Oh-Pwn-VPN! Security Analysis of OpenVPN-based Android Apps [⋆]

Qi Zhang, Juanru Li, Yuanyuan Zhang(✉), Hui Wang, and Dawu Gu

Shanghai Jiao Tong University, Shanghai, China

Abstract. Free VPN apps have gained popularity among millions of users due to their convenience, and have been massively used for accessing blocked sites and preventing network eavesdropping. As a popular open-source VPN solution, OpenVPN is widely used by developers to implement their own VPN services. Despite the prevalence of OpenVPN, it can be insecurely customized and deployed by developers in lack of security guide.
In this paper, we perform a systematic security analysis of 84 popular OpenVPN-based apps on the Google Play store. We analyze the deployment security of OpenVPN on Android from the aspects of client profile, code implementation, and permission management. Our experiment reveals three types of misconfigurations that exist in several apps: insecure customized protocols, weak authentication at the client side, and incorrect file permissions on Android. The misconfigurations found by us can lead to some serious attacks, such as VPN traffic decryption and Man-in-the-Middle attacks, endangering millions of users' privacy. Our work shows that, although OpenVPN protocol itself has withstood security analysis, insecure custom modification and configuration can still compromise the security of VPN apps. We then discuss potential causes of these misconfigurations and make practical recommendations for developers to securely deploy OpenVPN services.

Keywords: OpenVPN, Android Apps, Security Assessment

1 Introduction

Security concerns about network communications of Android apps have been raised in recent years. A straightforward protection approach is to use a virtual private network (VPN) as a secure connection between the device and VPN server over the Internet. VPN services are useful for securely accessing sensitive content in a public network and are commonly used to circumvent censorship. On Android, mobile app developers can use native support to create VPN clients

⋆ This work was partially supported by the Key Program of National Natural Science Foundation of China (Grants No.U1636217), the Major Project of the National Key Research Project (Grants No.2016YFB0801200), and the Technology Project of Shanghai Science and Technology Commission under Grants No.15511103002.

through the Android VPN Service class [2]. Thus many apps legitimately use the VPN permission to offer online anonymity by intercepting and taking full control of the network traffic on device.

However, the use of VPN within an Android app is a new scenario for most developers. Previous researches [22,25,26] have revealed several privacy issues and security flaws in implementations of these VPN services and applications. The most serious security flaw found is the usage of insecure VPN tunneling protocols. Various VPN tunneling protocols are used among different Android VPN-based apps. Despite promising online anonymity and security to their users, many VPN apps still implement unencrypted tunneling protocols. Since implementing a secure VPN tunneling protocol from scratch is sophisticated, a group of VPN apps utilize OpenVPN, the most popular open-source VPN solution [4,17], to build their own VPN services. Because OpenVPN is open-source and has been tested by security analysts over a long period of time, it is generally considered as a secure VPN solution and is widely used on both desktop and mobile platforms.

Although OpenVPN-based apps (in short, OpenVPN apps) are believed to guarantee better security and anonymity compared to those apps with homebrewed tunneling protocols, unfortunately, real world OpenVPN apps are not always secure. On Android platform, how OpenVPN should be incorporated and deployed in these VPN apps is not regulated. Android developers may misuse OpenVPN or modify the original execution flow of it and thus lead to an insecure VPN service.

In this paper, we conduct an in-depth misuse analysis on widely used Android OpenVPN apps. To unveil those misuses, we focus on the variation of OpenVPN apps' tunnel implementations and deployment policies. Our analysis finds that due to three kinds of misuses, the security of the VPN tunnel is weakened or even completely broken. The first one is misuse of modified OpenVPN protocol. Developers add custom operation to the standard OpenVPN protocol implementation, as we called *custom obfuscation*, for the purpose of obfuscating the VPN traffic. The VPN connection is configured to replace the standard OpenVPN encryption with custom obfuscation, and finally leads to an insecure VPN tunnel. The second one is weak authentication at the client side, which leaves the identity of server insecurely validated and finally induces a Man-in-the-Middle attack. The third one is incorrect usage of native library, which assigns an improper privilege to the management interface and finally causes a Denial-of-Service threat. More seriously, the implementation and deployment of these apps cannot be modified by users. Users are generally unaware of relevant security flaws, and can be easily attacked if using such apps to protect their network communications.

