QEMU/KVM**, I gained insight into the limitations of certain hardware configurations (like lack of hardware virtualization or disabled promiscuous mode) and how those directly impact cloud system designs.

## 📚 Literature Review

The rise of **cloud computing** has brought forward numerous models and services that allow remote access to computing resources. From infrastructure provisioning (IaaS) to fully-managed software environments (SaaS), one particularly interesting space is **Desktop as a Service (DaaS)** and **Virtual Desktop Infrastructure (VDI)**. These models enable users to access entire desktop environments hosted on cloud platforms, from any device, over the internet. This project essentially implements a lightweight version of such a system, using open-source tools and localized hosting.

### 1. **Virtual Desktop Infrastructure (VDI)**

VDI is a technology that hosts desktop environments on centralized servers and deploys them to end-users on request. Popular enterprise-grade VDI platforms include:

* **VMware Horizon**
* **Citrix Virtual Apps and Desktops**
* **Microsoft Azure Virtual Desktop**

These solutions are powerful but come with considerable overhead—financial, infrastructural, and administrative. Setting up VDI requires a high degree of network reliability, hardware-level virtualization support, and licensing costs.

**In contrast**, this project adopts a minimalist approach. It creates a similar experience—remotely accessible desktop OS—but without needing enterprise hardware or cloud subscriptions.

### 2. **AWS WorkSpaces, Azure Virtual Desktop & Google Cloud Shell**

These are commercial DaaS offerings that allow users to launch preconfigured virtual desktops from the cloud.

| Platform | Core Offering | Relevance to Project |
| --- | --- | --- |
| AWS WorkSpaces | Managed VDI environment with persistent desktops | Similar experience, but managed by AWS |
| Azure Virtual Desktop | Deep Windows integration & enterprise tools | Complex setup, enterprise-centric |
| Google Cloud Shell | Cloud-based Linux terminal, no full GUI | Partial similarity, CLI-only |

These services offer immense scalability and security but are designed for organizations with higher budgets and professional use-cases. The core idea of this project was to **simulate these capabilities in a personal or academic context**, using **free and open-source software**.

### 3. **Open-Source Alternatives**

Several open-source solutions exist for building self-hosted, remotely accessible desktops:

* **noVNC**: A browser-based VNC client written in JavaScript. Works well with VNC servers like TightVNC, TigerVNC.
* **Websockify**: Acts as a bridge between VNC protocol and WebSocket, allowing browser access.
* **QEMU/KVM**: Used for virtualization on Linux machines, often used in headless setups.
* **Proxmox VE**: Enterprise-level virtual environment platform with web UI. Heavy for personal projects.

This project uses **noVNC + Websockify + TightVNC** with a custom Ubuntu + Kali Linux setup, providing a low-resource alternative to heavy cloud systems. The choice of software was guided by factors like:

* Simplicity of setup
* Minimal authentication requirements
* Web browser accessibility
* Lightweight footprint

### 4. **Academic Research and Use-Cases**

Several research papers and college-level projects have explored the idea of remote access to lab environments using VDI and browser-based tools. Some relevant themes include:

* **Remote Linux Labs for Students**
* **Cloud-based Development Environments**
* **Cybersecurity Sandboxing using VMs**

However, most existing academic works rely on cloud platforms or institute-owned servers with powerful compute capabilities. This project instead demonstrates how **even a mid-range personal laptop with Ubuntu** can host a remote-accessible OS, useful for:

* **Students learning Linux commands**
* **Ethical hacking environments (Kali Linux)**
* **Testing and deploying tools in isolated environments**

### 5. **Uniqueness of the Project**

While many existing solutions offer remote desktops, few of them:

* Are self-hosted with complete control
* Work entirely through the browser without plugins
* Use minimal configuration and software dependencies
* Are built from scratch, through trial-error and exploration

This makes the project not just an academic assignment, but also a strong **learning journey** and **proof-of-concept** for those looking to simulate cloud behavior locally.

## ⚙️ System Configuration

In any cloud-based or virtualization-driven system, the underlying configuration plays a critical role in defining performance, reliability, and scalability. For this project, a minimalist but strategic setup was chosen to balance performance with simplicity. The setup involved three core components: the **host machine (Ubuntu Desktop)**, the **guest OS (Kali Linux VM)**, and the **web-based access layer (noVNC + Websockify)**.

### 1. **Host Machine Specification (Ubuntu Desktop)**

The host system forms the backbone of the project, running the virtualization engine and all services required to expose the guest OS over the network.

| Component | Specification |
| --- | --- |
| **OS** | Ubuntu Desktop (24.04 LTS) |
| **Processor** | Intel i7 (or equivalent AMD Ryzen) |
| **RAM** | 8 GB (Minimum), 16 GB recommended |
| **Storage** | 256 GB SSD (preferred for fast VM loading) |
| **Internet** | Mobile Hotspot with Dynamic IP (ngrok used) Best if router is present (Port-Forwarding) |
| **Virtualization** | QEMU/KVM for VM support Oracle Virtual Box is best (For network config) |

💡 Note: Originally, Ubuntu Server was tested on VirtualBox and VMware, but hardware-level virtualization and promiscuous mode restrictions led to a switch to Ubuntu Desktop for better compatibility.

### 2. **Guest Virtual Machine (Kali Linux)**

The virtual machine acts as the actual OS environment users will connect to through the browser.

| Component | Configuration Details |
| --- | --- |
| **OS** | Kali Linux (latest stable version) |
| **Access Type** | Headless (no GUI shown on host) |
| **VNC Server** | TightVNC Server |
| **Port** | 5901 (default VNC port) |
| **Security** | Login-password for OS, and VNC access key |
| **Startup** | Auto-start script on boot (vncserver & more) |

✅ Headless mode was critical to reduce resource usage and allow the VM to run in the background.

