black hat
ASIA 2023

MAY 11-12

BRIEFINGS

Sweet Dreams:
Abusing Sleep Mode to Break Wi-Fi Encryption & Disrupt WPA2/3 Networks

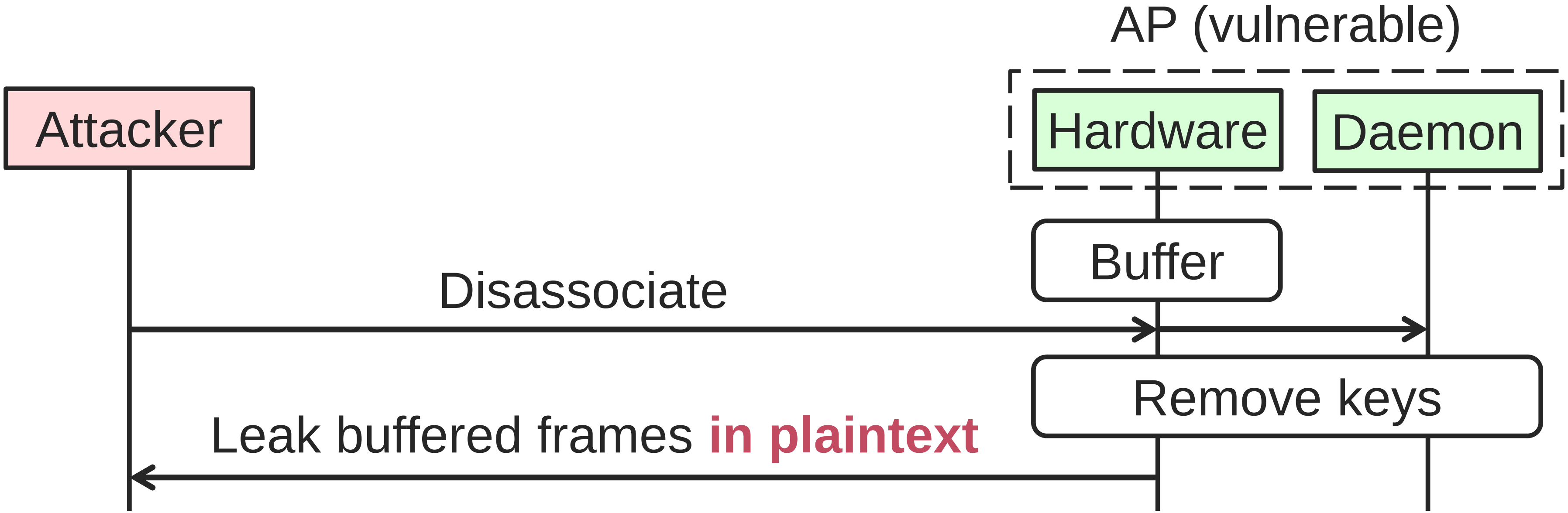
Mathy Vanhoef, Domien Schepers,
and Aanjhan Ranganathan

KU LEUVEN DistriNet Northeastern University

History of Wi-Fi

- WEP (1999): quickly broken [FMS01]
- WPA1/2 (~2003)
  - Offline password brute-force
  - **KRACK** & **Kraken** [VP17,VP18]
- WPA3 (2018):
  - **Dragonblood** side-channels [VR20]

Background: Kr00k implementation flaw



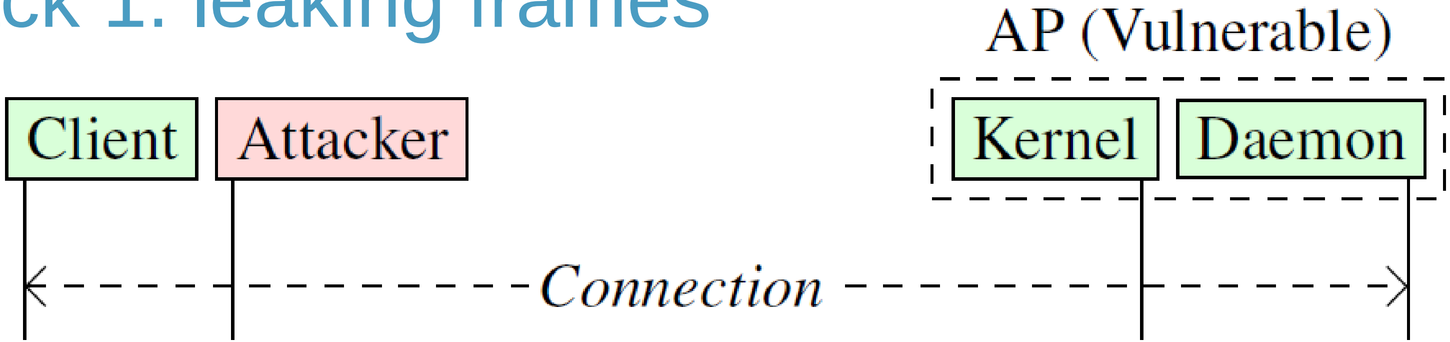
Question: **how are “security contexts” managed**?

