Analysing Leakage during VPN Establishment in Public Wi-Fi Networks

Christian Burkert, Johanna Ansohn McDougall, Hannes Federrath and Mathias Fischer
Universität Hamburg
{burkert,ansohn.mcdougall,federrath,mfischer}@informatik.uni-hamburg.de

***Abstract*—The use of public Wi-Fi networks can reveal sensitive data to both operators and bystanders. A VPN can prevent this. However, a machine that initiates a connection to a VPN server might already leak sensitive data before the VPN tunnel is fully established. Furthermore, it might not be immediately possible to establish a VPN connection if the network requires authentication via a captive portal, thus increasing the leakage potential. In this paper we examine both issues. For that, we analyse the behaviour of native and third-party VPN clients on various platforms, and introduce a new method called *selective VPN bypassing* to avoid captive portal deadlocks.**

***Index Terms*—wi-fi, hotspot, vpn, privacy, captive portal**

I. Introduction

Public Wi-Fis supply Internet connectivity on the go. However, their usage comes with considerable privacy risks: A Wi-Fi operator can monitor all traffic, analyse the metadata and, in case of unencrypted connections, even expose its users to nearby sniffers and attackers [1], [2]. VPNs are used to mitigate these dangers by applying an additional layer of encryption. However, they can also give a false sense of security: Leakage of traffic can already occur while a user attempts to connect to a VPN, and a captive portal might even force the user to temporarily disable the VPN altogether, because—as we will show in this paper—many VPN clients interfere with captive portal detection. After joining the network, running applications like mail or chat clients will themselves attempt to connect to their servers. During the time the VPN is not yet established, this might leak potentially sensitive information about a user's habits, preferences, or work environment to the network.

VPNs were originally designed to establish connectivity to remote private networks and to access their remote services. Nowadays, they are mostly used for privacy-friendly surfing: They aim at masking the original source IP addresses with that of the VPN endpoint and thereby protecting their users, e. g., from observation by Internet Service Providers (ISPs). For that use case, it is crucial that *all* traffic is routed via the VPN tunnel and nothing is leaked to the intermediate network besides the VPN connection itself.

With this paper, we are the first to examine the issue of secure VPN establishment in captive networks and present evidence that native VPN clients shipped with Windows, macOS, iOS, Android, and Ubuntu/GNU Linux, as well as popular third-party clients fall short in protecting user privacy during the establishment of VPN connections. To summarise, we make the following contributions:

- We systematise the current state of system APIs for VPNs.
- We analyse OS mechanisms for captive portal detection.
- We examine the behaviour and leakage of native and third-party VPN clients, including in captive networks.
- We introduce *Selective VPN Bypassing*, a concept of gradual and selective network capability management to avoid leakage during captive network remediation and VPN connection establishment.

The remainder of this paper is structured as follows: In Sect. II, we provide background and terminology. Sect. III presents related work. In Sect. IV, we define the requirements for a secure VPN establishment. Sect. V describes the status quo on VPN APIs. In Sect. VI, we analyse VPN clients and APIs for violations of our security requirements. Sect. VII proposes a design for a leak-free VPN establishment, before Sect. VIII concludes the paper.

II. Background and Terminology

In this section, we briefly describe key concepts of Wi-Fi communication, captive portals and VPNs, and introduce additional terminology used throughout the paper.

A *Public Wi-Fi* or *Hotspot* is a 802.11 Wi-Fi network that is open to the public, i. e., accepts connections from any client. Unless explicitly stated, we assume public Wi-Fis to operate without encryption. To mitigate the potential dangers of surfing in an unencrypted public Wi-Fi, users can decide to increase their security by utilising a VPN. With respect to VPNs, we introduce the term *VPN Bootstrapping*: It describes the process of blocking all traffic except that required to establish the VPN connection until the VPN tunnel is successfully established.