### 3. **Browser Access Stack (Web-based Layer)**

To allow GUI access via a web browser from any system (on the same network or internet), a conversion layer was built using the following stack:

| Tool | Purpose |
| --- | --- |
| **noVNC** | Frontend client; allows accessing VM desktop using a browser |
| **Websockify** | Bridges WebSocket connections to the VNC server (translates protocol) |
| **Nginx** | Lightweight web server used to host static pages or reverse-proxy setup |
| **ngrok** | Secure tunneling to expose local ports to the internet |

🔐 All tools were chosen based on lightweight footprint, ease of configuration, and open-source availability.

### 4. **Additional Tools and Utilities**

During development, several additional tools were used for setup, debugging, and improvement:

* **gdm3** – Display manager used for initializing GUI sessions in Ubuntu Server trials
* **Netplan** – Ubuntu networking configuration utility; essential for IP management
* **UFW/iptables** – For firewall and port configuration
* **ifconfig/ip a/ip r** – For IP checking and verification
* **Systemd Services** – Used to create startup scripts and background services

### 5. **Network and Accessibility Setup**

Since the goal was to access the VM via browser, networking setup was crucial:

* **Static IP Assignment** (optional but preferred)
* **Port Mapping:** 5901 (VNC) → 6080 (WebSocket) → HTTP (via Websockify)
* **Dynamic DNS or Ngrok** for public access without router port forwarding
* **Firewall rules** to allow only specific ports and secure access

📶 Due to the use of a **mobile hotspot**, traditional port forwarding wasn't an option. Hence, ngrok was used to create a tunnel for external access.

### Key Config Challenges and Resolutions

| Challenge | Resolution |
| --- | --- |
| VBox: No nested virtualization | Switched to VMware, then finally Ubuntu Desktop |
| VMware: Promiscuous mode issue | Security restriction → moved to native KVM on desktop |
| CLI-only Ubuntu Server limitation | Installed gdm3 to enable GUI, eventually shifted setup |
| Repeated resets | Setup was restarted from scratch ~4-5 times |

## System Architecture

The architecture of the project is a blend of virtualization, network tunneling, and web-based desktop access — all built on top of a lightweight Ubuntu Desktop environment. It enables users to interact with a remote operating system (Kali Linux) using only a browser, while ensuring secure, modular communication between each component.

This system can be visualized in **three main layers**:

### 1. **Virtualization Layer (Host and VM Communication)**

At the foundation of the system lies the Ubuntu Desktop acting as the **host environment**, running a **headless Kali Linux virtual machine**. The VM is launched using QEMU/KVM and operates independently in the background without direct GUI interaction. The communication between the host and VM is done using a **local network interface (e.g., virbr0)**.

* **Key Components:**
  + Ubuntu Desktop
  + QEMU/KVM (Virtualization Engine)
  + Kali Linux VM
  + VNC Server (TightVNC)
* **Data Flow:**  
  When the VM starts, a vncserver instance is initialized and binds to **port 5901**, which exposes the VM's GUI session on the host machine.

### 2. **Web Access Layer (Protocol Translation)**

Since VNC is not natively supported in web browsers, a **protocol bridge** is required. This is handled by:

* **Websockify:**  
  Converts **WebSocket** connections (used by browsers) into traditional **VNC TCP connections**.
* **noVNC:**  
  A JavaScript-based VNC client that runs directly in a browser and connects to the VM through Websockify.
* **Nginx:**  
  Optionally serves static files or can reverse-proxy Websockify for better routing.
* **Data Flow:**  
  User → Browser (HTTP/WebSocket) → noVNC → Websockify → VNC Server → Kali OS