New attack 1: leaking frames

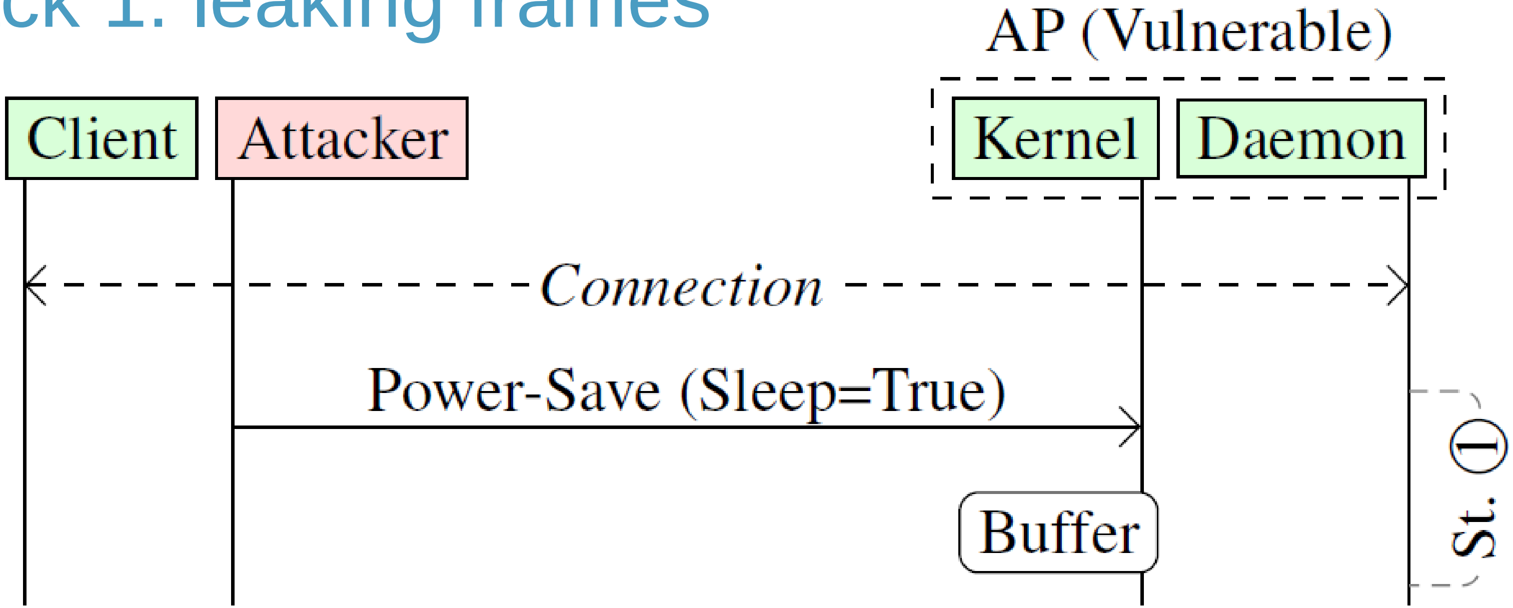
Attack 1: leaking frames

AP (Vulnerable)
Client
Attacker
Kernel
Daemon

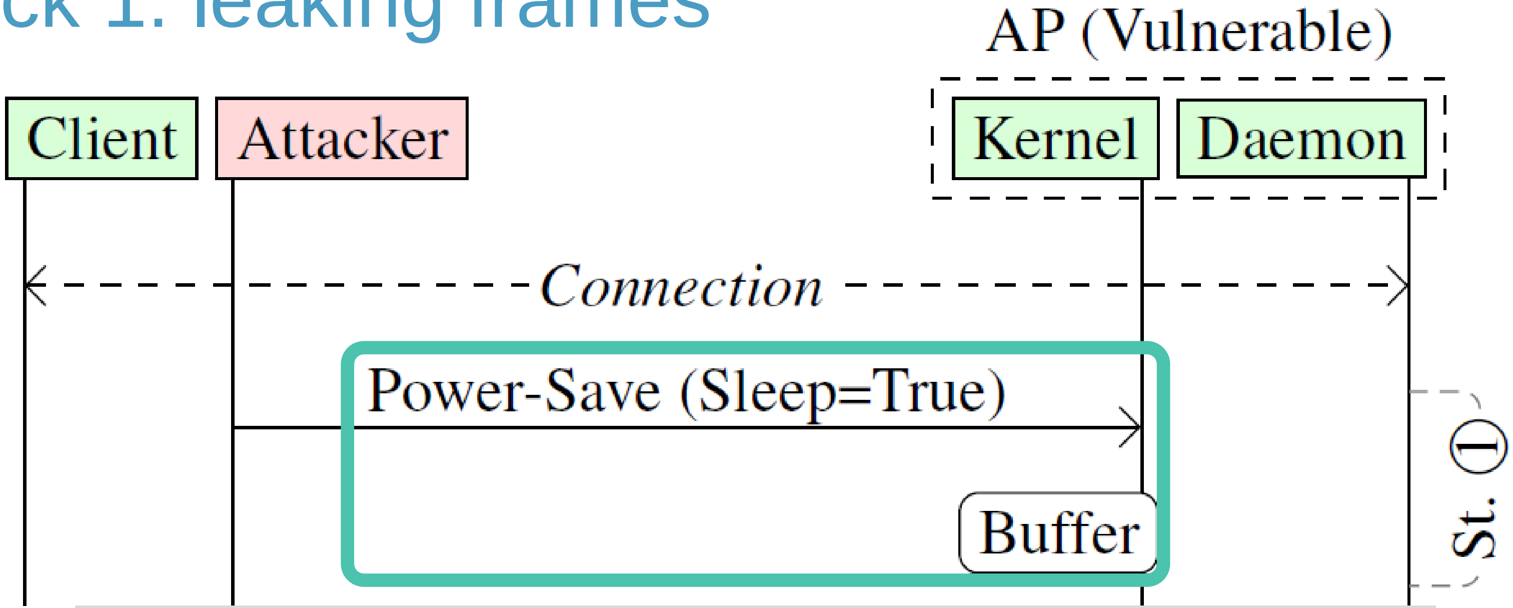
Attack 1: leaking frames



Attack 1: leaking frames

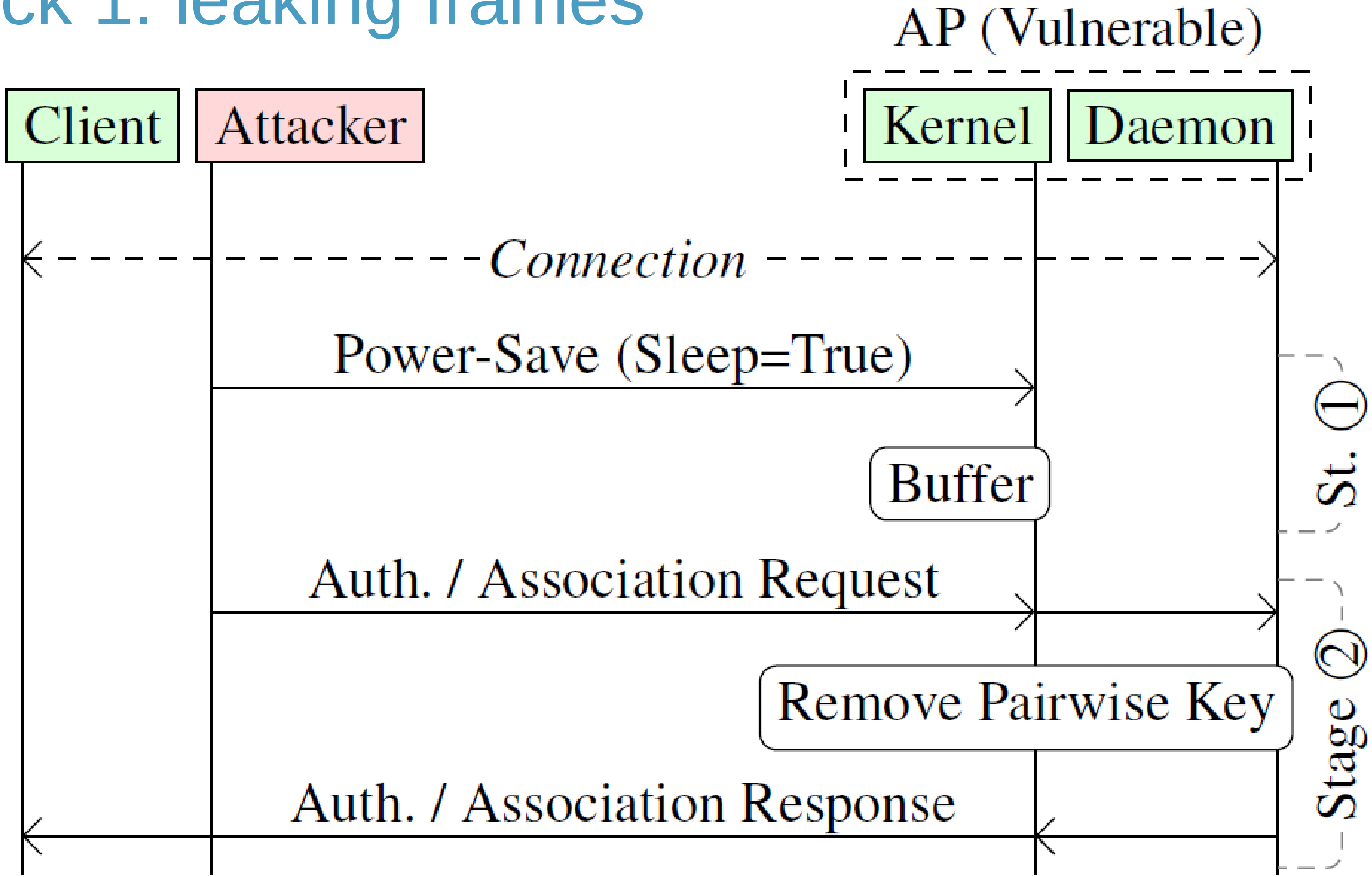


Attack 1: leaking frames

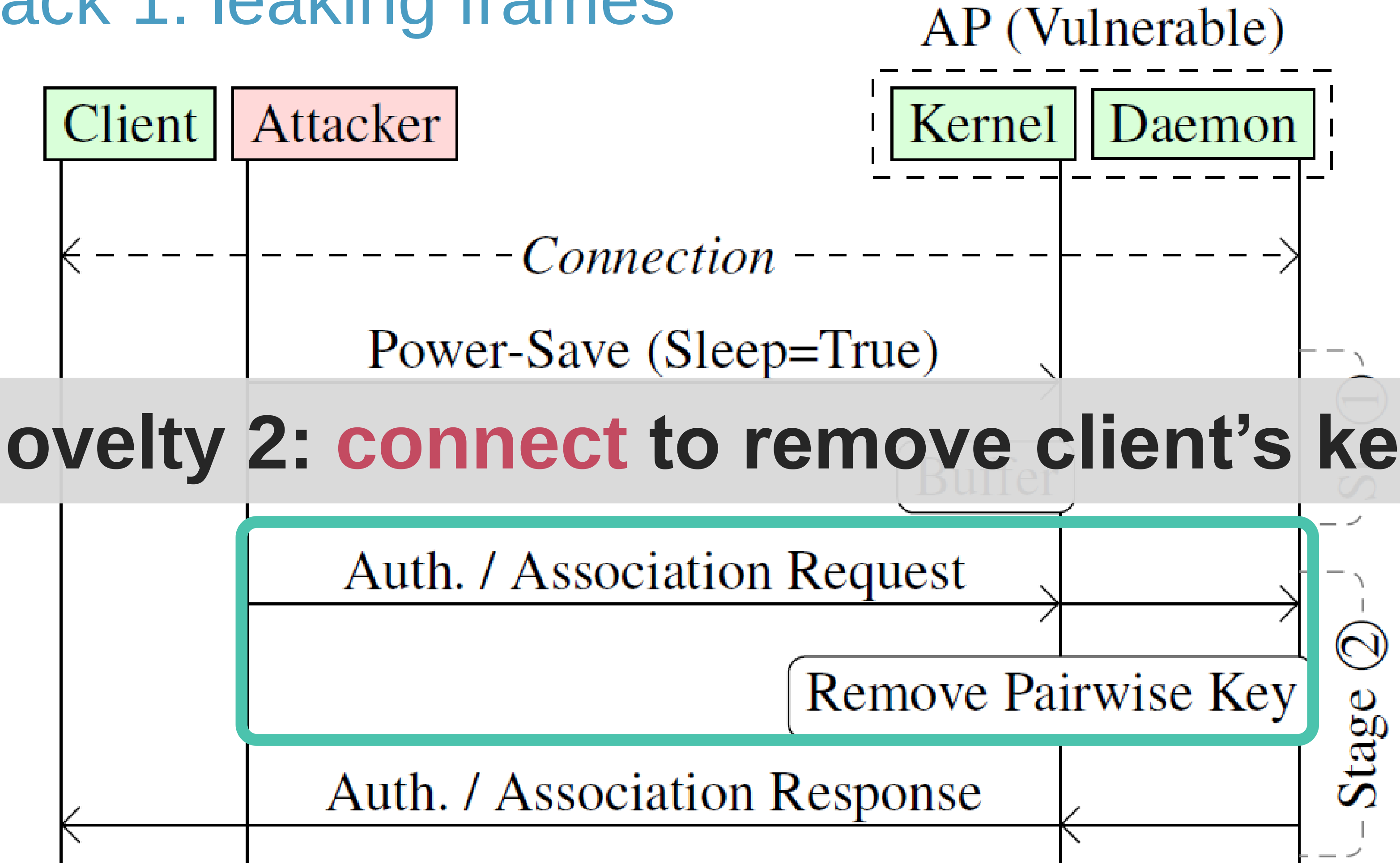


Novelty 1: controlled buffering

Attack 1: leaking frames



Attack 1: leaking frames



AP (Vulnerable)

Client

Attacker

Kernel

Daemon

Connection

Power-Save (Sleep=True)