We analyzed 84 popular free OpenVPN apps in Android market and found that such misuses widely exist. Among them, 11 apps replace the standard encryption of OpenVPN with custom obfuscation. Due to vulnerable key agreement of the custom obfuscation, VPN traffic can be completely decrypted by attackers. Seven of the apps are susceptible to Man-in-the-Middle (MITM) attacks as a result of weak authentication at the client side. Four of the apps suffer from

Denial-of-Service attacks by reason of unprotected management interface. Our study indicates that even if the VPN apps adopt an robust VPN library, situations of insecure deployment are still common and severely threaten users' security and privacy.

The main contributions of our work are summarized as follows:

- We summarize how OpenVPN is incorporated and utilized by Android VPN apps. We conclude the typical usage of OpenVPN on Android and spot developers' customizations by analyzing popular OpenVPN apps and auditing source code of forked OpenVPN projects.
- We conduct an in-depth analysis of OpenVPN misuses. Our assessment methodology is able to find misconfigurations of OpenVPN apps in the aspects of client profile, code implementation, and permission management.
- We uncover a typical previously unknown security issue in OpenVPN apps. Specifically, we find that some apps add a new tunnel protocol into OpenVPN following the security-by-obscurity policy: these implementations of tunnel modify the original protocol to hide the feature and evade network censorship technologies such as deep packet inspection (DPI). However, our study demonstrates that the modified protocols often adopt vulnerable key agreement that leads to complete insecure communications.

2 Background

2.1 OpenVPN Security Mechanisms

The goal of a VPN system is to provide private communications. To secure the network traffic, OpenVPN has implemented many features for authentication, encryption and management. OpenVPN utilizes SSL as the underlying cryptographic layer for authentication and encryption. There are two channels in OpenVPN: the control channel for authentication and key exchange, and the data channel for traffic encryption. Moreover, OpenVPN also provides an interface for managing the VPN process.

Authentication In the control channel, OpenVPN has two modes of authentication [16]: *a*) **Static key mode** static keys are pre-shared by client and server. All traffic are encrypted by the same static key, thus this mode cannot provide perfect forward security. *b*) **SSL/TLS mode** A mutual authentication is established inside an SSL session. Most security related features are implemented in this mode.

In SSL/TLS mode, the identity of server is validated by its certificate the same way as in HTTPS. Meanwhile, multiple ways of client authentication are provided by OpenVPN. The client can be authenticated by the traditional username/password mechanism, by client certificate, or by the combination of these two types. To mitigate possible vulnerabilities in the TLS handshake, e.g., the famous *Heartbleed* bug [7], additional HMAC of TLS control channel packets can be required by enabling *tls-auth* option. After the authentication, session keys are generated by Diffie-Hellman key exchange and updated periodically.

Encryption After authentication and session key generation, OpenVPN uses data channel to tunnel the actual network traffic. *Encrypt-then-Mac* scheme is used to protect data channel packets. Specifically, the encryption and HMAC algorithm are determined by option *cipher* and *auth* in the configuration file. The cipher algorithm used for data encryption must be specified at both the client and the server side.
Management OpenVPN provides a management interface [11] that allows itself to be administratively controlled from an external program via a TCP or UNIX domain socket. Control commands such as *setting proxy address*, *providing passwords* or *suspending VPN service* can be transmitted through the management interface.

2.2 OpenVPN on Android

Since Android version 4.0, the *VpnService* API [2] is provided for developers to build their own VPN solutions. This API returns a descriptor of a virtual network interface (the *tun* interface) for apps to read and modify all the network traffic on device. While the *VpnService* API makes it convenient for developers to build VPN services, malicious apps may use this API to eavesdrop network activity of other apps. Android system takes several actions to prevent the abuse of VPN Service API. To obtain the VPN interface by using this API, apps have to request the *BIND_VPN_SERVICE* permission, and the first time a VPN connection is created, Android alerts users by displaying system dialog and notification.

Typically, OpenVPN is ported to Android platform as a shared library. For instance, *ics-openvpn* [10], a popular open-source OpenVPN app, implements OpenVPN as an ELF shared library *libopenvpn.so*. The shared library is invoked by a native process on Android (the native layer). Other functions of the app are usually implemented at the Java layer. To handle the Inter-Process Communication (IPC) between the Java layer (i.e., the UI thread) and the native layer (i.e., the OpenVPN process), UNIX domain socket is adopted to implement the management interface.