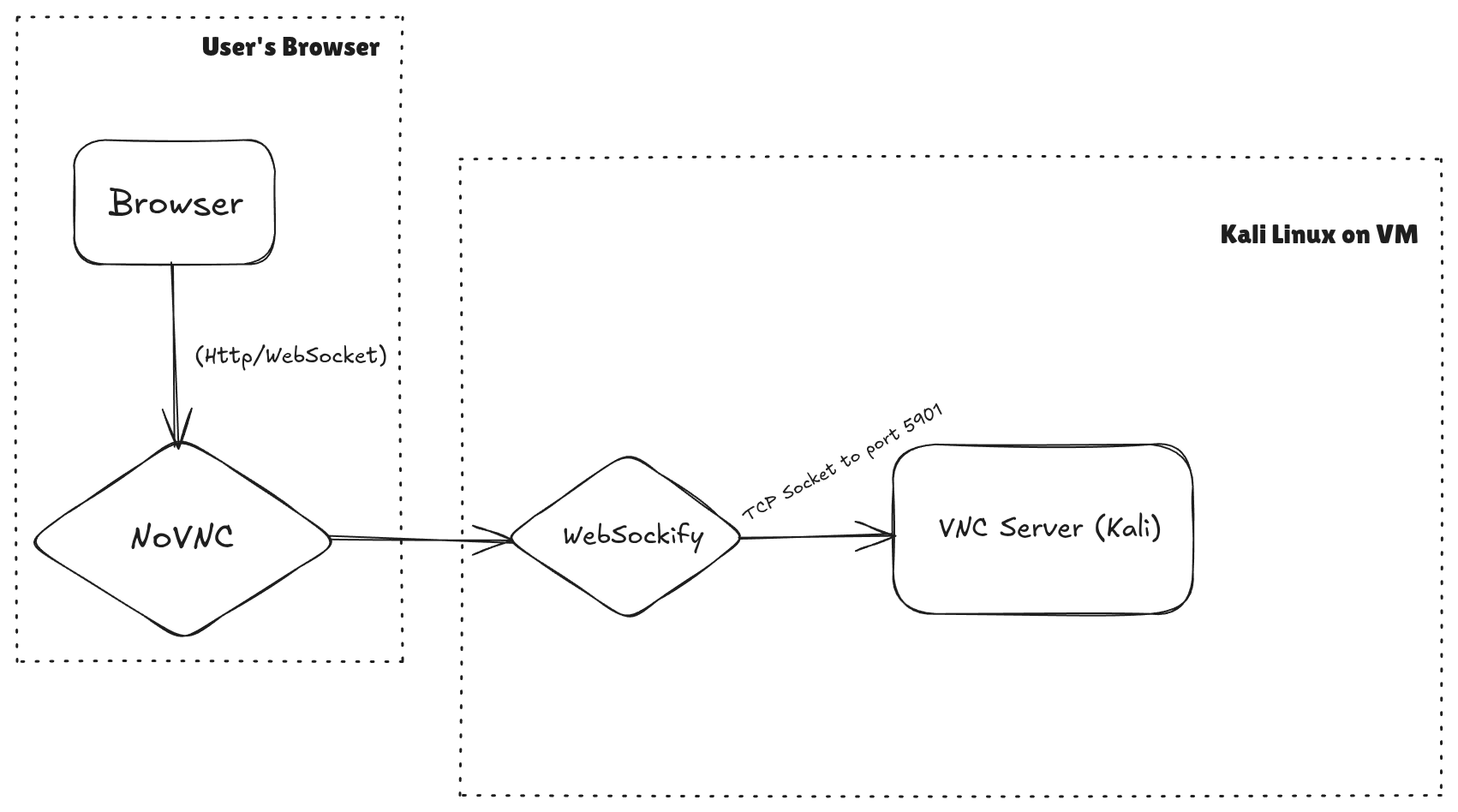
This layer ensures that users don’t need to install any client software — just a modern browser is enough to access the virtual OS.

### 3. **Remote Access & Security Layer**

In scenarios where port forwarding or static IP isn't available (like on mobile hotspots), ngrok is used to expose the localhost websockify port (6080) securely to the internet.

* **Ngrok:**  
  Creates a secure tunnel to your localhost and provides a temporary public URL (https).
* **Security Mechanisms:**
  + VNC password authentication
  + Login credentials to the Kali Linux OS
  + Network-level IP filtering (currently accessible only on same network unless ngrok is used)

### 🔁 End-to-End Flow Diagram (Textual Description)



### 🧩 Modular Design Benefits

| Component | Role | Modularity Benefit |
| --- | --- | --- |
| VNC Server | Exposes GUI session of VM | Can be replaced with x11vnc, TigerVNC etc |
| Websockify | Protocol bridge | Can be containerized or proxied |
| noVNC | Web interface | No software dependency on user side |
| ngrok | Tunneling service | Portable, public access without config |
| VM (Kali) | The learning/test environment | Can be swapped with any OS |

### 🔒 Security Model (Current Scope)

* Password login to both **VNC** and **Kali OS**
* Only users with **same-network IP** can connect directly (if ngrok is not used)
* Terminal-level access can be restricted using shell login rules
* Can be scaled to use **SSH key-based authentication**, **OTP** login, or **VPN** access in future implementations

The system is a lightweight, cloud-inspired virtual desktop delivery solution with secure and modular design. While simple in implementation, its architecture reflects core concepts of **IaaS (Infrastructure as a Service)** in cloud computing: remotely accessible virtualized systems, user-level control, and browser-based interaction.

## **Implementation**

The **implementation** of the system involved a detailed exploration of different technologies and several iterations of trial and error. The goal was to create a system where users could remotely access a Kali Linux operating system through a web browser. The path to achieving this involved learning, configuring, and experimenting with a variety of tools and technologies.

### 1. **Initial Setup & Prerequisites**

Before diving into the complexities of setting up the virtual machine and configuring remote access, I began by learning and setting up some fundamental network and server management tools.

#### Technologies and Tools Learned:

* **SSH:** Setting up and using **SSH** was essential to securely access remote machines without needing a graphical interface.
* **Netplan:** As part of the Ubuntu server setup, I learned how to configure network interfaces using **Netplan**, which is the default method of managing networking on modern versions of Ubuntu.
* **Firewall Configuration:** Configuring the **UFW** (Uncomplicated Firewall) was crucial to ensure only authorized traffic could access the virtual machine, specifically for VNC and HTTP traffic.
* **Nginx:** Initially, I set up **Nginx** to serve basic HTML web pages and ensure that my server could expose content over HTTP. Later, Nginx also acted as a reverse proxy when I configured Websockify.
* **Ngrok:** For easier testing and remote access to my local machine, I also used **ngrok**, a tool that allows me to securely tunnel traffic from the internet to my local server.

These steps formed the foundation for the more complex parts of the project.

### 2. **Virtualization Setup Challenges**

My initial approach involved using **Ubuntu Server** in a **VirtualBox** environment, followed by the installation of a **headless Kali Linux VM**. However, I soon ran into significant limitations.