Buffer

Stage ①

Auth. / Association Request

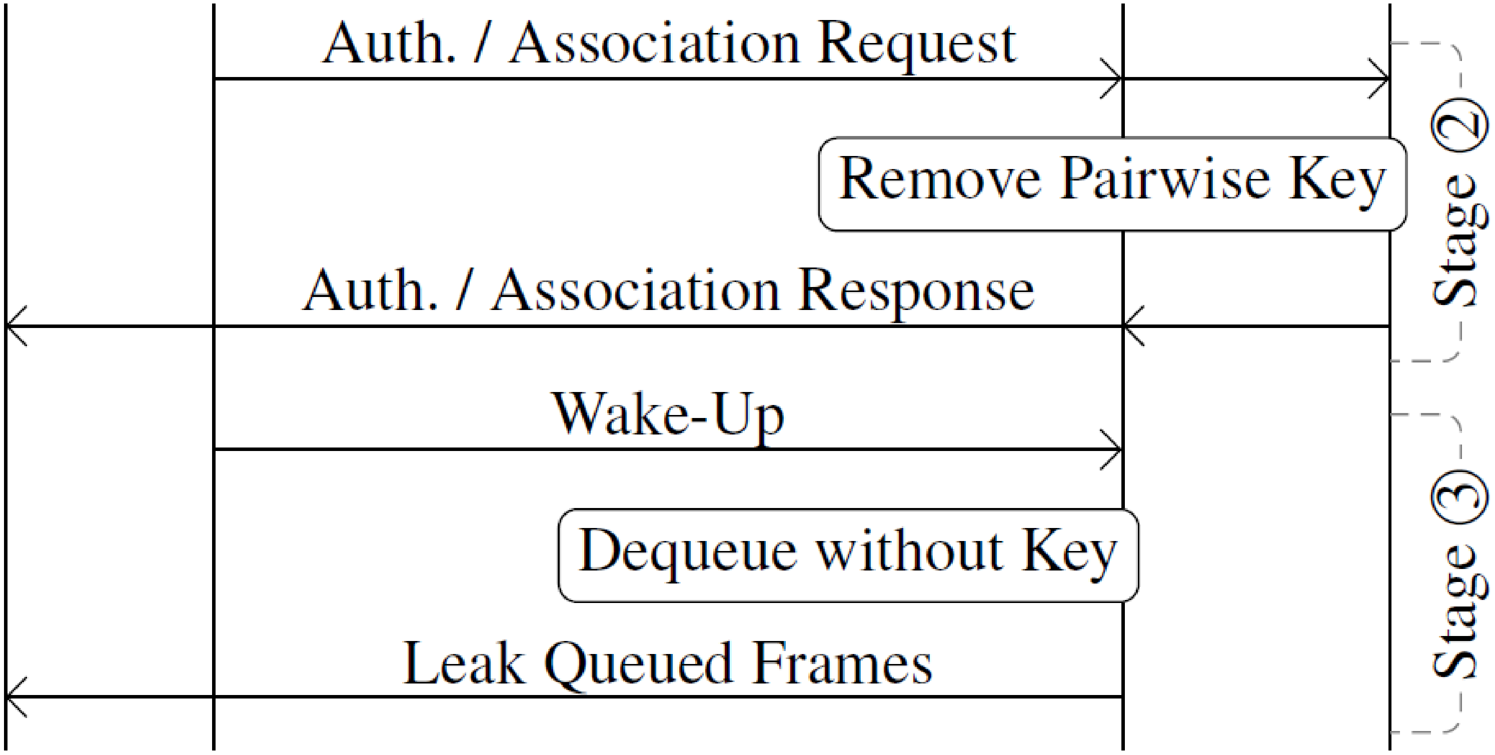
Remove Pairwise Key

Stage ②

Auth. / Association Response

Novelty 2: connect to remove client’s keys

Attack 1: leaking frames



Auth. / Association Request

Remove Pairwise Key

Auth. / Association Response

Wake-Up

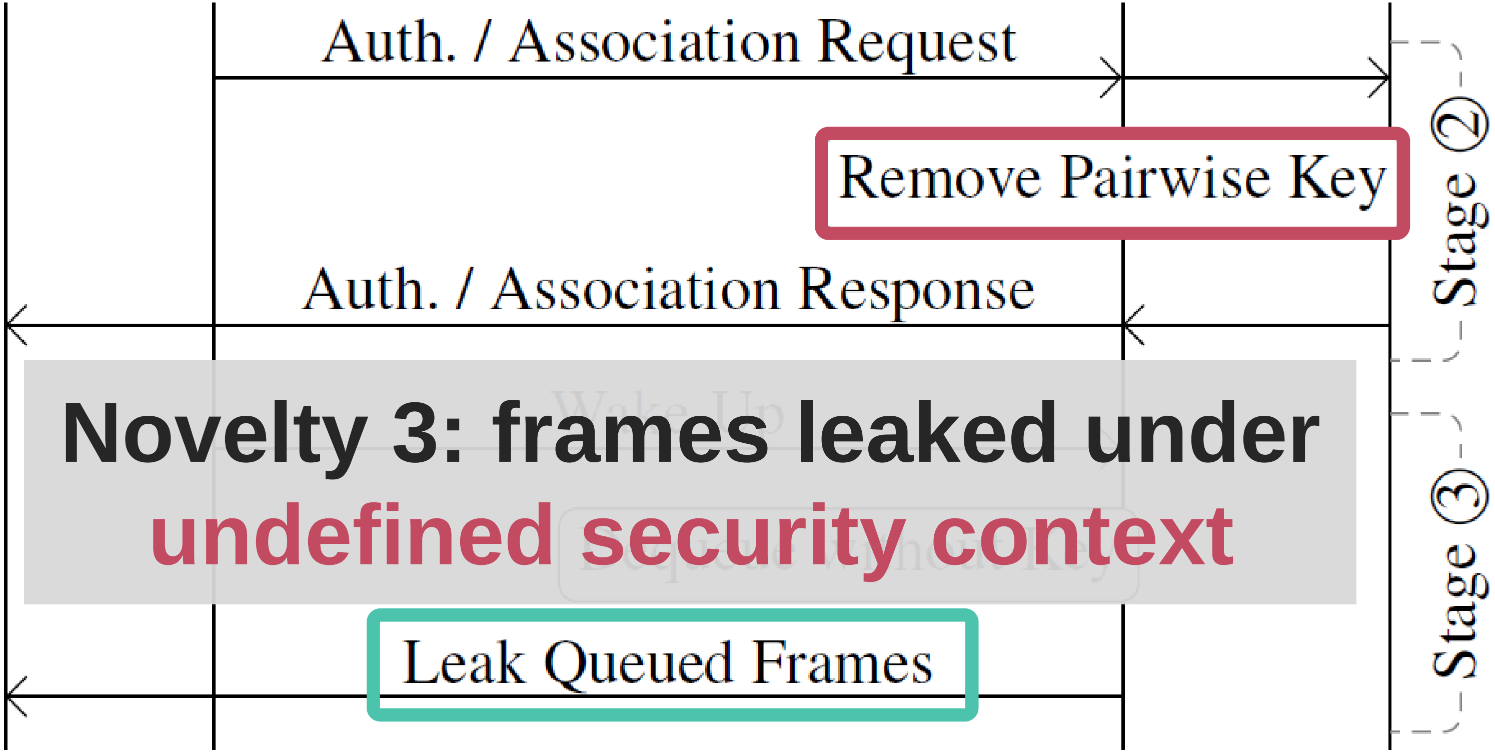
Dequeue without Key

Leak Queued Frames

Stage ②

Stage ③

Attack 1: leaking frames



Novelty 3: frames leaked under undefined security context

Undefined security context: FreeBSD example

How the frame is leaked depends on kernel version & driver:

Version	driver (vendor)	Leakage
13.0	run (Ralink)	Plaintext
13.1	run (Ralink)	WEP with all-zero key
13.1	rum (Ralink)	CCMP with group key
13.1	rtwn (Realtek)	CCMP with group key

- Malicious insiders know the group key!
- Linux, NetBSD, open Atheros firmware also affected

Root cause

Standard isn’t explicit on how to manage buffered frames

- Should drop buffered frames when refreshing/deleting keys

Frames are buffered in plaintext

- Alternative: encrypt frames before buffering them (like TLS)

New attack 2:

Network Disruptions

Background: DoS attacks

Well-known DoS attacks:

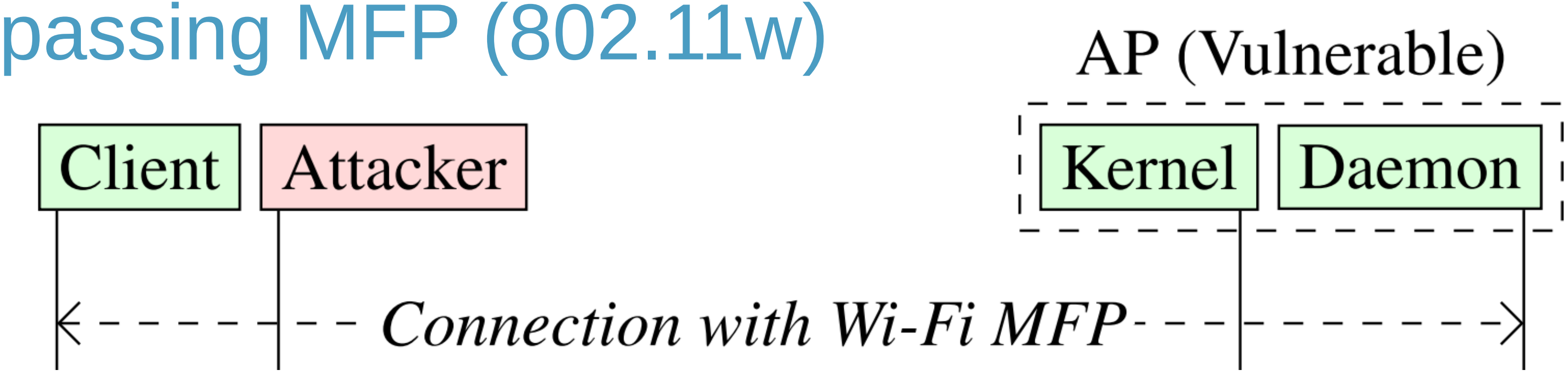
- Deauthentication: spoof “disconnect” frames
- Association: spoof “I want to connect” frames

Both remove connection state of the victim

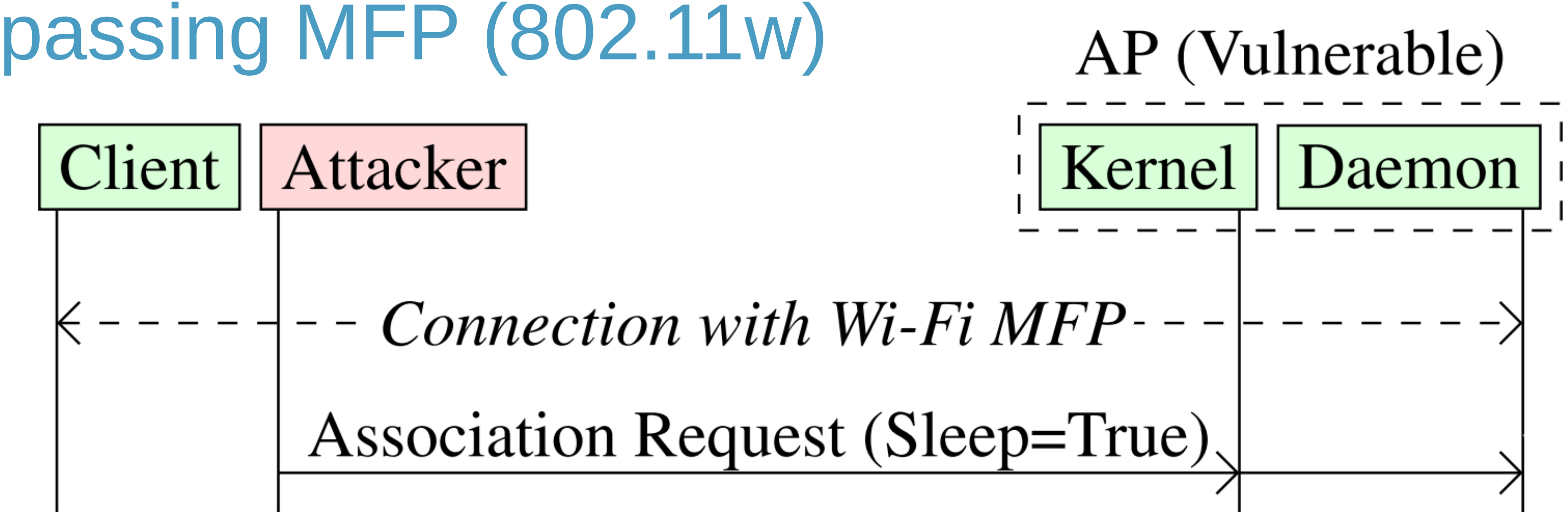
Defense:

- Management Frame Protection (**MFP** = 802.11w)
- This defense is required in WPA3

Bypassing MFP (802.11w)



Bypassing MFP (802.11w)


AP (Vulnerable)