While public Wi-Fis can often be used without special access rights, providers can present their customers with a *Captive Portal* (CP): A website that users are automatically redirected to that contains terms of service and sometimes the necessity to input credentials. Until the terms of the captive portal are fulfilled, access to the Internet is blocked. The process of signing-in and lifting the network block is denoted as *remediation*. We use the term *Captive Network* (CN) to refer to hotspots containing a captive portal. CPs can be explicitly announced via a DHCP option or a Router Advertisement (RA) extension, which informs the client of the

Accepted at IEEE ICC 2021.

URI needed to access the authentication page. While these announcement options exist and have been standardised [3], they are not widely adopted in practice. Instead, platforms apply heuristics to detect captive networks: Upon successfully connecting to a network, clients send out HTTP requests to a predefined URL, expecting a predefined response, e. g., an HTTP status code 204. A CN instead replies with an HTTP redirect (e. g., status code 307), redirecting the user to the CP [4]. Thereby, the OS assumes a CN and displays the CP.

When attempting to use a VPN in a CN, a *Captive Deadlock* can occur: in it, the leak prevention of a VPN client blocks the communication with the CP that is necessary to gain an Internet uplink, and thereby indirectly also blocks the route to the VPN endpoint.

III. Related Work

The security and privacy of public Wi-Fis and VPN client software has been extensively studied in the literature. [1], [2], [5] examine risks caused by public Wi-Fi and captive portals, and the reason why people use them nonetheless. [6] and [7] analyse the VPN clients on mobile platforms. [6], [8] and [9] verify the security and privacy claims of commercial VPN clients. Among other things, they discover severe leakage of IPv6 and DNS traffic: Up to 84% of VPN apps don't tunnel IPv6 [6], and around 60% of VPN apps use Google's DNS servers, while only about 10% use own DNS resolvers [6].

However, regarding traffic leakage during VPN connection establishment, there is very little prior work and—to the best of our knowledge—we are the first to analyse VPN clients and their behaviour in captive networks. Karlsson et al. [10] present a prototypical device that connects to public Wi-Fis, opens up a VPN tunnel and then creates an encrypted Wi-Fi for the user to connect to, such that all traffic is routed through the VPN tunnel on the intermediate device. This mitigates the startup leakage issue by moving it from the user device to the intermediate device which presumably exposes less sensitive traffic of its own. However, we argue that the requirement to maintain an additional device is impractical to most users.

IV. Requirements for Secure VPN Bootstrapping

To ensure a secure and privacy-preserving establishment of VPN connections in public Wi-Fis, we propose the following requirements for secure VPN bootstrapping:

R1: Always-on Functionality: A client offers an always-on or similar functionality that asserts that a VPN tunnel is established when network connectivity is available. If not automated, the user must be able to activate this functionality without an existing Internet connection, e. g., before joining a public Wi-Fi.

R2: Captive Network Support: The VPN client allows the OS to perform captive portal detection and remediation or performs it itself. The client does not cause a captive deadlock.

R3: Minimal startup traffic: No traffic is sent from the client device that is not necessary to remediate a captive network or to establish a VPN tunnel.

R4: Blocking Fail State: Outbound traffic continues to be blocked if a VPN tunnel cannot be successfully established (e. g., if the VPN endpoint is unreachable).

R5: No Tunnel Bypass: After successful VPN tunnel establishment, no non-VPN traffic, such as previously started TCP streams, bypass the tunnel. Instead, any preexisting connection is interrupted and reestablished through the tunnel. Periodic requests to check the state of the captive network are exempted.

V. VPN API Status Quo

In this section, we describe the current state of system APIs available for VPNs on major platforms according to developer documentation.

a) Apple macOS and iOS: As part of their network extension framework, Apple offers an API for creating VPN apps that build on Apple's system VPN functionality (Personal VPN [11]) or provide custom protocol implementations (Packet Tunnel Provider [12]). This API offers always-on functionality (R1) via so called on-demand rules, which can be configured to trigger, e. g., when a Wi-Fi connection is established [13]. According to the documentation, such on-demand connection rules block outgoing traffic until the VPN tunnel is established (R3).