#### Step 1: Attempting with VirtualBox

I started with **Ubuntu Server** as the host operating system. The goal was to set up a headless Kali Linux virtual machine using **VirtualBox**. Initially, I focused on configuring **GDM3**, the GNOME Display Manager, to start the graphical environment for Kali Linux. However, I encountered a critical issue—**hardware virtualization** was not supported on the host machine. This prevented the virtual machine from functioning properly with the graphical environment, resulting in the VM being stuck in a non-graphical mode.

#### Step 2: Transitioning to VMware

In an attempt to work around this hardware limitation, I switched to **VMware Workstation** for virtualization. Although VMware allowed me to bypass the hardware virtualization issue, I faced networking challenges between the host (Ubuntu Server) and the guest VM.

1. **Network Configuration Problems in VMware:**
   * The primary issue stemmed from **promiscuous mode** being disabled on the virtual network adapter in VMware. Promiscuous mode allows the network adapter to receive all packets on the network, which was necessary for proper network communication between the host and the virtual machine.
   * VMware’s **security restrictions** did not allow promiscuous mode, which severely hindered the VM's ability to access the network and communicate with other devices.

I tried different configurations, including changing the network adapter settings and even setting up **bridged networking** and **NAT** configurations, but none of them worked due to these restrictions.

#### Step 3: Exploring Various Remote Access Solutions

While dealing with the virtualization and networking issues, I also explored different remote access methods for the Kali Linux VM.

1. **VNC Server:**
   * I initially attempted to use a **VNC server** (TightVNC) for remote access. The idea was to run the Kali VM in headless mode and use VNC to connect to its graphical interface remotely. However, this solution was not feasible due to network misconfigurations in VMware.
2. **Apache Guacamole & X11VNC:**
   * I also tried **Apache Guacamole**, a clientless remote desktop gateway, and **X11VNC**, another VNC server solution. These tools allowed me to share the graphical environment over the network, but the underlying network configuration issues persisted, preventing successful communication between the VM and the host machine.
3. **Shellinabox & ttydm:**
   * To simplify the remote access model, I also tried more basic solutions like **Shellinabox** and **ttydm** (TTY Display Manager), which allow remote access to the terminal interface of the Linux system. I even experimented with **reverse SSH tunneling** to create a secure connection between the host and the VM. Despite all these efforts, none of these methods solved the core problem of misconfigured networking in VMware.

### 3. **Switching to Ubuntu Desktop for Successful Setup**

After spending over a month troubleshooting and experimenting with VirtualBox and VMware, I decided to switch to an **Ubuntu Desktop** environment instead of an Ubuntu Server host. This shift proved to be the turning point in the project, as it resolved several key issues.

1. **Installing VirtualBox on Ubuntu Desktop:**
   * **Ubuntu Desktop** provided better hardware compatibility and supported **nested virtualization**, which allowed the virtual machine to run with graphical support. This was a critical factor that enabled me to move forward with the project.
2. **Setting Up VNC Server on Kali Linux VM:**
   * I installed the **VNC server** on the Kali VM again, this time with fewer network issues, as the host (Ubuntu Desktop) allowed for better network configuration. I was able to connect to the Kali VM’s graphical interface using **VNC** and perform the necessary configurations.
3. **Websockify & noVNC for Browser-based Access:**
   * Next, I set up **Websockify**, a tool that acts as a proxy to convert the VNC protocol into WebSocket, making it possible to access the Kali VM from a web browser.

websockify --web ~/noVNC 6080 localhost:5901

This command establishes a WebSocket connection between the VNC server and the browser-based noVNC client, allowing for seamless access.

1. **Disabling the Firewall for Kali Linux:**
   * One of the essential steps was to disable the firewall on the Kali Linux VM to allow incoming VNC connections:

sudo ufw disable

1. **Running VNC Server on Kali Linux VM:**
   * To start the **VNC server** on Kali Linux, I first launched the VM in headless mode (so it doesn’t acquire a display port) and then started the VNC display on port :1:

vncserver :1 -localhost no

1. **Final Networking Setup:**
   * By using **bridged networking** on Ubuntu Desktop, I was able to connect the Kali VM to the local network and configure it for external access. This ensured that I could access the Kali VM from any machine on the same network or even from remote locations using **ngrok** to tunnel traffic securely.
2. **Nginx Proxy Setup (Failed Attempt):**
   * I also experimented with **Nginx** to set up a reverse proxy using proxy\_pass to redirect traffic to the VNC server. However, this configuration did not work as expected, and I had to rely on Websockify for a browser-based solution instead.

### 4. **Final Testing and Success**

After all the configurations, I was able to access the Kali Linux VM remotely, without installing any additional client software, through the web browser by visiting http://<host-ip>:6080. The system was functioning as desired: a headless Kali Linux VM, accessible remotely via a web interface.