The execution flow of an OpenVPN app is depicted in Figure 1 and divided into four steps: *profile assembly*, *VPN initiation*, *management interaction* and *VPN connection*. The client profile for the OpenVPN app is retrieved in various ways (step 1). Based on the configuration file, OpenVPN process is initiated at the native layer (step 2), then the Java layer controls the OpenVPN process via the management interface (step 3) and the VPN tunnel is established by the OpenVPN process (step 4). Details of these steps are described as follows:

Step 1: The VPN client assembles a configuration profile for connecting to a remote VPN server. Note that this step can be implemented differently by VPN providers. The client can directly obtain the configuration from the APK file, or the client retrieves VPN server address from a server (the profile server), and then assembles a complete configuration at the client side, or the configuration is fully downloaded from the profile server. Other options such as file location of the management interface and protocol used (TCP or UDP) are also included

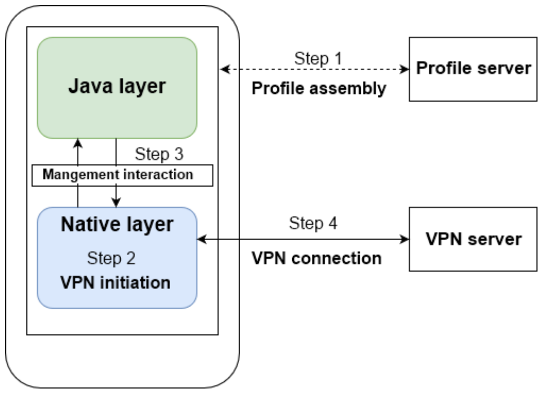


Fig. 1: A typical workflow of Android OpenVPN apps

in the configuration profile. An example of Android OpenVPN client profile is shown in Figure 2.

Step 2: The OpenVPN library is loaded and invoked by a native process. Based on the client profile, the OpenVPN process is initiated. Network parameters of the *tun* interface such as IP address, DNS server are pushed from the VPN server. As shown in Figure 2, the *management-client* option is enabled, thus OpenVPN process acts as the client and the management interface is created by the Java layer. The management interface is created in the app's private directory (e.g., */data/data/pkg.name/cache/*) and is waiting for connection.

Step 3: The OpenVPN process connects to the management interface and then it is controlled by the UI thread. To utilize the Android *VpnService* API, the Java layer sends several commands to the OpenVPN process to gather network parameters of the *tun* interface. After the call of *VpnService*, the descriptor of the *tun* interface is sent from the Java layer to the native layer and the descriptor of the link to VPN server is sent in the reverse direction.

Step 4: The OpenVPN process has obtained two descriptors for controlling the traffic between the device and the remote server. After that, network traffic on device is tunneled inside the VPN connection.

```
management /data/data/pkg.name/cache/mgmtsocket unix
management-client
client
remote vpn.server.address
cipher BF-CBC
ca  ca.crt
cert client.crt
key client.key
```

Fig. 2: An example of Android OpenVPN client profile

3 Attacking OpenVPN Apps

3.1 Adversary Model

Our adversary model consists of two types of attackers:

1. **A network attacker** can passively monitor the traffic, or can actively intercept and modify network connections between the client and OpenVPN server. Mobile devices are commonly used under different network environments. Users may connect to a free public Wi-Fi for convenient Internet access, and then protect network activity by using a VPN app. The public Wi-Fi could be controlled by a network attacker, thus the VPN traffic can be observed and manipulated by the attacker.
2. **A malicious app** that attempts to attack OpenVPN apps is installed on the user device. Apps installed on the user device cannot be all trusted. Users may install apps from third-party app markets, where the attacker can repackage malicious payload into popular apps and distribute them.

3.2 Vulnerabilities and Attacks

From the attacker's perspective, the profile distribution (step 1) is the critical step for discovering vulnerabilities in the execution flow of OpenVPN apps. Most security related information can be found from the VPN client profile, which can be obtained after step 1. The client profile provides all the prerequisites for attacking the management interaction and the VPN connection procedure, such as the address of VPN server, cipher algorithm, authentication types and file location of the management interface. Without these critical information, it is impossible for attackers to find vulnerabilities in other steps of the OpenVPN workflow.

