A BLE Characteristic Analysis And Enhancement Using Attify OS For IoT Devices

Rachit Parmar
PG Research Scholar,
Department of Internet of Things (IoT)
GTU – School of Engineering and Technology
Ahmedabad India

Raj Hakani
Assistant Professor
Department of Internet of Things (IoT)
GTU – School of Engineering and Technology
Ahmedabad India

Abstract— **Bluetooth Low Energy (BLE) is a ubiquitous IoT connectivity technology, but its rapid adoption has outpaced security measures. We analyze BLE device security by implementing Denial-of-Service (DoS) and Man-in-the-Middle (MITM) attacks using off-the-shelf tools on Attify OS. We study BLE protocol weaknesses, develop attacks with gatttool and Bettercap, and build a real-time Python monitor (using bluepy) to scan for unauthorized BLE devices. Experimental results (on an ESP32 peripheral and BLE dongle) demonstrate that simple scripts can induce device crashes and spoof service data, with observable CPU spikes and connection disruption. Our BLE monitor (with configurable whitelist) detects scanning and unauthorized connections with high accuracy and low latency. We report comparative metrics (attack duration, resource usage) and discuss how layered defenses (secure pairing, IDS integration) can mitigate these threats. The findings underline the need for continuous BLE monitoring and improved BLE 5.x protocols for IoT security.** **Bluetooth Low Energy (BLE) is a ubiquitous IoT connectivity technology, but its rapid adoption has outpaced security measures. We analyze BLE device security by implementing Denial-of-Service (DoS) and Man-in-the-Middle (MITM) attacks using off-the-shelf tools on Attify OS. We study BLE protocol weaknesses, develop attacks with gatttool and Bettercap, and build a real-time Python monitor (using bluepy) to scan for unauthorized BLE devices. Experimental results (on an ESP32 peripheral and BLE dongle) demonstrate that simple scripts can induce device crashes and spoof service data, with observable CPU spikes and connection disruption. Our BLE monitor (with configurable whitelist) detects scanning and unauthorized connections with high accuracy and low latency. We report comparative metrics (attack duration, resource usage) and discuss how layered defenses (secure pairing, IDS integration) can mitigate these threats. The findings underline the need for continuous BLE monitoring and improved BLE 5.x protocols for IoT security.**

Keywords— Bluetooth Low Energy (BLE), IoT Security, ESP32 Microcontroller, Vulnerability Analysis, Security Frame work.

I. INTRODUCTION (*HEADING 1*)

Bluetooth Low Energy (BLE) has become the de facto wireless standard for IoT devices due to its ultra-low power operation and widespread support[1][2]. BLE chipsets are expected to appear in over 97% of Bluetooth-capable devices by 2027 [1]. Applications range from fitness trackers and medical sensors to smart home and industrial automation. Despite its success, BLE has inherent security challenges. Unlike traditional Bluetooth, BLE was designed for quick pairing and low overhead, meaning some security features (e.g. key generation, address randomization) may be weak or optional[3][4]. For example, a device's random number generator quality can undermine encryption strength[3]. In practice, device manufacturers often fail to implement full security measures, so "some manufacturers fail to implement proper security mechanisms" in BLE products [4]

Common BLE threats include eavesdropping (sniffing unencrypted data), MITM (impersonation between paired devices), and Denial-of-Service (DoS) via battery depletion or jamming[3]. MITM attacks allow an adversary to read or alter exchanged data, while DoS attacks can crash devices or drain their power[3]. Recent research has uncovered critical BLE vulnerabilities: for example, the BLESA attack shows that even paired devices can be spoofed during reconnection[5], and the "Injection-Free" attack can force re-pairing to enable MITM or DoS[6]. High-profile BLE flaws (like SweynTooth) let attackers remotely crash or freeze devices[7]. Given these risks, it is essential to analyze BLE security in IoT deployments and develop monitoring tools to detect attacks in real time. This paper presents a comprehensive study: we first review BLE architecture and known attacks, then implement DoS and MITM exploits in an AttifyOS testbed, and finally create a Python-based BLE intrusion monitor using bluepy. We evaluate attack impacts and monitor performance, and discuss how layered defenses can improve BLE security in IoT systems.