b) Android: Developers can build VPN apps using the system API and the `BIND_VPN_SERVICE` permission. VPN apps can run in, among others, always-on (R1) and per-app mode. Always-on VPN connections are kept alive unconditionally by the system as long as the device is running and Internet connectivity is available. Developers of VPN apps can specify lists of allowed and disallowed apps whose traffic is to be tunnelled through the VPN. It is also possible to block all connections outside the VPN tunnel, which results in disallowed apps losing all network connection [14].

c) Windows 10: Always-on functionality (R1) is built in as an auto-trigger for VPN profiles.[1] In general, VPN profiles can be provided by a VPN app[2] or via a mobile device management mechanism to remote-join clients to a domain[3].

d) Ubuntu GNU/Linux: Since the landscape of GNU/Linux distributions is very diverse, we focused our analysis on the popular desktop distribution Ubuntu. Ubuntu uses NetworkManager (NM) as its high-level daemon for networking including VPN. NM provides APIs to plug in VPN services via DBus[4] and libnm[5]. VPN services can declare themselves persistent, i. e., they will attempt to maintain the connection across link changes and outages.[6] VPNs can be further set as so-called *secondaries* for other

[1] https://docs.microsoft.com/en-us/windows/security/identity-protection/vpn/vpn-auto-trigger-profile

[2] https://docs.microsoft.com/en-us/uwp/api/windows.networking.vpn.ivpnprofile?view=winrt-19041

[3] https://docs.microsoft.com/en-us/windows-server/remote/remote-access/vpn/always-on-vpn/deploy/always-on-vpn-deploy

[4] https://developer.gnome.org/NetworkManager/stable/gdbus-org.freedesktop.NetworkManager.VPN.Plugin.html

[5] https://developer.gnome.org/libnm/stable/NMVpnServicePlugin.html

[6] https://developer.gnome.org/libnm/stable/NMSettingVpn.html

connections to be activated in reaction to the other connection going online (R1).

VI. Experimental Analysis

While the existing VPN APIs partially contain mechanisms to fulfil the requirements R1 and R3, apps using the API don't necessarily utilise the functionality or fail to fulfil the other requirements. In this section, we present an experimental analysis concerning OS-specific Captive Portal Detection (CPD) mechanisms, and then test different VPN clients with respect to their fulfilment of the requirements.

A. Testbed Setup and General Procedure

1) Setup: Our testbed comprises a Raspberry Pi that runs the Wi-Fi Access Point (AP) and also captures the incoming Wi-Fi traffic of our test clients. We use a Raspberry Pi 3 Model B+ running Raspbian GNU/Linux 10 and with hostapd 2.2.7 providing the WPA2-secured AP and Nodogsplash 4.5.1 beta providing the captive portal functionality. The captive network is set up to redirect (status code 307) plain HTTP requests to the captive portal sign-in page running on the same Raspberry Pi, where the user can gain Internet access by clicking a continue button. The traffic capturing is done with Wireshark/tshark.

Regarding the test clients, we used the following setup: A Google Nexus 5X running Android 10.0[7], an iPad running iOS 13.7, a MacBook Pro running macOS 10.15.7, Ubuntu 20.04 LTS with NetworkManager 1.22.10, and Windows Education 10.0.19041 both running on a Dell laptop.

2) Leak Classification: We consider all outgoing traffic that is not required for VPN setup as leakage. However, the low-level protocols ARP, EAPOL, DHCP, ICMP, IGMP, LLMNR, and mDNS are not considered leakage. We further distinguish leakage to hosts operated by the platform maintainer (e. g., Apple for iOS and macOS, or Microsoft for Windows) from leakage that is going to other third-party hosts.

3) General Test Procedure: We test each VPN client in a captive network (denoted as *captive mode*) and in a unrestricted network (*open mode*) as well as in a network that selectively but permanently blocks, i. e. drops, the traffic to the respective VPN endpoint (*block mode*). Note that in block mode, both the safe fail state requirement (R4) is tested, as well as the probability to detect potential race conditions during the VPN startup is increased. If a client does not offer auto-connect functionality, we instead manually activate the connection before joining the testbed Wi-Fi.