Free VPN apps indeed expose VPN client profiles to attackers, which makes conducting a certain attack feasible. Most free VPN apps do not require user registration, or some even provide same private key for different users [29]. Any user can obtain a valid client profile, by just connecting to the VPN servers in these apps. The attacker can utilize the profile distribution step of these apps on his own device to collecting the configuration profiles of VPN clients. Except client credentials like certificates and private keys which may be user-unique, the attacker can obtain the same configuration as other normal users due to the same client implementation and server logic. After that, the attacker can explore configuration profiles and client implementations to find vulnerabilities and attack specific VPN apps.

Based on our adversary model and the leakage of client profile, we present three types of attacks against OpenVPN apps, which compromise the confidentiality, authenticity and availability of the OpenVPN service. These attacks are caused by insecure customization and deployment of OpenVPN apps, not by OpenVPN protocol itself.

1. **Traffic Decryption** Some VPN service providers claim that they use some proprietary VPN protocols or Anti-DPI [9,21] technology to prevent VPN traffic from being identified or blocked. Also, a few custom OpenVPN patches intended to obfuscate the OpenVPN traffic and bypass firewalls have been proposed in the OpenVPN community [19] and GitHub [20]. We identify a typical misuse that developers disable the encryption of OpenVPN and use custom obfuscation to replace the standard encryption. These custom obfuscations are commonly implemented by scramble operations such as XOR, and adopt vulnerable key agreements (e.g., hard-coded keys). Thus the misconfiguration of replacing standard encryption with custom obfuscation will lead any passive network attacker to completely decrypt the VPN traffic. Details of custom obfuscation and its misconfiguration are discussed in Section 5.1.
2. **Man-in-the-Middle Attack** The publicly available client profiles of these free VPN apps may lead to possible MITM attacks. This MITM attack happens when the client certificate is signed by the same CA of server certificate and the usage of server's certificate is not verified at the client side. A valid client certificate and private key are sufficient to conduct this MITM attack if the OpenVPN app is misconfigured. An active network attacker can truncate the connection request from the client, then claim to be the server by using a valid client certificate. OpenVPN provides several ways to defend this attack [13], however, developers may not enable these security features, leaving their apps vulnerable to this attack.
3. **Denial of Service** Besides network attackers, threats can also come from a malicious app at the client side. Since Android is a multi-app platform, improper permission of the management interface may allow other apps on the same device to control the OpenVPN process, prevent the normal connection and cause a Denial-of-Service attack.

4 Methodology

This section describes our approach of analyzing the deployment security of Android OpenVPN apps in consideration of the three attacks we proposed. Figure 3 illustrates the procedure of our analysis. In detail, our approach consists of three phases: OpenVPN identification, profile collection, and security assessment. Most prior studies on security and privacy of VPN services focus on the network traffic. However, security flaws in code implementation and permission management can also break the security of OpenVPN. We propose a comprehensive assessment methodology that evaluates OpenVPN apps from three aspects: client profile, code implementation and permission management.

4.1 OpenVPN Identification

Given a set of Android VPN permission-enabled apps, we need to determine the tunneling protocols used by them. We propose two general methods to identify

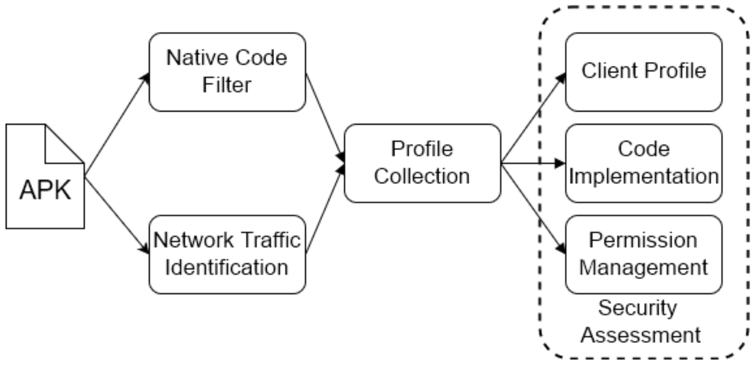


Fig. 3: An overview of how we analyze the security of OpenVPN apps

the usage of OpenVPN among these VPN apps: native code filter and network traffic identification.