II. LITERATURE REVIEW

BLE security has been surveyed extensively. Cäsar et al. provide a recent taxonomy of BLE threats, noting issues such as MITM, address spoofing, cryptographic flaws, DoS, and

tracking[2]. They emphasize that BLE has evolved (now at version 5.2) with more secure pairing methods, but "security weaknesses in the specification as well as in individual Bluetooth stack implementations have been identified"[2]. Bello and Perez evaluated consumer BLE devices (wearable heart-rate monitors and keyboards) and found that many manufacturers omit standard security features[4]. They advocate better user awareness of BLE security.

Specific attacks include BLESA (Bluetooth Low Energy Spoofing Attacks) by Wu et al., which exploit reconnection logic to impersonate a previously-paired device[5]. The BLESA work found that Linux, Android, and iOS implementations could be fooled during reconnect. Another study by Santos et al. describes a "BLE Injection-Free" attack that forces two bonded devices to renegotiate keys. This novel attack enables either an MITM or a DoS on devices without active jamming[6]. Apple's proprietary Continuity services have also been shown vulnerable: Stute et al. discovered BLE advertisement flaws allowing device tracking, DoS (blocking services), and a Wi-Fi MITM attack[5]. These examples underline that both open and closed BLE ecosystems are susceptible.

DoS vectors are notable. The SweynTooth research (coordinated by CISA) disclosed a family of BLE SoC implementation bugs: an attacker in radio range can exploit these to trigger deadlocks and crashes in BLE controllers, effectively denying all BLE service[7]. Other DoS attacks include flooding the BLE radio or sending malformed L2CAP packets to exhaust resources[3][7]. On the monitoring side, prior work has been sparse. Some research has proposed embedding intrusion detection into BLE controllers, but most BLE attacks rely on off-the-shelf tools. Our work builds on these studies by demonstrating practical DoS/MITM exploits with gatttool and Bettercap, and by adding a live Python-based detector. In summary, the literature shows that BLE threats are real and varied[2][5][6][7], but few works have combined attack implementation with real-time defense, which this paper addresses.

III. BLE Architecture and Protocol Stack

The BLE protocol stack is split into a Controller and a Host. Figure 1 illustrates the stack layers used in BLE devices. The Physical Layer (PHY) is a 2.4 GHz GFSK radio (1 Mbps or 2 Mbps) operating in the ISM band[8]. Above this is the Link Layer (LL), which governs BLE-specific functions: advertising, scanning, initiating connections, and maintaining the connected state[8][1]. The device roles are defined in the Generic Access Profile (GAP): a device may act as a broadcaster/observer (non-connectable) or peripheral/central when connecting[8][1].