Client

Attacker

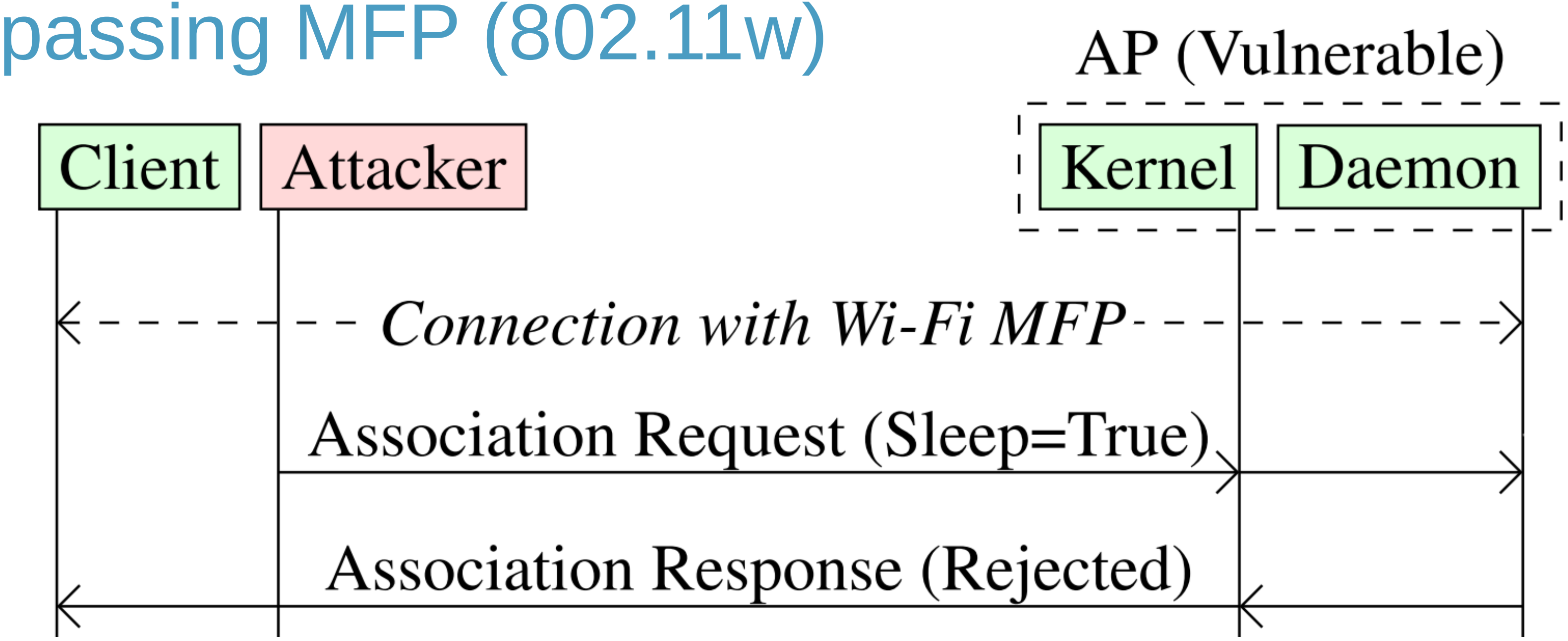
Kernel

Daemon

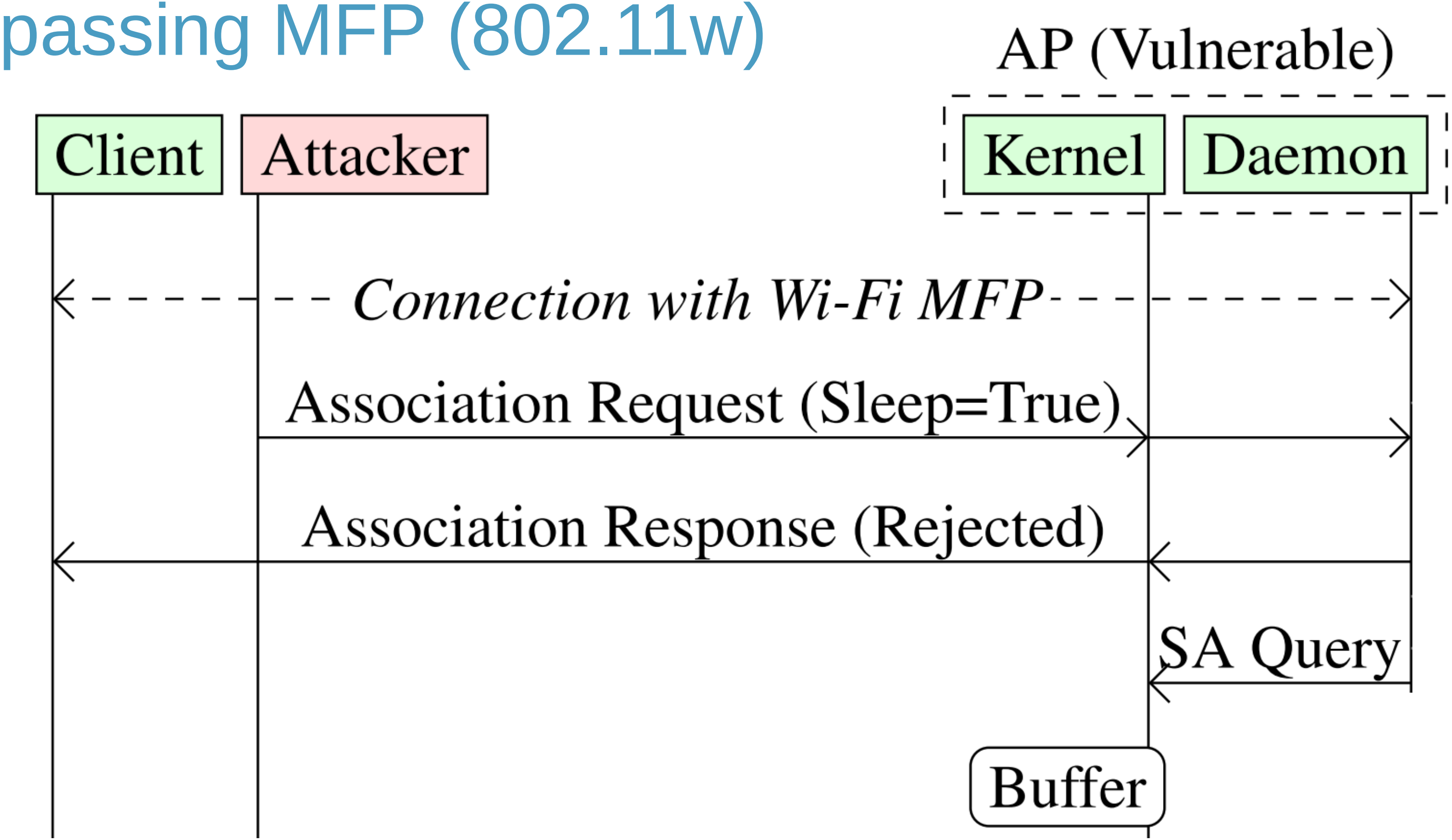
Connection with Wi-Fi MFP

Association Request (Sleep=True)

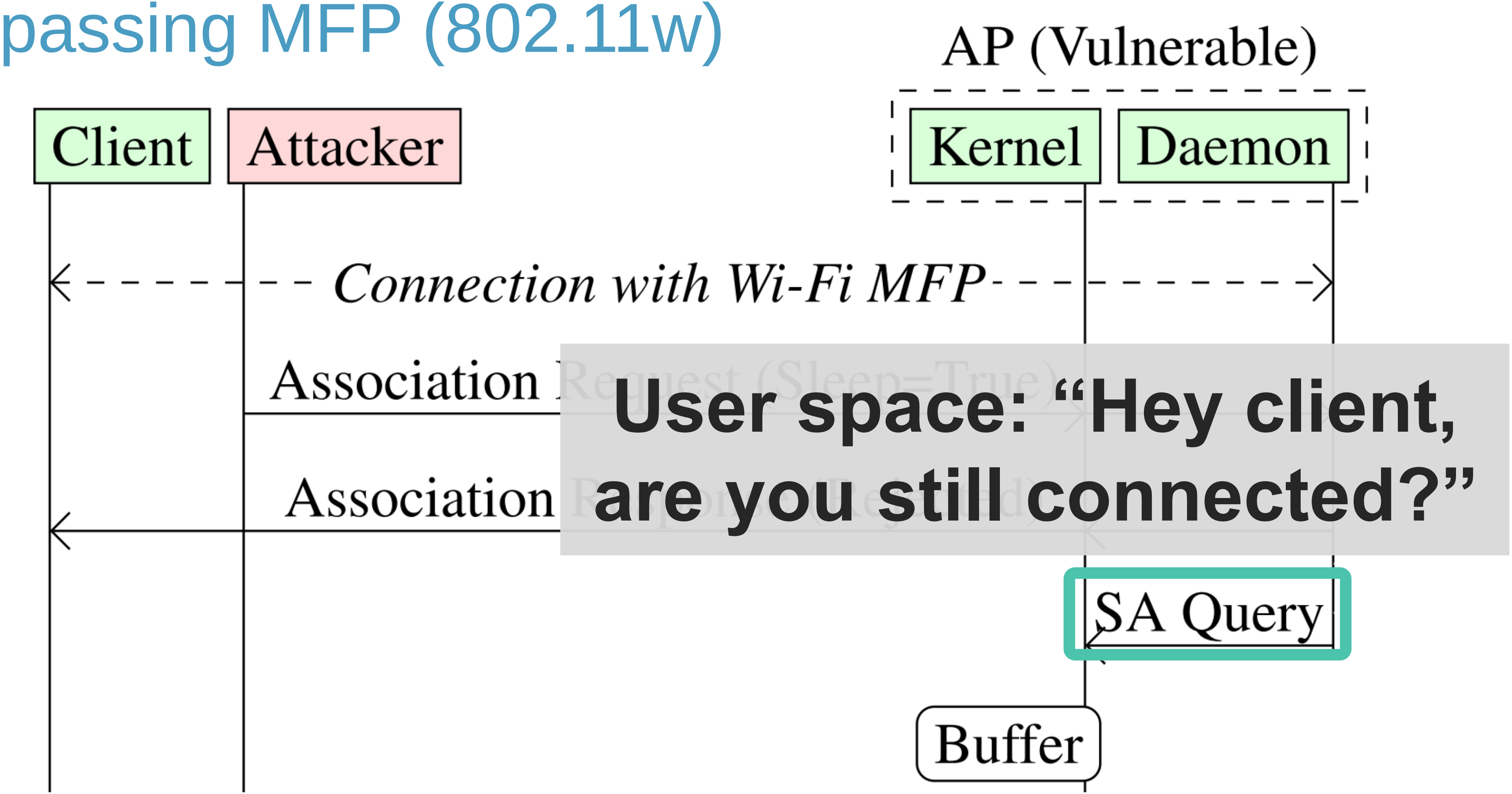
Bypassing MFP (802.11w)


AP (Vulnerable)
Client
Attacker
Kernel
Daemon
Connection with Wi-Fi MFP
Association Request (Sleep=True)
Association Response (Rejected)

Bypassing MFP (802.11w)