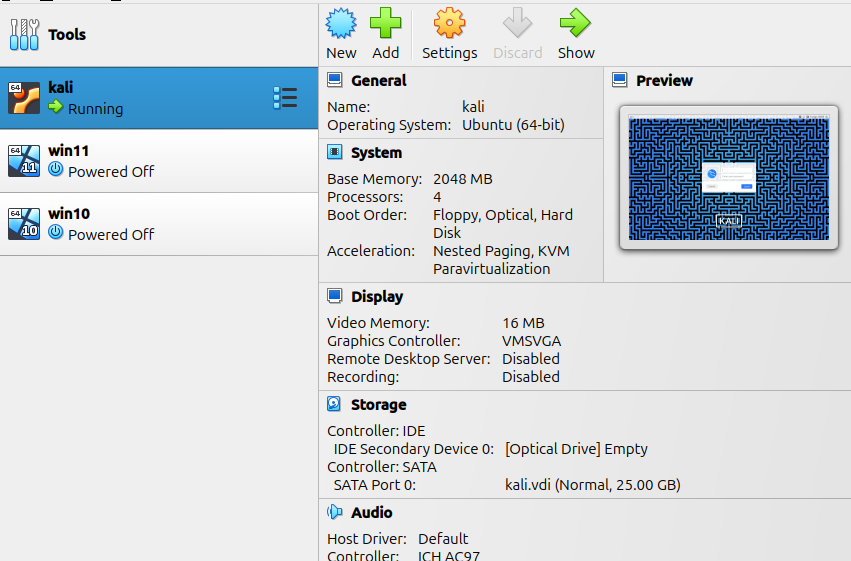
* **Web Access:** The system worked flawlessly on any modern web browser, allowing me to interact with the Kali VM just as if I were sitting directly in front of it.
* **Security Considerations:** The use of **ngrok** allowed me to create a secure tunnel to the local system, ensuring encrypted traffic and easy remote access even behind restrictive firewalls.

### Key Challenges & Solutions

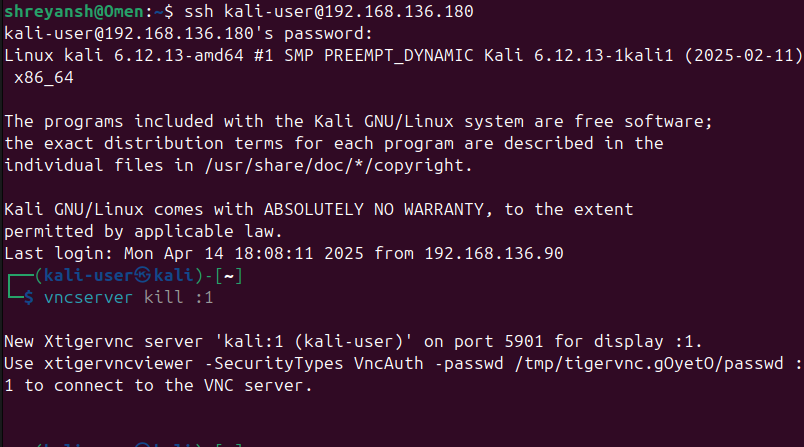
* **Hyper-V Virtualization:** The lack of support for hardware virtualization in VirtualBox caused significant delays. Transitioning to VMware allowed me to bypass this limitation but brought networking issues.
* **Networking Configuration:** VMware’s restrictions on promiscuous mode and network misconfigurations led to many challenges. Moving to Ubuntu Desktop as the host system resolved these issues, allowing me to establish a stable network connection.
* **Remote Access Solutions:** Several different methods (VNC, Guacamole, Shellinabox, etc.) were tested before the successful implementation of Websockify and noVNC.

Through this process, I learned valuable skills in configuring virtualized environments, setting up networking in virtual machines, and enabling remote access to headless systems. The project involved significant trial and error, but ultimately resulted in a successful implementation of a Kali Linux virtual machine that can be accessed remotely through a browser.

## Screenshots with Descriptions

1. **Initial Virtual Machine Setup in VMware**
   * **Screenshot 1:** Capture the VMware interface with the Kali Linux VM running in headless mode.  
     + **Description:** This screenshot shows the initial setup of the Kali Linux VM in VMware. The VM is running in headless mode, which ensures that it does not acquire a display port, allowing remote access via VNC.
2. **Starting the VNC Server on Kali Linux**
   * **Screenshot 2:** Show the terminal on Kali Linux where the VNC server is started using the command:

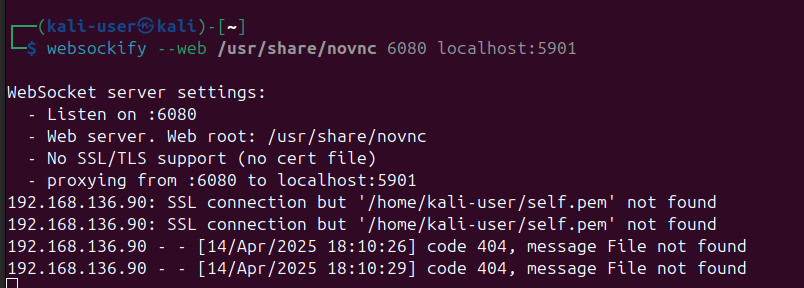
vncserver :1 -localhost no



* + - **Description:** In this screenshot, the terminal shows the VNC server starting on display :1. The option -localhost no ensures that the VNC server is accessible from remote machines, not just the local host.