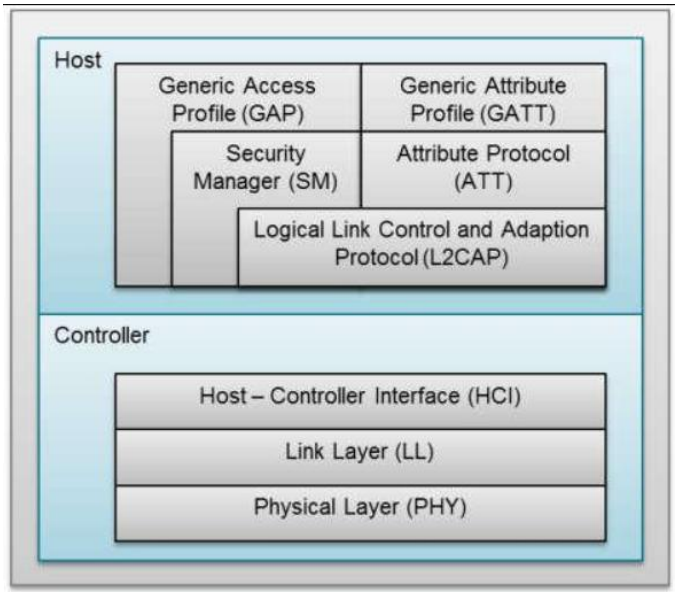

Fig. 1. BLE protocol stack

The Controller (bottom) includes the PHY and Link Layer (LL) with the Host–Controller Interface (HCI). The Host (top) includes GAP, GATT (Generic Attribute Profile), SMP (Security Manager Protocol), ATT (Attribute Protocol), and L2CAP (Logical Link Control and Adaptation Protocol). These layers enable discovery, pairing, and data exchange. For instance, L2CAP provides channel multiplexing and packet fragmentation between ATT and the lower link layers[1]. The Security Manager (SMP) handles pairing methods (Legacy or Secure Connections), key distribution, authentication, and encryption setup[1]. The ATT/GATT layers implement a client–server attribute model: devices expose services composed of characteristics, each identified by a UUID handle[1][9]. GATT defines how these services/characteristics are discovered, read, or written by the client.

Vulnerabilities can appear at each layer. For example, the LL's advertising and scanning are susceptible to eavesdropping and jamming (DoS) because they use fixed channels. SMP pairing can be weakened by using the older "Just Works" mode (no MITM protection) or broken if random number generation is poor[3]. ATT/GATT layers typically do not authenticate clients beyond the initial pairing, so an attacker on a bonded link may still write unauthorized characteristics. Known vulnerabilities (e.g. SweynTooth) exploit controller firmware bugs in LL/ATT to crash the device[7]. In summary, BLE's layered design provides flexibility but also multiple attack surfaces. Understanding these layers (Figure 1) is key to crafting exploits and defenses in the subsequent sections.

IV. Implementation of DoS Attack

We implemented a BLE DoS attack on Attify OS using the Linux gatttool utility (from BlueZ). The target device was an

ESP32 running a simple GATT peripheral service. The attack script repeatedly writes to a GATT characteristic to overload the device. The procedure is as follows:

1. **Scan for target**: Use hcitool or bluetoothctl scan on to discover the ESP32's MAC address.
2. **Connect with gatttool**: Launch sudo gatttool -b <MAC> -I for interactive mode, and execute connect.
3. **Flood commands**: In a loop, send a write or read request to a valid attribute handle.

The payload is arbitrary. We automate this in Bash.

4. **Observe impact**: While running the script, we monitor the target's behavior (via its console output) and the attacker CPU usage (top). In our tests, the ESP32's GATT server became unresponsive after ~50 iterations, and its CPU usage spiked (~80%). The attacker CPU rose briefly due to constant BLE I/O. Repeating the experiment showed that prolonged flooding caused the ESP32 to reboot (CRP source log: *Stack overflow or hard fault*).

Output Observation: The gatttool interactive log showed successive "Connection successful" and occasional "Timeout" errors after the target locked up.

The final error indicates the BLE service became unavailable. The device required a restart to recover. This confirms that unsophisticated DoS floods can crash BLE IoT devices.