Each network configuration is tested and captured at least three times while performing the following steps: 1) disconnect client device from Wi-Fi, 2) activate VPN client if necessary, 3) clear CP state and start capture, 4) connect client to Wi-Fi, 5) complete CP sign-in if prompted, and 6) continue capture for 20 seconds. Afterwards, we inspect the captured traffic and classify it according to our leakage definition.

[7]Using PixelExperience ROM version 10.0-20200912-1735

TABLE I
Overview of platform behaviour during captive network remediation.

	macOS	iOS	Windows	Android	Ubuntu
System employs CPD	✓	✓	✓	✓	✓
Blocking of platform traffic	✓	✗	✗	✗	✗
Blocking of third-party traffic	✓	✓	✗	✓	✗

B. Captive Portal Detection Mechanisms

To establish a baseline, we analysed each platform's behaviour when confronted with a captive network. These tests were conducted without any VPN enabled. We analysed two scenarios: a) CP sign-in is completed, i. e., the continue button on the CP website is clicked, and b) CP sign-in is omitted, leaving the client captured. We observed that all platforms employ effective Captive Portal Detection (CPD) mechanisms and prompt the user to sign in to the captive network. However, the platforms exhibit the following differing behaviour regarding leakage during CPD, which is also summarised in Table I: On macOS and iOS, we find that CPD takes place before connectivity is available to the rest of the system. If the sign-in is omitted, macOS and iOS will continue to block all other outgoing traffic. However on iOS, DNS queries were leaked about non-CPD-related platform hostnames, followed by IP traffic to those hosts. On Android, DNS lookups as well as TCP traffic to non-CPD Google hosts take place before remediation. On Ubuntu, the CPD appears non-blocking. We observed outbound traffic to third-party global and local destinations. On Windows, we observed non-CPD traffic to Microsoft hosts as well as to third-party hosts.

In the following sections, we present the findings of our analysis of the native VPN clients, the macOS API and third-party VPN clients. The results are summarised in Table II and will be discussed in Sec. VI-F.

C. Native VPN Clients

All tested operating systems ship with built-in clients that offer basic VPN functionality. Note that this analysis only covers the functionality that is exposed by the OS's native GUI (e. g., via a network settings dialogue) and not functionality that is only exposed via APIs or configuration files.

1) macOS and iOS: Although provided by system APIs (cf. Sect. V), always-on functionality is neither part of the native VPN client on macOS (version 10.15.7) nor iOS (version 14.0.1). VPNs set up via the on-board configurators have to be started manually while Internet connectivity is available and do not always (re-)connect if network connectivity is interrupted. Note that iOS additionally supports remotely deploying always-on VPN profiles via device supervision, i. e. mobile device management.[8] As we do not have the necessary prerequisites for device supervision, this option was not included in our analysis.

[8]https://support.apple.com/en-gb/guide/deployment-reference-ios/iore8b083096/1/web/1

TABLE II
OVERVIEW OF THE VPN CLIENT ANALYSIS. SYMBOL USAGE: RACE CONDITION (➠), PLATFORM TRAFFIC LEAK (✳), NOT TESTABLE (?), NOT APPLICABLE (–)

Platform	Client	R1: Always-on	R2: CPD	R3: Minimal	R4: Blocking	R5: No Bypass
macOS	Native	✗	–	–	–	–
	Demo	✓	✓	✗	✗	✗
	EncryptMe	✓	✓	✗	✗	✗
	ExpressVPN	✓	✗	(✓)	✓	?
	Mullvad	✓	✗	✓	✓	✓
	ProtonVPN	✓	✓	✗	✓	✗
iOS	Native	✗	–	–	–	–
	EncryptMe	✓	✓	✳	✓	✗
	ExpressVPN	✓	✓	✳	?	✗
	Mullvad	✓	✓	✓	✓	✓
	ProtonVPN	✓	✓	✳	✳	✗
Windows	Native	✗	(✗)	–	–	–
	EncryptMe	✓	✓	✗	✗	✗
	ExpressVPN	✓	✓	✗	✗	✓
	Mullvad	✓	✗	✓	✓	✓
	ProtonVPN	✓	✗	✓	✓	✓
Android	Native	✓	✗	✓	✓	✓
	EncryptMe	✓	➠	✗	✗	✗
	ExpressVPN	✓	✗	✳	✓	✓
	Mullvad	✓	✗	✳	✓	✓
	ProtonVPN	✓	➠/✗	✗/✓	✗/✓	✳/–
Ubuntu	Native	✓	✗	✗	✗	✓
	ExpressVPN	✓	✗	✗	?	✓
	Mullvad	✓	✗	✓	✓	✓