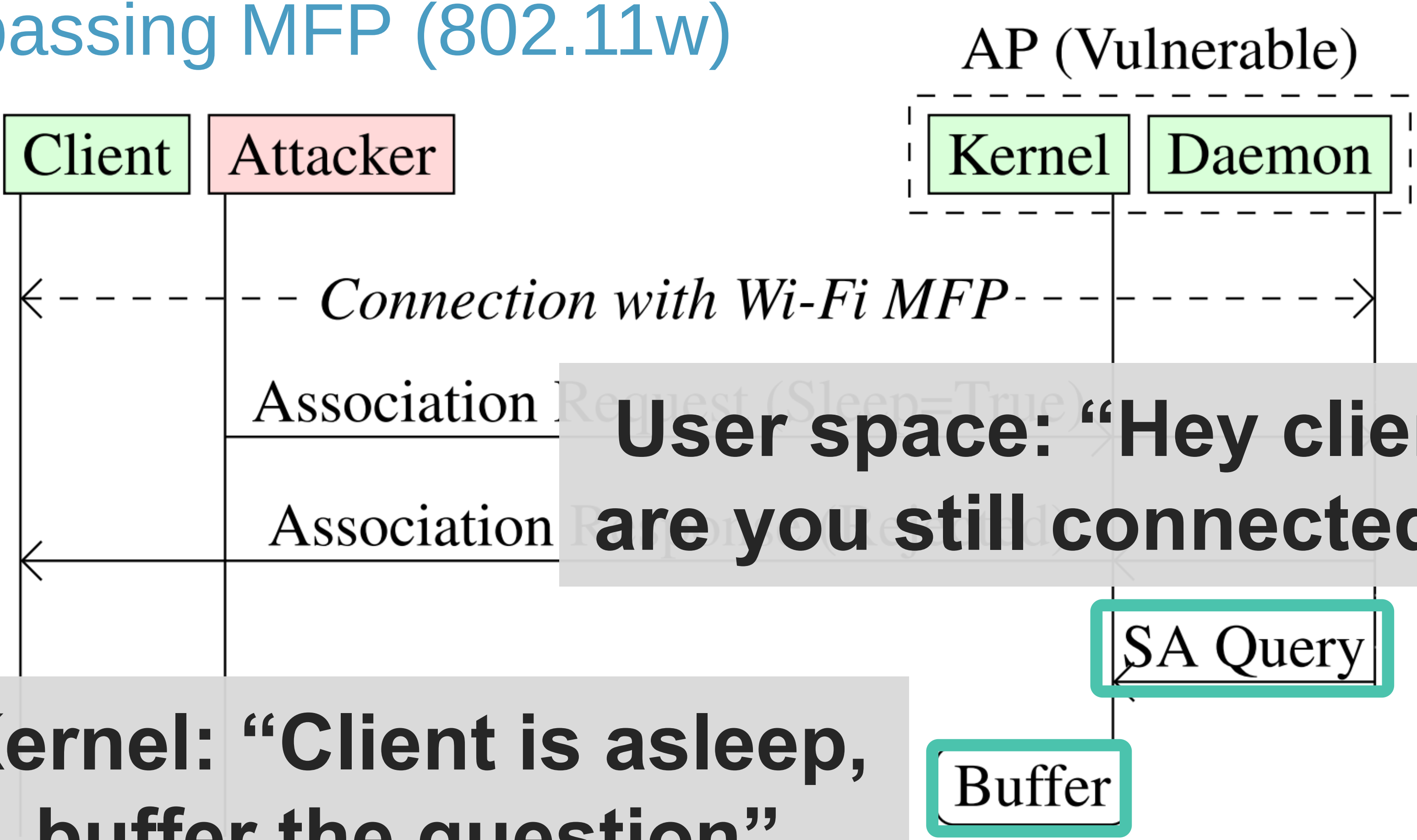

AP (Vulnerable)
Client
Attacker
Kernel
Daemon
Connection with Wi-Fi MFP
Association Request (Sleep=True)
Association Response (Rejected)
SA Query
Buffer

Bypassing MFP (802.11w)


AP (Vulnerable)
Client
Attacker
Kernel
Daemon
Connection with Wi-Fi MFP
Association Request (Sleep=True)
Association Response (Rejected)
SA Query
Buffer

User space: “Hey client, are you still connected?”

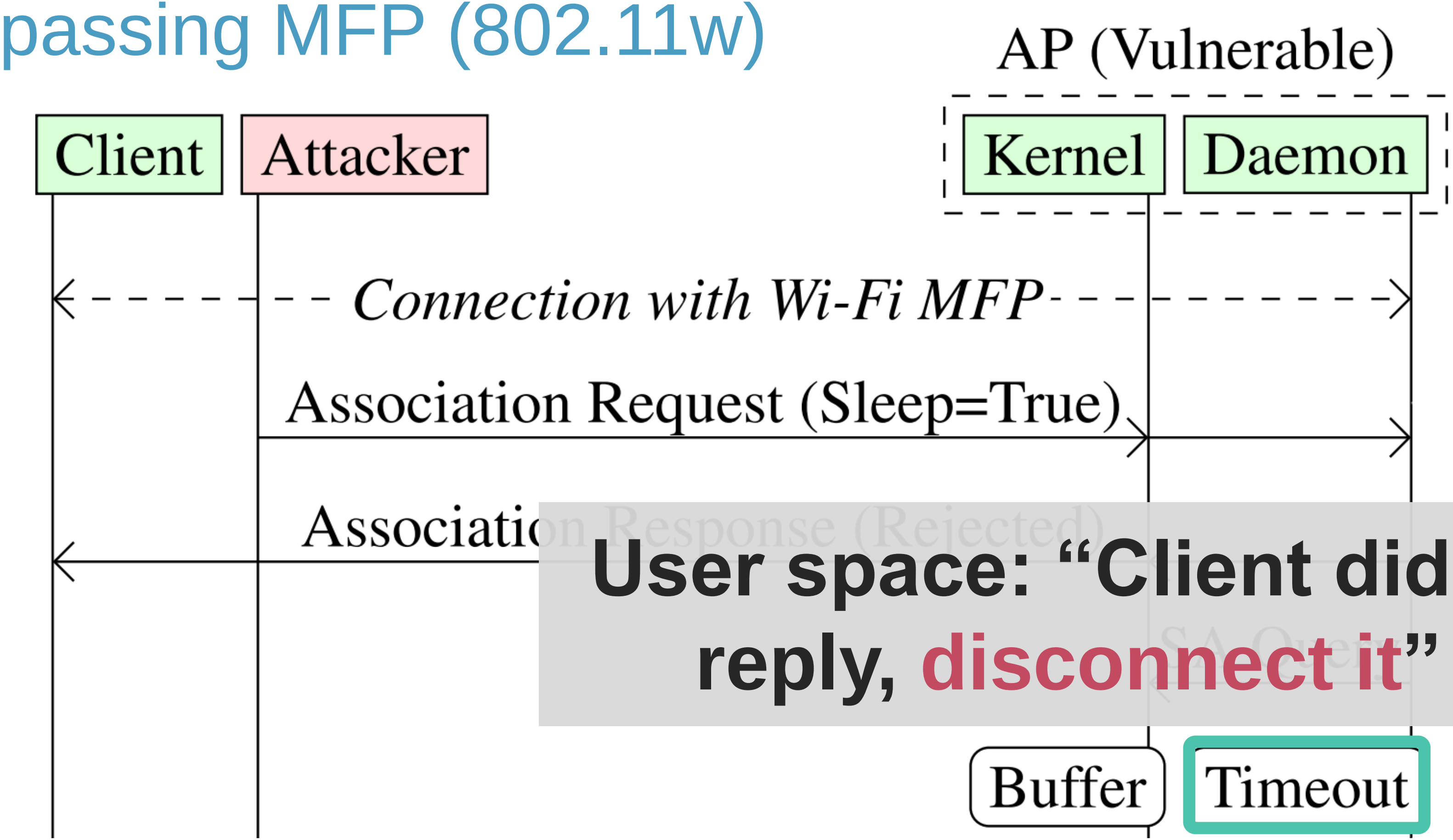
Bypassing MFP (802.11w)


AP (Vulnerable)
Client
Attacker
Kernel
Daemon
Connection with Wi-Fi MFP
Association Request (Sleep=True)
Association Response (Rejected)
SA Query
Buffer

User space: “Hey client, are you still connected?”

Kernel: “Client is asleep, buffer the question”

Bypassing MFP (802.11w)


AP (Vulnerable)
Client
Attacker
Kernel
Daemon
Connection with Wi-Fi MFP
Association Request (Sleep=True)
Association Response (Rejected)
SA Query
Buffer
Timeout

User space: “Client didn’t reply, disconnect it”

Other Attacks & Defenses

Can also **force buffering of Fine Timing Measurements** frames

- Used to measure distance to AP and localize device
- For details, see our paper “Framing Frames: Bypassing Wi-Fi Encryption by Manipulating Transmit Queues” (USENIX Security)

Defenses:

- Never buffer “are you still connected?” frames
- Authenticate the sleep bit in the header of Wi-Fi frames
- **Standard should be updated** with one of these defenses

New attack 3:

Bypassing client isolation

What is client isolation?

Blocks traffic between clients:

- Clients **cannot attack each other**
- ARP spoofing is not possible

All clients have unique encryption keys:

- Prevents “Hole 196” attack (Black Hat ’10)

→ **Defends against malicious insiders**

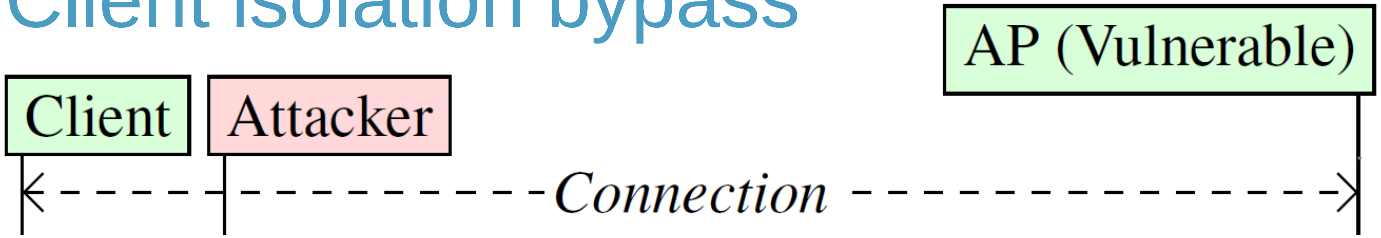
Attack 2: bypassing Wi-Fi client isolation

Target is networks that use **client isolation**. Examples:

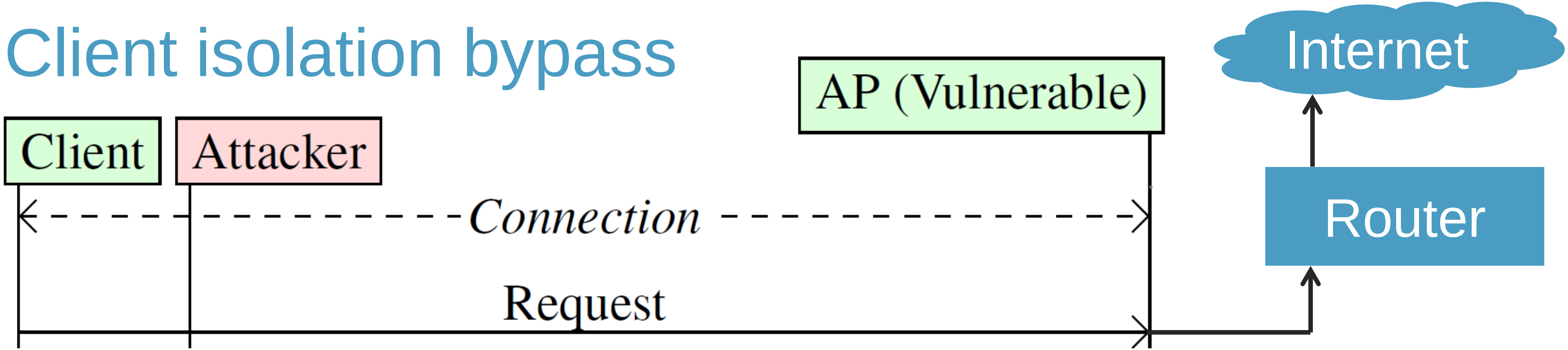
- Company network with malicious/compromised clients
- Public hotspots that require authentication