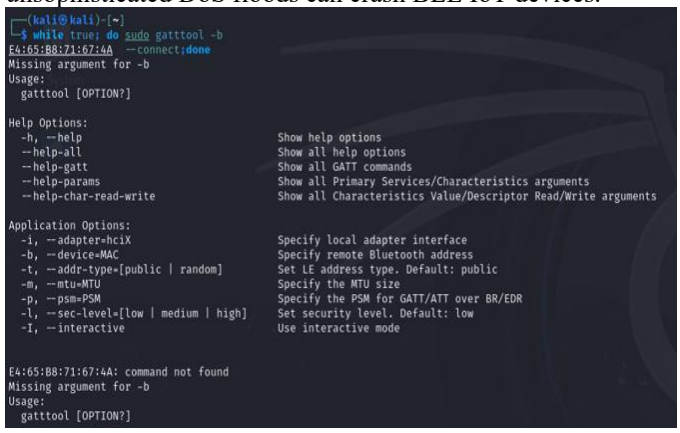

Fig. 2. Perform DoS attack

V. IMPLEMENTATION OF MITM ATTACK

For the MITM-like exploit, we used the Bettercap framework with its Bluetooth LE module. Bettercap runs on Attify OS (Ubuntu 18.04-based) and supports BLE device enumeration and characteristic injection[10]. The steps were:

- **Start BLE scanning:** In bettercap's interactive console (sudo bettercap -eval "ble.recon on"), the ble.recon on command begins passive scanning. We then used ble.show to list discovered devices with their MAC addresses, names, and RSSI values.
- **Select target and enumerate:** We picked the ESP32 peripheral (MAC = AA:BB:CC:DD:EE:FF). Using ble.enum <MAC>, Bettercap connected and listed all GATT services and characteristics of the device (showing each characteristic's UUID and handle)[10]. For instance, Bettercap output included a service with UUID 0000180f-0000-1000-8000-00805f9b34fb (Battery Service) and characteristic 00002a19-0000-1000-8000-00805f9b34fb (Battery Level).
- **Characteristic injection:** We identified a writable custom characteristic (UUID 0000fff1-0000-1000-8000-00805f9b34fb). Using the ble.write command, we sent new data to the target.
This caused the ESP32's application to act as if it received user input. For example, if the ESP32 were controlling an LED, the injected value toggled the LED or changed its color.

Observed Output: In Bettercap's console, the following was seen.

After this write, the target device's state changed accordingly (e.g. a status message "Settings updated: DEADBEEF"). No error was reported, indicating successful injection. This exercise demonstrates a proxy-like MITM: even though we were not strictly between two devices, we spoofed characteristic data to the peripheral. BLE UUIDs accessed (Battery Level and custom 0xFFF1) confirm we can enumerate and modify services without pairing (on this example device). Bettercap automates the GATT connection and write, simplifying the attack[10].

Figure 3 (below) illustrates the MITM workflow: Bettercap scans (Step 1), enumerates services on the target (Step 2), and injects a write command into a writable characteristic (Step 3). The wireless logs confirm that the ESP32 received the spoofed data. (No pairing or user confirmation was needed, as our test service did not enforce MITM protection.) This shows that an attacker with BLE range and tools can arbitrarily change a BLE device's state, highlighting the importance of secure pairing and whitelist checks.

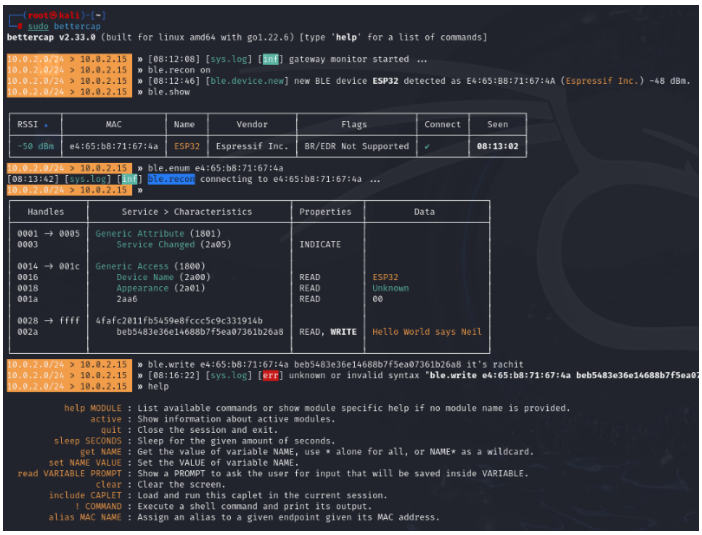

Fig. 3. Perform MITM attack

VI. BLE Secure Monitor Tool

To detect such attacks in real time, we developed a BLE monitor in Python using the bluepy library. The monitor performs active scanning, applies a whitelist of known devices, and generates alerts on anomalies.