2) Android: The built-in VPN client [14] offers the option to turn on *always-on* VPN if the VPN server address is provided numerically and a DNS server is set. We found that with always-on enabled, Android entered a captive deadlock. Disabling the captive network allowed Android to establish the VPN tunnel. We found that apart from the CPD traffic, TLS traffic was sent to `www.google.com` before the tunnel was established. Otherwise, no further traffic was leaked.

3) Ubuntu GNU/Linux: Ubuntu ships with NM as a VPN and networking client. Provided that a VPN profile is configured, NM allows to set an always-on VPN profile per connection (e. g., an SSID) in the connection editor. In captive mode and with auto-connect enabled, NM stops forwarding CPD requests, thus causing a captive deadlock. In open mode, the VPN starts as expected. In any case, NM's auto-connect has no blocking effect and outbound traffic is unrestricted, both during establishment and if the VPN is unreachable. In fact, NM appears to react to ICMP notifications stating that the VPN destination is unreachable by setting the connectivity to offline, thus preventing further leaks. However, during the time NM attempts to connect to the VPN, no restrictions are in place for other process to send traffic. In block mode, when no ICMP notification is received because the VPN traffic is dropped, NM retains online connectivity until the VPN times out, thus increasing the time for other processes to leak traffic.

4) Windows: Windows 10's built-in graphical VPN configurator is not able to set up an always-on VPN profile. However, we found that VPN connections are not immediately disconnected if a network interface goes offline. If the network goes online again before the VPN connection times out, we observed that CPD is suppressed and the system is in a deadlock until the VPN timeout occurs, after which outbound traffic is unrestricted. Note, that before the timeout and in open mode, we also observed VPN bypasses by DNS lookups for Microsoft hosts like `www.bing.com`.

D. VPN API Demo

Platforms like macOS and iOS provide dedicated APIs that supposedly offer blocking VPN bootstrapping. However, as they currently don't integrate this functionality in their native VPN clients, we implemented a minimal demo for macOS to test the validity of the documented properties. We selected macOS for our demo because of previous development experience on that platform.

The demo builds on the Personal VPN API (cf. Sect. V) and simply registers an on-demand VPN that auto-connects with an `NEOnDemandRuleConnect` rule every time a Wi-Fi connection is used. While testing our demo, we found that the VPN tunnel is reliably started by the system after we connect to our test Wi-Fi. However, our traffic captures consistently show outgoing platform (towards Apple) and third-party traffic between the CP authentication and the tunnel establishment in both open and captive mode. The documented blocking feature of the `NEOnDemandRuleConnect` rule appears to be insufficient. We further found that after the tunnel establishment, TCP traffic bypasses the tunnel that originates from before the VPN establishment, i. e., for ongoing TCP streams. We also observed ongoing TCP re-transmission attempts that started before the establishment. This corroborates a bypassing vulnerability that has been reported in March 2020 but remains unfixed (in Oct. 2020) [15]. In block mode, we observed continued traffic to platform and third-party hosts after the failed VPN connection attempts. The on-demand connection rule appears not to sufficiently block on a failed establishment.

E. Third-Party VPN Clients

We additionally included the following third-party clients in our analysis: ExpressVPN as the self-declared market leader, EncryptMe as a benchmark used by [10], ProtonVPN as Top 10 provider with open source clients and an active stand in VPN leak prevention [15], and Mullvad VPN as an open source provider.