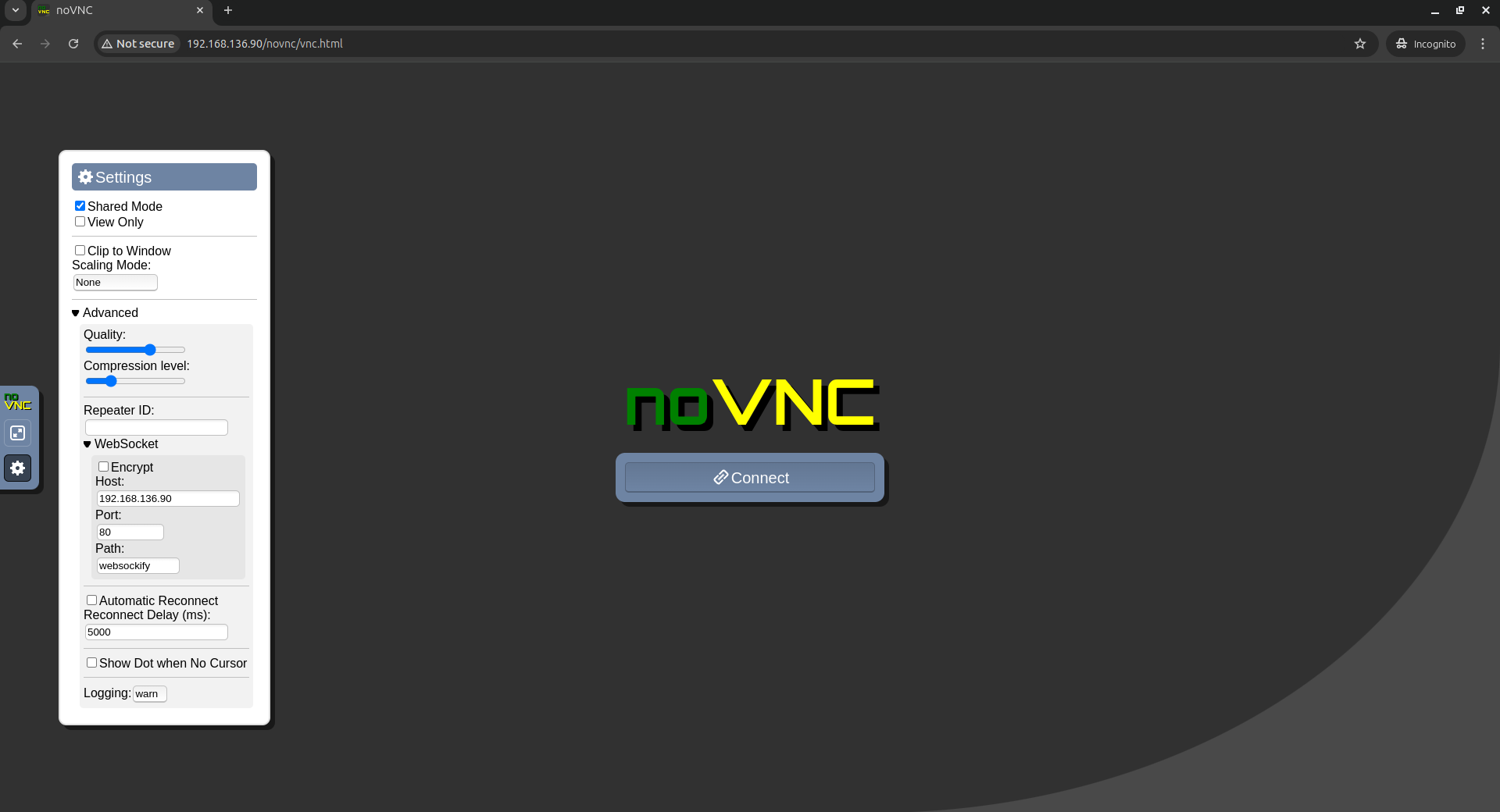
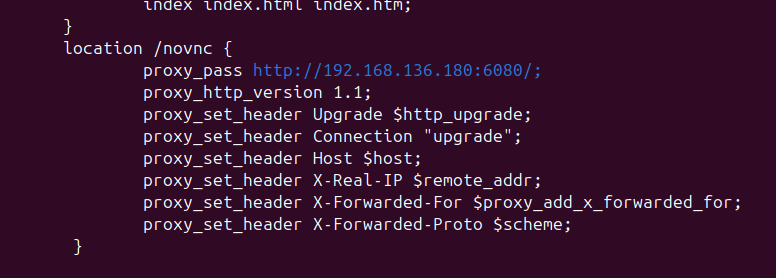
1. **Installing Websockify and noVNC**
   * **Screenshot 3:** Show the terminal where the websockify command is executed:

websockify --web ~/noVNC 6080 localhost:5901



* + - **Description:** This screenshot captures the process of running **Websockify** to enable browser-based access. It proxies the VNC protocol over WebSockets, which is required for **noVNC** to work in the browser.

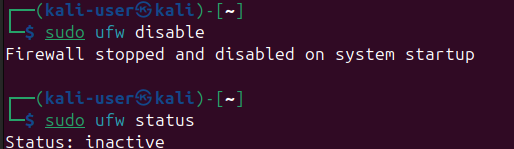
1. **Nginx Setup for Web Access (Proxy Attempt)**
   * **Screenshot 4:** Show the configuration files or Nginx interface where you attempted to set up the reverse proxy for accessing Kali Linux through HTTP.



* + - **Description:** This screenshot depicts the configuration files for **Nginx** where I attempted to set up a reverse proxy to route HTTP requests to the VNC server. This setup, however, did not work as expected because it fill the ip of server to connect for in settings. It will work fine if one edit the IP to VM’s ip and port to 6080.