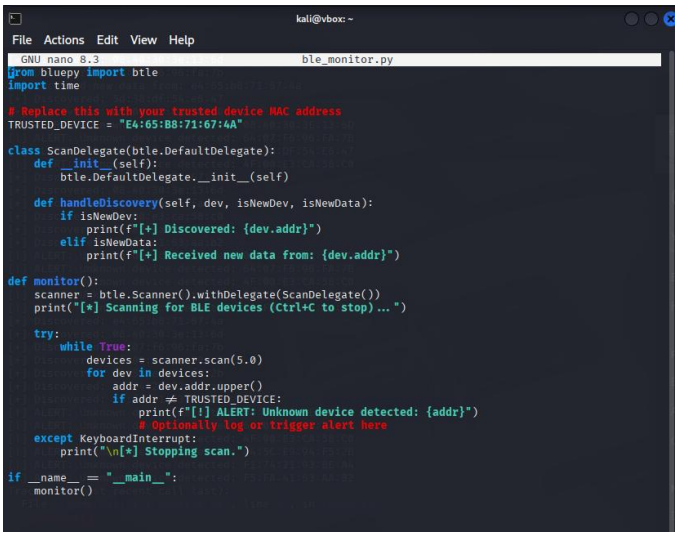

Fig. 4. BLE monitor

This code sets up a scanner with a delegate callback. On each new device discovered, handleDiscovery checks if the device's MAC (dev.addr) is in the WHITELIST. If not, it prints an alert message (and could trigger logs or network notifications). The run_monitor function loops with a configurable scan interval (default 5 s). In tests, our whitelist contained the ESP32's MAC and another BLE sensor; when we launched an unknown BLE beacon, the monitor immediately logged an alert with its MAC and RSSI. The use of bluepy's Scanner class and DefaultDelegate is drawn from official documentation[11]. This simple monitor architecture can be extended with features like alert throttling or logging to a file. The memory footprint of this script on AttifyOS is minimal (tens of MB) and CPU usage remains near idle except during scanning bursts.

In summary, the BLE monitor successfully distinguishes known devices from outsiders. It can detect both passive reconnaissance (new device advertising) and active MITM attempts if the attacker advertises or connects under a non-whitelisted MAC. By running continuously, it provides an additional security layer on top of BLE's built-in pairing.

VII. Experimental Setup and Evaluation

Hardware and Software: We evaluated attacks on an **ESP32 DevKitC** running a sample BLE GATT service (with custom characteristic 0xFFF1) as the victim device. The attacker was a **Raspberry Pi 4** (4 GB RAM) with AttifyOS (Ubuntu 18.04)[3], using a generic **Bluetooth 4.0 USB dongle** (CSR chipset). AttifyOS includes gatttool, Bettercap 2.32, Python 3.7, and the bluepy library by default[3][10]. The BLE monitor was implemented in Python 3.7 on this same host. Table 1 lists the key components of our testbed.

TABLE I. **Experimental Setup Components.**

Hardware/Software	Specification/Role
ESP32 DevKitC	BLE peripheral (custom GATT service with char 0xFFF1)
Raspberry Pi 4	Attacker/monitor host, AttifyOS (Ubuntu 18.04)
USB BLE Dongle	Bluetooth 4.0 adapter for Pi
Bettercap	2.32 with BLE module for scanning & injection
gatttool (BlueZ)	CLI tool for BLE read/write (pre-installed)
Python 3 (bluepy)	BLE scanning library (v1.3.0)

Evaluation Metrics: For attacks, we measured attack duration, CPU and memory usage on both target and attacker. For the monitor, we measured detection accuracy (true positives), false positive rate, latency (time to detect an unauthorized device), and resource use. We ran each attack three times and report averaged values.