→ Adversary can connect to the network, but can’t attack others

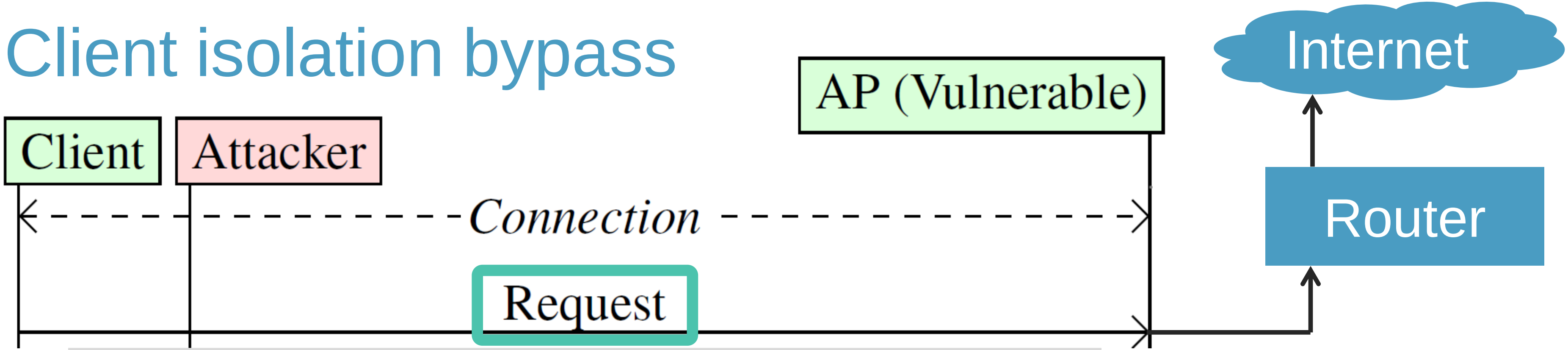
Client isolation bypass



Client isolation bypass

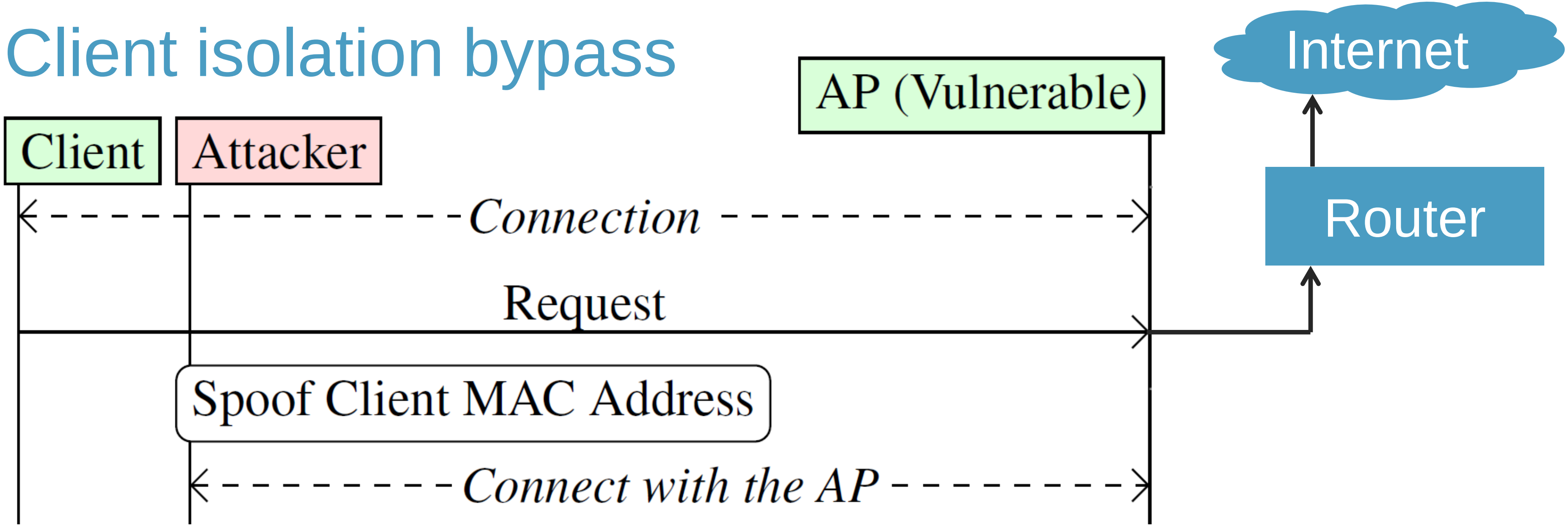


Client isolation bypass

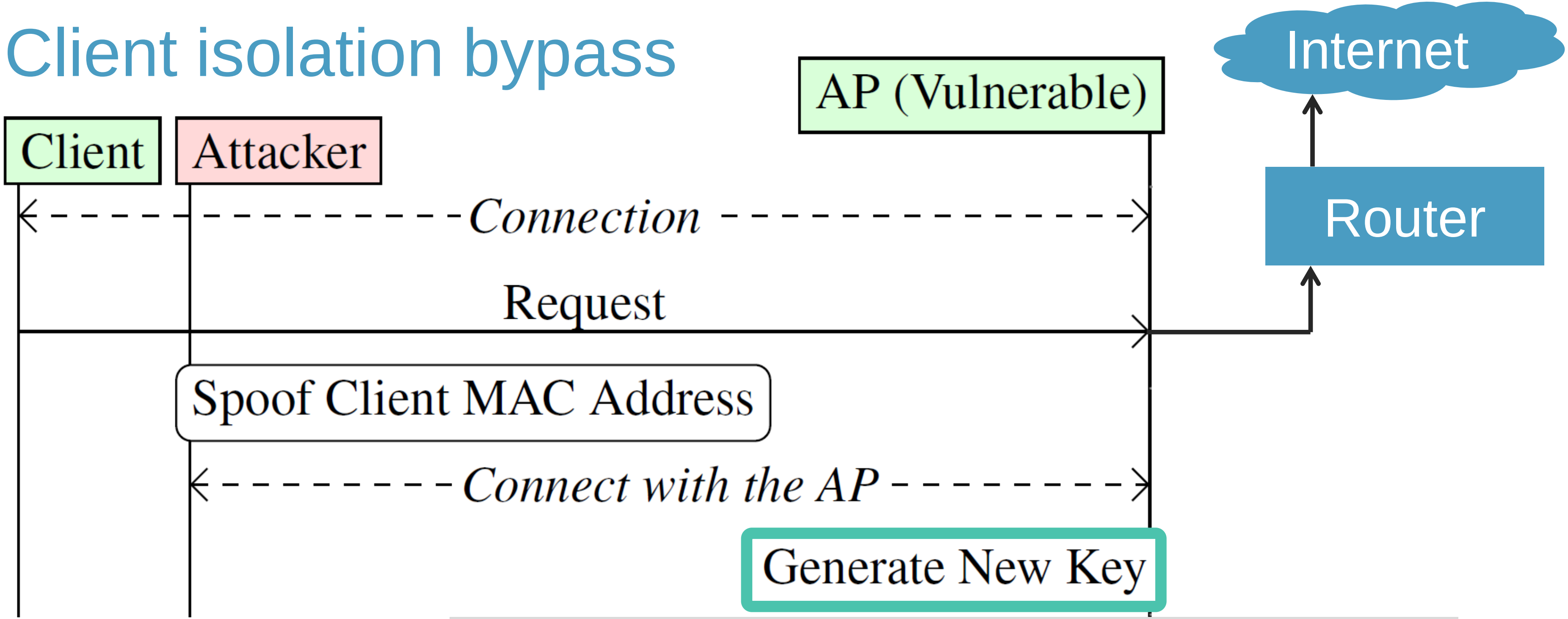


E.g., DNS or HTTP request

Client isolation bypass

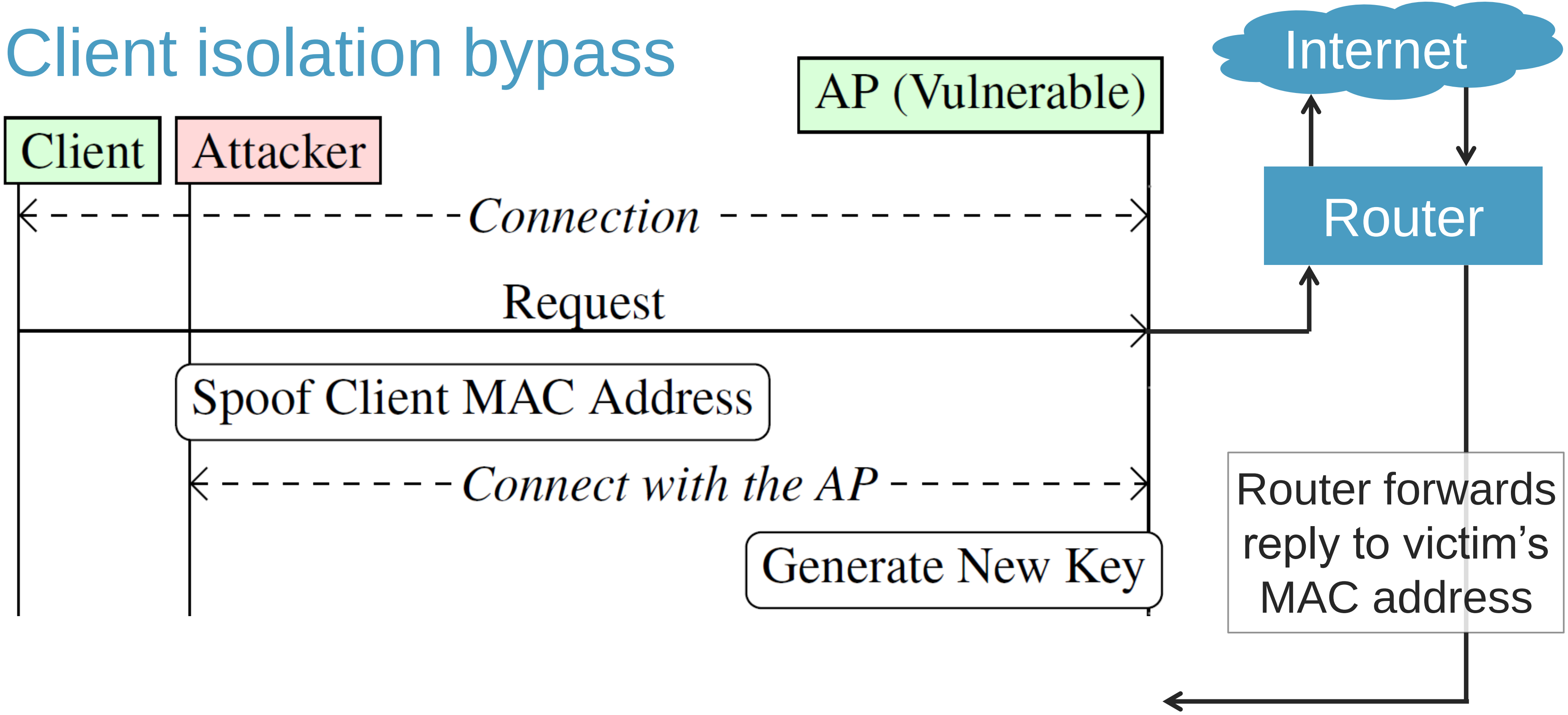


Client isolation bypass