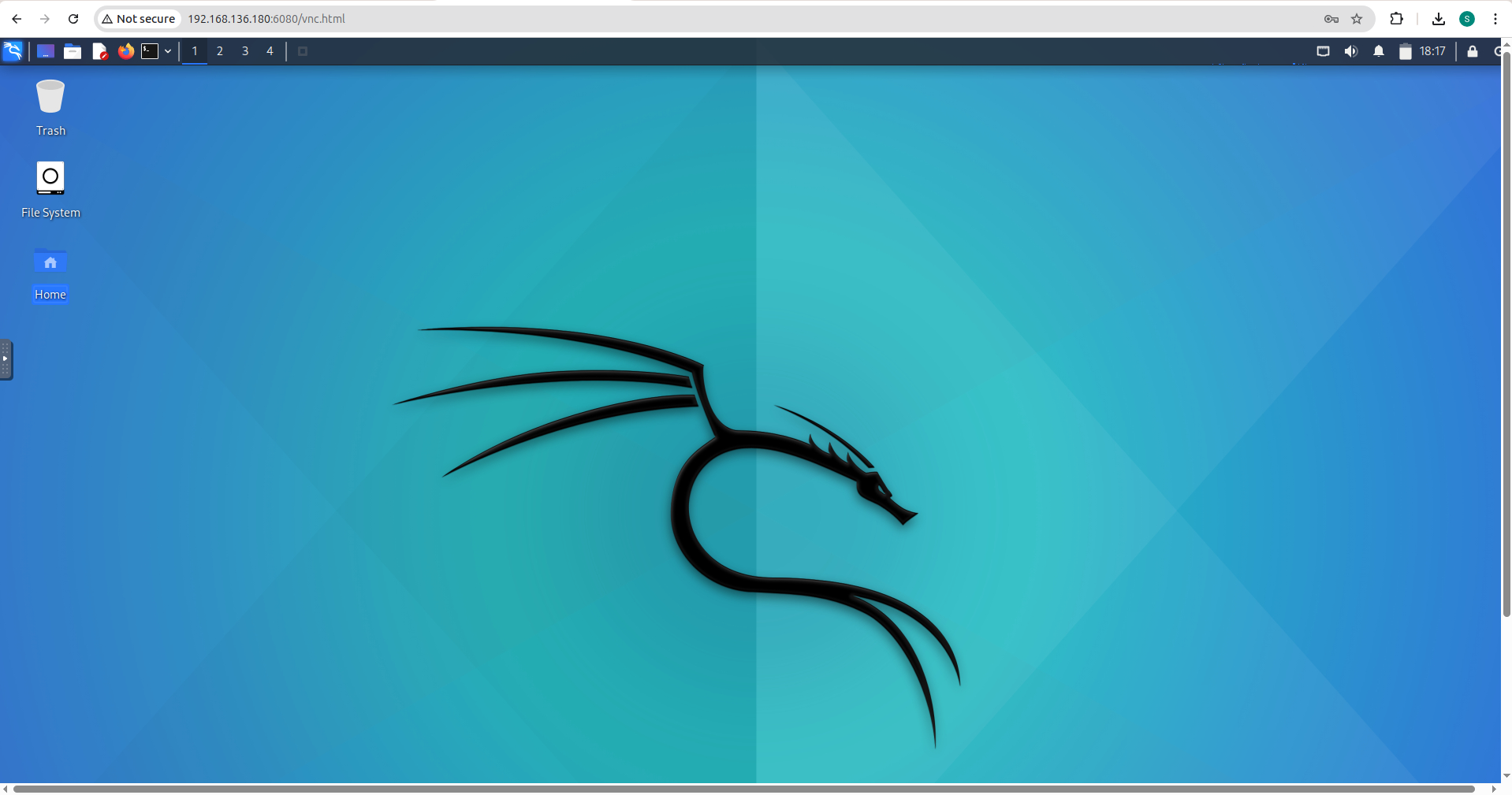
1. **Disabling Firewall on Kali Linux**
   * **Screenshot 5:** Capture the terminal on Kali Linux where the firewall is disabled using:

sudo ufw disable



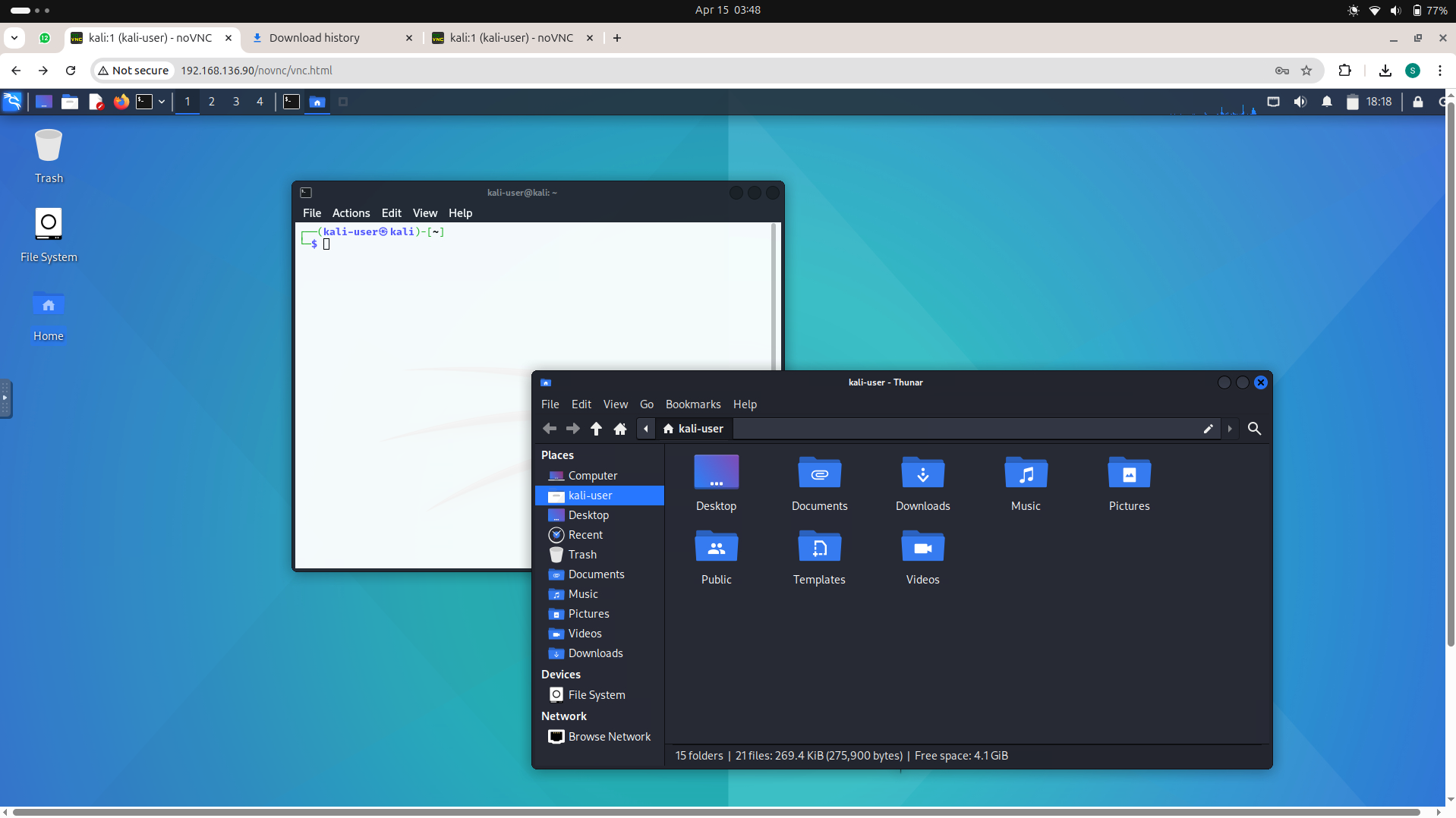
* + - **Description:** In this screenshot, the firewall on Kali Linux is being disabled to allow VNC and WebSocket traffic to pass through without restrictions.