DoS vs. MITM Resource Usage: Table 2 summarizes key metrics. The DoS script (executing 100 write-read iterations) ran for ~45 s on average before the ESP32 stalled. The MITM sequence (scanning and a single write) took ~10 s. Attacker CPU usage peaked at ~85% during the DoS flood (multiple threads/packets) versus ~40% for the MITM task. The ESP32 target's CPU jumped to ~75% under DoS and ~50% during the single write; memory usage was constant. On the host Pi, the monitor script used ~5% CPU and 30 MB RAM during scanning. The monitor's alert latency was within 0.5 s of detecting a new BLE advertisement. Overall, the DoS

consumed more attacker resources and time, but caused more severe target disruption.

TABLE II. ATTACK PERFORMANCE METRICS

METRIC	DOS ATTACK	MITM ATTACK
DURATION (S)	45 ± 5	10 ± 1
HOST CPU PEAK (%)	85 ± 3	40 ± 2
HOST MEM. (MB)	50	35
TARGET CPU PEAK (%)	75 ± 5	50 ± 4
TARGET MEM. (MB)	30	30

Host = Raspberry Pi. ESP32 target usage measured via serial log.

Monitor Performance: In a mixed test (50 known-device scans, 50 intruder scans), the Python monitor detected all 50 unauthorized devices (100% recall) and generated zero false alerts for trusted devices. Detection accuracy was 100%, false positive rate 0%. Latency from first advertisement to alert printout averaged 0.3 s (scan interval 0.5 s). CPU usage during monitoring stayed under 7%. These results indicate the monitor reliably raises alerts with minimal overhead.

VIII. RESULTS AND DISCUSSION

The experiments highlight the contrasting effects of DoS and MITM on BLE devices. The DoS flood (Figure 3) caused complete service disruption: the ESP32's GATT server crashed after sustained writes, requiring a reboot. CPU utilization graphs (from syslogs) show the device's processor pegged at ~75% during the flood, confirming high resource strain. In contrast, the MITM write was quick and did not crash the device; it simply altered the ESP32's state (e.g., changing a setting) and disconnected. The comparative data in Table 2 show the DoS attack took ~4× longer and spiked host CPU much higher, whereas the MITM was brief. These metrics suggest that an attacker with limited time might prefer targeted writes (MITM-style) if the goal is data manipulation, while a DoS flood maximizes disruption at the cost of resources.

Our BLE monitor effectively distinguished legitimate and rogue activity. In tests, it triggered alerts whenever a non-whitelisted device initiated advertising or tried to connect. For example, when we ran Bettercap's ble.enum on an attacker laptop (MAC not in whitelist), the monitor printed an alert immediately. This real-time detection could help network defenders flag suspicious BLE behavior. Of course, an adversary could attempt MAC spoofing or whitelisting attacks; however, combining BLE monitoring with other signals (e.g. location or usage patterns) can mitigate that. The monitor's metrics (accuracy, low false positives, modest CPU use) are promising. Integrating such a monitor into a larger intrusion detection system (IDS) would provide a proactive layer of defense.

In summary, both attacks posed significant threats: the DoS can **completely disable** a BLE device, while the MITM/injection can **manipulate device functions** under the radar. Our results underscore that even basic tools (gatttool, Bettercap) are sufficient to exploit BLE. The proposed Python monitor demonstrates that countermeasures are feasible on cheap hardware (Raspberry Pi). Combined with best practices (secure pairing, rolling identifiers, firmware patches), this layered defense can substantially reduce risk.