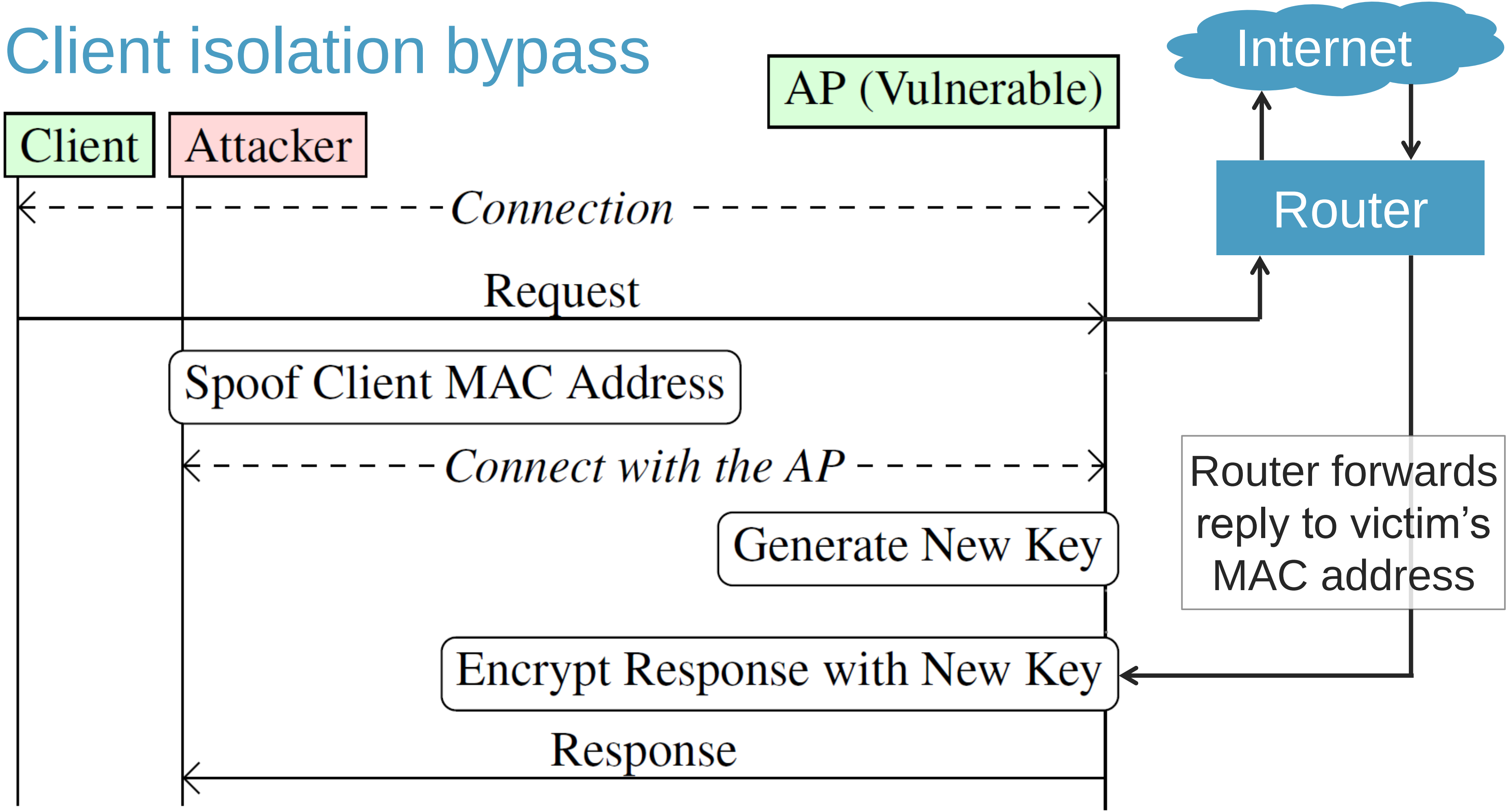
New key is associated with the victim's MAC address

Client isolation bypass



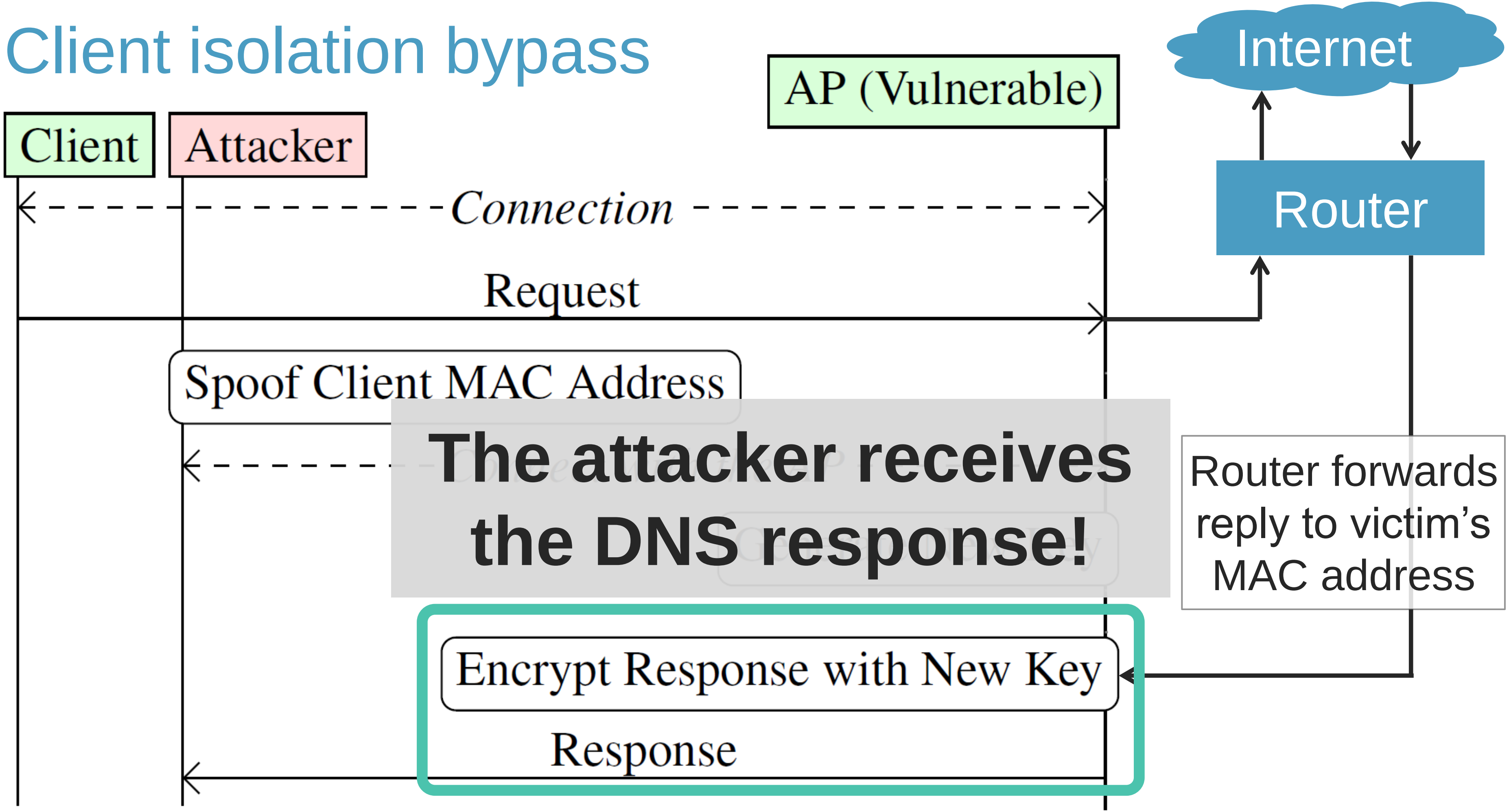
Router forwards reply to victim’s MAC address

Client isolation bypass



Router forwards reply to victim's MAC address

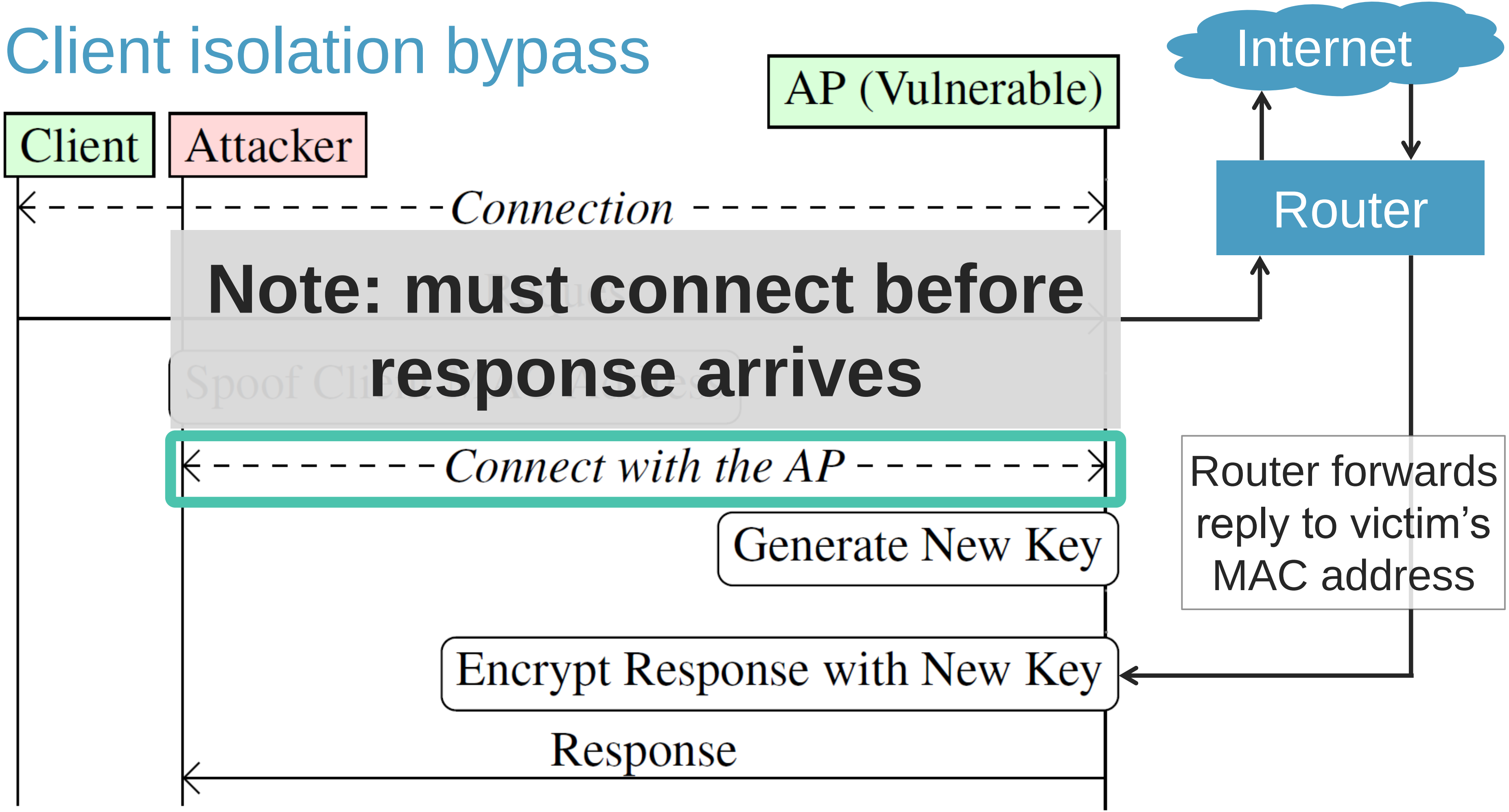
Client isolation bypass



The attacker receives the DNS response!