1) ExpressVPN: On Ubuntu, version 3.0.2.12 deadlocked in captive mode. In open mode, we observed no leaks besides local traffic. Block mode tests were not practical because the endpoint address switched unpredictably. We further found that when the app enters a blocking "unable to connect" state, it could only be re-established with full leakage, despite using the connect statement as described in the info text, which apparently temporarily disables the traffic block.

On iOS, we tested version 8.3.5 which uses the on-demand API to automatically reconnect the tunnel once it is activated. We found non-CPD platform traffic during CPD in captive and open mode. Additionally, we saw TCP resets of previous

platform connections bypassing the tunnel. No deadlocks occurred. Block testing was skipped again.

On macOS, version 7.11.6(6), according to network settings, does not use Apple's on-demand API. In captive mode, we observed leak-free deadlocks as the CPD is interrupted. In open mode, the CPD passes but the tunnel establishment repeatedly fails to complete within the capture time. However, within that time we found no leaks. In block mode, ExpressVPN holds outgoing traffic and prevents leaks.

On Windows, we tested version 9.1.0(258). In captive and open mode, we saw non-CPD platform and third-party traffic before remediation and tunnel establishment. In block mode, no additional leakage occurs. The windows client appears to not sufficiently block startup leaks.

On Android, version 9.0.40 deadlocked in captive mode without leaks except non-CPD platform traffic to `www.google.com`. In open mode, we saw the same leaks but no deadlocks. Block mode caused no additional leaks.

2) EncryptMe: This client has been used as a benchmark by [10]. It offers VPN-activation based on network trustworthiness. On macOS, we tested version 4.2.3 with activated auto-start and leak protection (OverCloak). We found platform and third-party traffic leaks in open, captive and block mode before VPN startup and afterwards. No deadlocks occur.

On iOS, we analysed version 4.4.4 with enabled auto-protect which uses iOS's on-demand functionality. In open mode, the app shows non-CPD platform leaks. During captive mode, we also found continued platform traffic bypassing the VPN. In block mode, we saw no additional leakage.

On Windows, we analysed version 1.1.0. In captive mode, we observed third-party leaks and no deadlocks. In open mode, the VPN establishment completes faster, hence we found fewer and only platform leaks. However, blocking VPN traffic causes third-party traffic to surface again.

On Android, version 4.2.0.1.81964 exhibits indeterministic captive deadlocking and third-party leaks, indicating a race condition between the CPD and the tunnel establishment. After blocking the VPN in captive mode, we observed consistent leak-free deadlocks. In open mode, we found no leaks. However, blocking the VPN in open mode causes third-party traffic to leak. A Linux version is not available.

3) Mullvad VPN: On iOS, version 2020.4 established a tunnel after remediation without leaks. A source code inspection confirms that Mullvad uses the on-demand connection API. Open or block mode cause no leaks either.

On macOS, Windows and Ubuntu, we tested the respective client in version 2020.5 and observed the same behaviour: In captive mode, the clients caused a leak-free deadlock by suppressing the CPD traffic. In open and block mode, the client prevents leakage and we observed only VPN traffic.

On Android, Mullvad is still in beta version (2020.6-beta2). In captive mode, CPD requests are sent and redirected, but the request to the CP is lost, thus causing a deadlock. In open mode, we observed non-CPD platform traffic before tunnel establishment. Block mode triggers no additional leaks.

4) ProtonVPN: On iOS, version 2.2.4 offers always-on functionality that cannot be turned off. We observed non-CPD platform DNS and IP traffic between remediation and tunnel establishment. We also found traffic to the global IP address of our testbed gateway throughout the capture. Blocking the VPN traffic exhibited the same leakage. We additionally observed traffic to Akamai servers, presumably operating for Apple.

On macOS in captive and open mode, ProtonVPN 1.7.2 exhibited no leaks during remediation, but we subsequently observed non-CPD platform and third-party DNS and IP traffic before tunnel establishment. After tunnel establishment, prior platform and third-party TCP streams and local traffic continued. We also noted reverse DNS lookups of the test client's local IP address. In block mode, we saw the same leakage up to the point of the attempted tunnel establishment. After that, no further leaks were visible.