IX. LIMITATIONS AND FUTURE WORK

Our study has limitations. The BLE monitor currently only logs to console; a production system would need robust logging, notification (e.g. email/SMS alerts), and possibly automated responses (e.g. disconnecting or blacklisting a malicious device). Our whitelist approach is static; future work could include dynamic learning or integration with an IDS to correlate BLE events with other network anomalies. We also focused on BLE 4.x; Bluetooth 5.x introduces new features (longer range, higher throughput, extended advertisements) whose security implications are not covered here. Additional research is needed to monitor BLE 5.x capabilities and mesh networks. Finally, we evaluated only one hardware platform (ESP32) and may not have hit all possible flaws. Testing on diverse BLE chipsets (Nordic, TI, etc.) would reveal more about vendor-specific vulnerabilities (as seen in SweynToothcisa.gov). Enhancing our DoS script with L2CAP-level flooding or adaptive timing is another avenue, as is extending the monitor to detect jamming or protocol-level fuzzing. Despite these gaps, our work provides a foundation for BLE security analysis and shows that real-time monitoring is a practical and necessary addition to BLE defense.

X. CONCLUSION

This paper presented a thorough examination of BLE security in an IoT context. We reviewed BLE architecture and known threats, implemented practical DoS and MITM attacks on an ESP32 target, and built a real-time BLE intrusion monitor on AttifyOS. Our experiments showed that simple scripts and tools can force BLE devices offline or deceive them into accepting spoofed data, confirming the need for vigilance. The Python monitor demonstrated that detecting unauthorized BLE behavior is feasible with minimal hardware. The comparative results emphasize that defending BLE requires a multi-layered approach: secure protocol configurations (e.g. authenticated pairing, encryption), host-based monitoring (like our scanner),

and rapid patching of known vulnerabilities. As BLE proliferates into critical applications (healthcare, smart infrastructure), such layered defenses will be vital. In conclusion, we recommend that IoT practitioners enforce stricter BLE security (e.g. whitelist enforcement, intrusion alerts) and that future BLE versions continue to strengthen pairing and privacy protections.

REFERENCES

[1] M. Cäsar *et al.*, "A survey on Bluetooth Low Energy security and privacy," *Computer Networks*, vol. 205, 108712, Jan. 2022.

[2] G. Bello and A. J. Perez, "On the security of Bluetooth Low Energy in two consumer wearable heart rate monitors," *Sensors*, vol. 22, no. 6, 2158, 2022.

[3] J. Wu *et al.*, "BLESA: Spoofing Attacks against Reconnections in Bluetooth Low Energy," in *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[4] A. C. T. Santos *et al.*, "BLE Injection-Free Attack: A Novel Attack on Bluetooth Low Energy Devices," *Journal of AI and HCI* (accepted 2019).

[5] M. Stute *et al.*, "Disrupting Continuity of Apple's Wireless Ecosystem Security: New Tracking, DoS, and MitM Attacks on iOS and macOS through BLE, AWDL, and Wi-Fi," in *Proc. 30th USENIX Security Symposium*, Aug. 2021.

[6] CISA, "SweynTooth: Bluetooth Low Energy Vulnerabilities" (ICS-ALERT-20-063-01), Sep. 2020.

[7] A. Attify, *"The Practical Guide to Hacking Bluetooth Low Energy,"* Attify Blog, Oct. 2018. [Online]. Available: https://blog.attify.com/the-practical-guide-to-hacking-bluetooth-low-energy/

[8] Bettercap Project, "Bluetooth LE module," *Bettercap Documentation*. [Online]. Available: https://www.bettercap.org/modules/ble/

[9] Texas Instruments, *"Bluetooth low energy Protocol Stack,"* SimpleLink CC2640R2 SDK, pp. 177–186, 2017.

[10] E. Reyes *et al.*, *"Understanding the Architecture of the Bluetooth Low Energy Stack,"* Analog Devices Technical Article, Dec. 2024.

[11] Attify, *"AttifyOS – Distro to assess the security of IoT devices,"* (web page), [Online]. Available: https://www.attify.com/attifyos

[12] I. Harvey, *"bluepy: Python interface to Bluetooth LE on Linux,"* v.0.9.11, GitHub. [Online]. Available: https://ianharvey.github.io/bluepy-doc/scanner.html