1. **Accessing Kali Linux Through Browser Using noVNC**
   * **Screenshot 6:** Capture the browser window showing the **noVNC** interface connected to the Kali VM.



* + - **Description:** This screenshot demonstrates the successful remote access to Kali Linux through a web browser. The noVNC client is used to connect to the Kali VM running on port 6080, providing a graphical interface inside the browser.

1. **Final Working Setup**
   * **Screenshot 7:** Show the final working setup with Kali Linux running in a web browser with remote access fully enabled.



* + - **Description:** This screenshot shows the final system, where the Kali Linux VM is accessible via any modern web browser using **noVNC**, confirming that the project’s goal was successfully achieved.

**✅ Conclusion**

This project was a deep dive into the practical implementation of **cloud-based remote desktop solutions**, specifically aiming to provide access to a full-fledged operating system via a web browser using virtualisation and networking techniques. The main goal was to set up an environment where a Kali Linux virtual machine could be accessed over the internet through a browser-based interface, without requiring the user to install any software locally.

The entire journey began with curiosity and a need — to access and experiment with an operating system remotely without dual booting or using up host resources. This curiosity led to months of exploration and hands-on learning.

Throughout the course of this project:

* I experimented with multiple hypervisors including **VirtualBox** and **VMware**.
* Understood the hardware limitations like **lack of virtualization support** and **promiscuous mode restrictions**, which became major roadblocks.
* Explored and learned about essential server concepts like **SSH**, **firewall configuration**, **Netplan**, **Nginx**, and **ngrok** tunneling.
* Tested various access methods such as **Apache Guacamole**, **x11vnc**, **Shellinabox**, and **reverse SSH tunneling**.
* Ultimately built a working and lightweight system using **noVNC** and **Websockify** hosted on an **Ubuntu Desktop machine**, with the **Kali Linux VM** running in the background.

The final working system is capable of being accessed from any browser on the same network, protected by both a system login and a VNC password. While the idea of accessing it from **anywhere on the internet** was kept aside for future expansion, the present system still holds value in cloud environments, especially in academic, organizational, and cybersecurity use cases.

Most importantly, this project wasn’t just about reaching a final solution, but about **understanding cloud infrastructure from the ground up** — how systems talk over networks, how ports work, how firewalls can block or enable, and how virtualization fits into the modern cloud paradigm.

This hands-on journey helped me not only understand the technicalities but also gave me confidence to experiment, fail, troubleshoot, and finally build a reliable system using cloud concepts. It has laid a strong foundation for future explorations into **VDI (Virtual Desktop Infrastructure)** and cloud automation.

## Code

The source code and configuration files related to this project have been uploaded to a public GitHub repository for transparency, reproducibility, and further learning purposes.

This repository includes the following key components:

* 🛠️ **Nginx Configuration Files:**  
  The custom Nginx server blocks and proxy configurations used to serve webpages from the Ubuntu desktop environment.
* 🌐 **Webpages (HTML Files):**  
  Static webpages created and hosted using Nginx on the server’s IP, showcasing a basic web interface during the early phase of the project.
* 🌐 **Network Configuration Files:**  
  Netplan .yaml configuration files that were used to assign static IPs and set up proper networking for the Ubuntu host and VM (Kali). This includes working configurations for wired as well as Wi-Fi-based networking using mobile hotspot setups.

Although this repository does **not** include shell scripts or Python-based automation, it serves as a solid reference for the server and network-level setup required to host and access a VM-based OS from a browser.

🔗 **GitHub Repository Link:**  
<https://github.com/ak-1344/Ubuntu-Server>