On Windows, version 1.17.3 does not connect automatically, but can be started manually before connecting to the Wi-Fi. In captive mode, no CPD requests are sent. The system is in captive deadlock. In open and block mode, we saw no leaks but traffic to `api.protonvpn.ch`.

On Android, we tested version 2.3.54.0. In captive mode, we observed inconsistent behaviour: the CP request is either suppressed and the system deadlocked, or sent before tunnel establishment alongside other platform traffic. This behaviour indicates a race condition in the CPD and VPN handling. In block mode, increased platform and third-party leaks appear. In open mode, we observed platform traffic leaks throughout. In its settings, ProtonVPN for Android contains instructions to activate always-on functionality by manually enabling Android's always-on and blocking VPN properties in the system VPN profile. With always-on enabled, captive and block mode lead to a leak-free deadlock. Because the app explicitly guides the user to that system feature, we also included these results.

Note that ProtonVPN also offers a command-line client for Linux, however we were unable to test it, because our credentials were rejected.

F. Summary and Discussion

Of the native and third-party VPN clients in our analysis (see Table II), only Mullvad for iOS managed to establish a connection in a captive network without deadlocks or traffic leaks. On all platforms but iOS and macOS, an effective leak protection coincided with deadlocks after a failed CPD. We identified issues with violations of the minimal startup requirement (R3) due to platform traffic leaks on Android and iOS, that appear to be related to system APIs making exceptions for platform destinations in an otherwise effective leak prevention (esp. on iOS). The appearance of race conditions on Android suggests that APIs do not sufficiently ensure a prioritised VPN startup or guide developers towards secure API usage. We found and reported several issues to Apple, Google, and GNOME, including third-party leaks during macOS's supposedly blocking on-demand connection handling. We are continuing to report and discuss found issues with the other vendors.

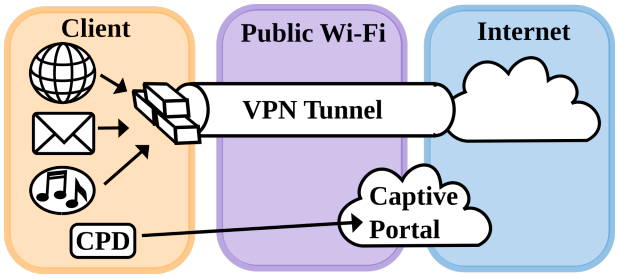

Fig. 1. Stage 1 selective VPN bypass: only CPD traffic is allowed

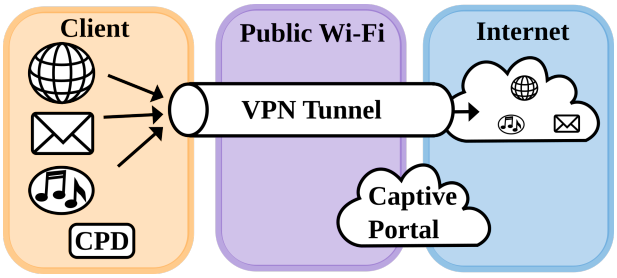

Fig. 2. Stage 3: Only traffic to and from the VPN provider is allowed

Based on our analysis, we are convinced that a secure and private VPN establishment, esp. in public Wi-Fi environments, relies on a deep integration with the OS and its CPD process. It therefore requires high-level system APIs that allow VPN clients to hook themselves into a multistage secure network startup process, which we detail in the following.

VII. Selective VPN bypass for Captive Portals

We propose the following design for a secure VPN startup implementation. The process has to selectively allow outgoing traffic in three stages:

Stage 1: Captive Portal Detection: In this stage, as depicted in Fig. 1, only outgoing CPD requests to a platform's predefined detection server should be allowed. Other outgoing traffic including communication with platform services should be blocked. This could be implemented by restricting networking capabilities to a dedicated CPD process and a minimal, isolated web browser instance to complete the CP sign-in, or by setting up a firewall that blocks all outgoing traffic except to the CPD server and the CP.