Router forwards reply to victim’s MAC address

Client isolation bypass



Note: must connect before response arrives

Router forwards reply to victim’s MAC address

Fixing client isolation

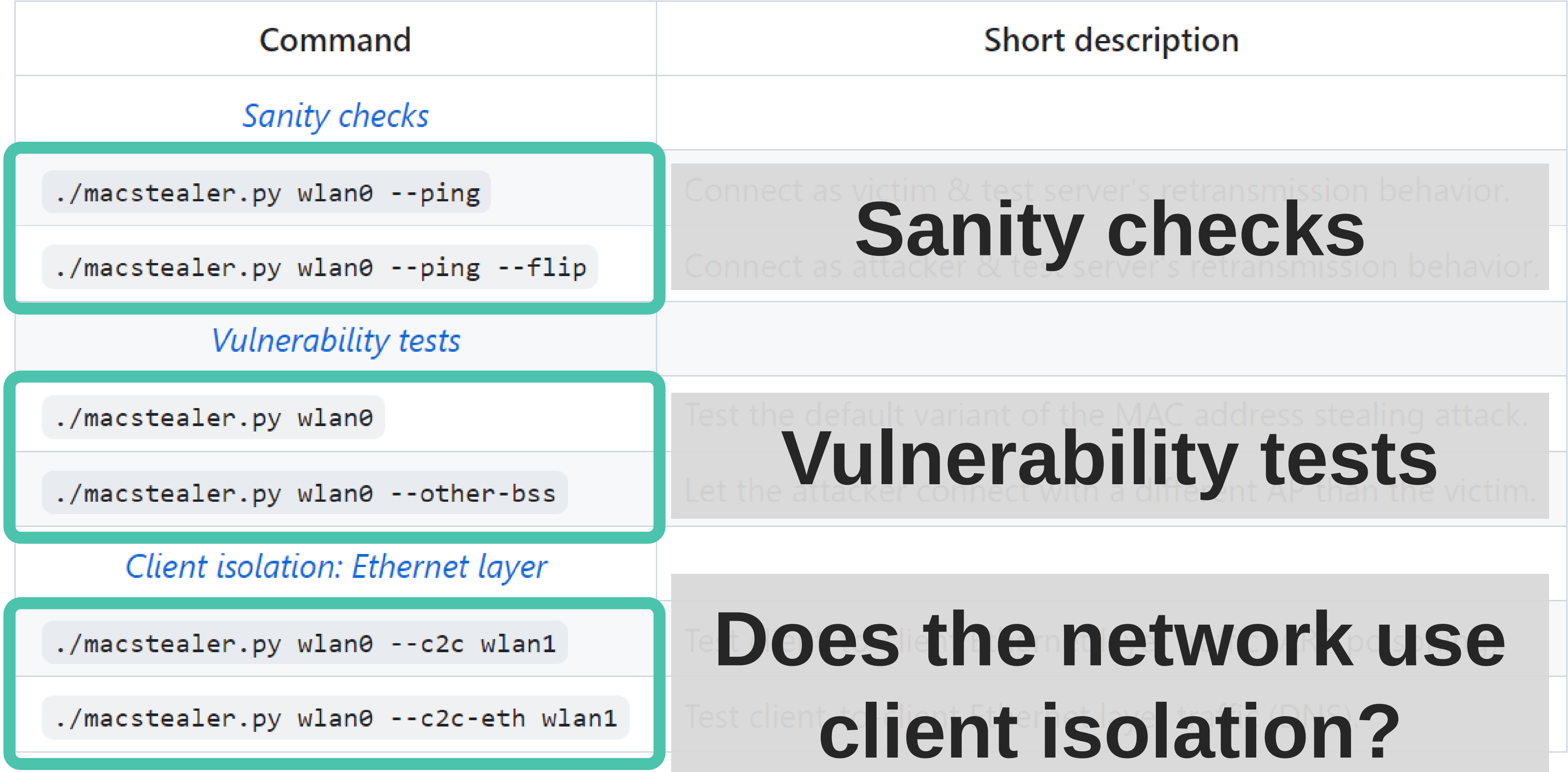
Disallow recently-used MAC address unless:

- Certain amount of time has passed (incomplete defense)
- We're sure it's the same user as before (complete defense)
  - Based on 802.1X identity or cached keys (not always available)

Currently few vendors implemented a defense or mitigation

- Client isolation is flawed but still useful
- Alternative: use VLANs to isolate groups

Tool to test devices: MacStealer



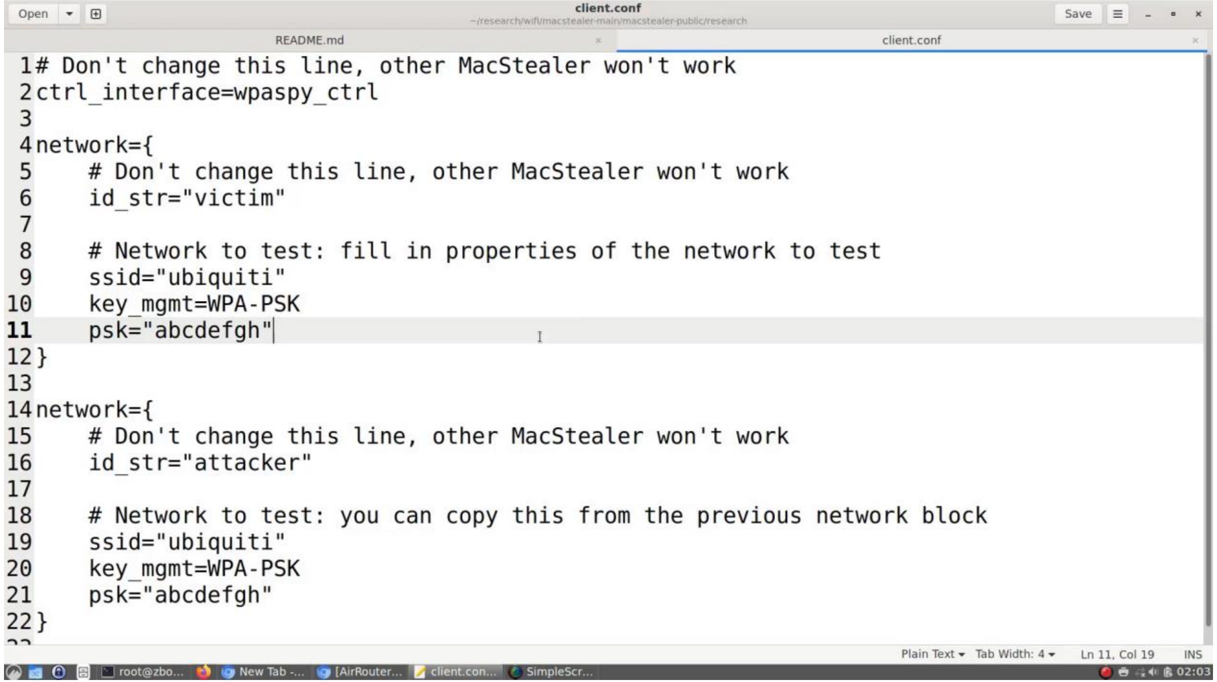
Command	Short description
Sanity checks	
`./macstealer.py wlan0 --ping`	Connect as victim & test server's retransmission behavior.
`./macstealer.py wlan0 --ping --flip`	Connect as attacker & test server's retransmission behavior.
Vulnerability tests	
`./macstealer.py wlan0`	Test the default variant of the MAC address stealing attack.
`./macstealer.py wlan0 --other-bss`	Let the attacker connect with a different AP than the victim.
Client isolation: Ethernet layer	
`./macstealer.py wlan0 --c2c wlan1`	Test client-to-client Ethernet layer traffic (ARP poisoning).
`./macstealer.py wlan0 --c2c-eth wlan1`	Test client-to-client Ethernet layer traffic (DNS).

Sanity checks

Vulnerability tests

Does the network use client isolation?

MacStealer demo



```
# Don't change this line, other MacStealer won't work
ctrl_interface=wpaspy_ctrl

network={
    # Don't change this line, other MacStealer won't work
    id_str="victim"

    # Network to test: fill in properties of the network to test
    ssid="ubiquiti"
    key_mgmt=WPA-PSK
    psk="abcdefgh"
}

network={
    # Don't change this line, other MacStealer won't work
    id_str="attacker"

    # Network to test: you can copy this from the previous network block
    ssid="ubiquiti"
    key_mgmt=WPA-PSK
    psk="abcdefgh"
}
```

→ Ubuiqiti is one of the few vendors that implemented a mitigation!

Experiments

All tested professional & home APs were vulnerable

→ **Design flaw** in Wi-Fi client isolation!

→ Useful test for auditors

github.com/vanhoefm/macstealer

Conclusion

Standard is vague on how to manage buffered frames

- Can **leak frames** under different security context
- Important to **model/define transmit queues**

Can partially **bypass client isolation**

- All devices vulnerable → **design flaw**
- Hard to fully prevent

Backup slide: root cause

Client identity not authenticated across the network stack:

- Wi-Fi security: 802.1X identity (username)
- Packet routing: IP/MAC addresses

Not bound to each other

→ Wi-Fi attacker can spoof client’s identity on other layers

Other observation: client isolation was “bolted on” by vendors

- Not part of IEEE 802.11 standard → less studied

Backup slide: fast security context override

Technique to quickly reconnect. Experiments:

- Minimum reconnect time: ~12 ms
- Average UDP response time: [Verizon]
  - Transatlantic connections: ~70 ms
  - Connections within Europe: ~13 ms
- TCP responses are retransmitted → trivial to intercept

[Verizon] Verizon IP latency statistics