Stage 2: Always-on VPN Activation: After the captive network is successfully remediated, the system should trigger the always-on VPN connection establishment and grant further networking capabilities to the VPN process or subsystem. If the VPN connection can be established successfully, the bootstrapping can continue with the next stage. Otherwise, all outbound traffic should remain blocked to avoid leakage and the user should be notified.

Stage 3: Open Connectivity: After a VPN tunnel is established, the system can grant networking capabilities to all other applications and services, as depicted in Fig. 2.

VIII. Conclusion

We are convinced that public Wi-Fis will continue to play an important role in providing mobile Internet connectivity in the future, whilst 5G and fast cellular networks become more ubiquitously available and data plans more affordable. It is therefore important that OS vendors and VPN service providers equip their software to mitigate privacy risks caused by public Wi-Fi usage. Our analyses shows that the vast majority of clients are currently unable to provide VPN establishment without leaking traffic or causing deadlocks when confronted with CPs. We propose a concept that ensures a leak-free VPN establishment in three stages and recommend OS vendors to adopt the proposal.

References

[1] N. Cheng, X. Oscar Wang, W. Cheng, P. Mohapatra, and A. Seneviratne, "Characterizing privacy leakage of public WiFi networks for users on travel," in *2013 Proceedings IEEE INFOCOM*. IEEE, pp. 2769–2777.

[2] N. Sombatruang, L. Onwuzurike, M. A. Sasse, and M. Baddeley, "Factors influencing users to use unsecured wi-fi networks: evidence in the wild," in *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*. ACM, pp. 203–213.

[3] W. A. Kumari, Ólafur Guðmundsson, P. Ebersman, and S. Sheng, "Captive-Portal Identification Using DHCP or Router Advertisements (RAs)," RFC 7710, Dec. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7710.txt

[4] Wikipedia. Captive portal. [Online]. Available: https://en.wikipedia.org/wiki/Captive_portal

[5] S. Ali, T. Osman, M. Mannan, and A. Youssef, "On Privacy Risks of Public WiFi Captive Portals," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, ser. Lecture Notes in Computer Science, C. Pérez-Solà, G. Navarro-Arribas, A. Biryukov, and J. Garcia-Alfaro, Eds. Springer International Publishing, vol. 11737, pp. 80–98.

[6] M. Ikram, N. Vallina-Rodriguez, S. Seneviratne, M. A. Kaafar, and V. Paxson, "An Analysis of the Privacy and Security Risks of Android VPN Permission-enabled Apps," in *Proceedings of the 2016 ACM on Internet Measurement Conference - IMC '16*. ACM Press, pp. 349–364.

[7] J. Wilson, D. McLuskie, and E. Bayne, "Investigation into the security and privacy of iOS VPN applications," in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ser. ARES '20. Association for Computing Machinery.

[8] M. T. Khan, J. DeBlasio, G. M. Voelker, A. C. Snoeren, C. Kanich, and N. Vallina-Rodriguez, "An empirical analysis of the commercial VPN ecosystem," in *Proceedings of the Internet Measurement Conference 2018*. ACM, pp. 443–456.

[9] V. C. Perta, M. V. Barbera, G. Tyson, H. Haddadi, and A. Mei, "A glance through the VPN looking glass: IPv6 leakage and DNS hijacking in commercial VPN clients," vol. 2015, no. 1, pp. 77–91.

[10] R. Karlsson, "Ezmole: A new prototype for securing public wi-fi connections," Master's thesis, Luleå University of Technology, 2017.

[11] Apple. Personal VPN. [Online]. Available: https://developer.apple.com/documentation/networkextension/personal_vpn

[12] ——. Packet Tunnel Provider. [Online]. Available: https://developer.apple.com/documentation/networkextension/packet_tunnel_provider

[13] ——. VPN On Demand Rules. [Online]. Available: https://developer.apple.com/documentation/networkextension/personal_vpn/vpn_on_demand_rules

[14] Android Developers. VPN. [Online]. Available: https://developer.android.com/guide/topics/connectivity/vpn

[15] Proton Team. VPN bypass vulnerability in Apple iOS. [Online]. Available: https://protonvpn.com/blog/apple-ios-vulnerability-disclosure/