1. **Native Code Filter** This method filters OpenVPN apps by inspecting the symbol table of native libraries. Since cryptographic operations inside VPN service are CPU-intensive, VPN protocols on Android are commonly implemented in native code. If the OpenVPN library is incorporated in the native libraries of the app, function names and other symbol information from OpenVPN source code are preserved in the binary code. By searching meaningful strings like function name *openvpn_encrypt* in the symbol table of shared library files, we can quickly determine whether OpenVPN is used in the native code. This static method is efficient, and is fully automatic. If symbols in the library are stripped, or developers intentionally obfuscate the binary code, this method cannot detect the usage of OpenVPN, thus we need runtime traffic identification.
2. **Network Traffic Identification** This method focuses on investigating network activity of apps and treats APK files as a black box. After capturing the network traffic of VPN apps, the tunneling protocol can be identified by using protocol parsers such as Bro [3]. The accuracy of this method depends on the precision of the protocol parser. As reported in the work by Ikram et al. [25], a large portion of the tunneling protocols used by VPN apps cannot be recognized by protocol parsers. If a VPN app has obfuscated its traffic by modifying the protocol implementation, common protocol analysis tools are incapable of identifying its network traffic. While this method cannot detect custom obfuscations and needs manual interference to establish VPN connection in apps, it helps us to find the usage of typical OpenVPN implementation regardless of the binary code information.

In this step our goal is to identify OpenVPN apps as many as possible, therefore we combine these two methods. We adopt native code filter as the main detection method, which narrows the assessment scope automatically, and is capable of detecting custom OpenVPN implementations. Then we utilize network analysis tools for those apps that are not identified by our native code filter.

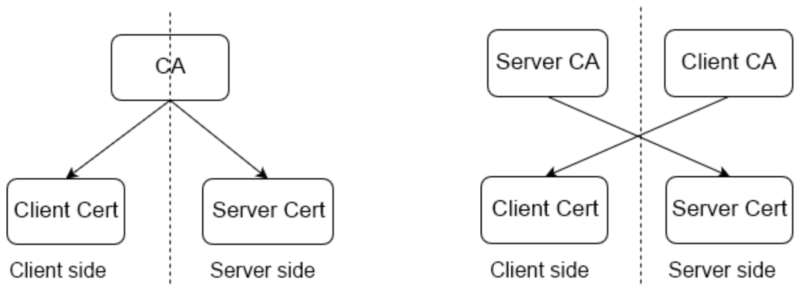
4.2 Profile Collection

As discussed before, the profile distribution of different apps can vary from each other, therefore it is complicated to collect client profiles from profile servers. Instead, for each OpenVPN app, we gather the runtime arguments of the OpenVPN process to figure out the client profile.

OpenVPN allows options to be provided either by the command line arguments or by a configuration file. Actually the configuration file is used as a command line option *–config*. Therefore, by inspecting the app's native process and its command line arguments (i.e. */proc/PID/cmdline*), we are able to extract the client connection configuration of VPN apps.

We build a semi-automatic tool for collecting client profiles of OpenVPN apps. While the VPN service is running, this tool automatically parses the arguments of OpenVPN process belong to each VPN app, then it records all the configurations or directly extracts the configuration file on device. The necessary manual part is that for each app we need to actively connect to the VPN server and approve the VPN connection in the Android system dialog.

4.3 Security Assessment



(a) Same CA for client and server (b) Different CAs for client and server

Fig. 4: Two types of CA trust model

1. **Client profile** We implement a parser of OpenVPN profile to extract all the options from configuration files which we collect from different apps. Then, we perform a statistical survey of the usage of security related options. Particularly, we inspect the cipher algorithm used in OpenVPN apps (*cipher*), whether the client is authenticated by passwords (*auth-user-pass*), by certificates (*cert*), or additional TLS authentication is used (*tls-auth*). When

client and server are authenticated by certificates, there are two types of CA trust model, as shown in Figure 4. The certificate of client and server can be signed by the same CA (Figure 4a), or by different CAs (Figure 4b). Under the CA model in Figure 4a, if the usage of certificate is not checked at the client side, an active network attacker can impersonate a valid server by using another client certificate retrieved from the OpenVPN app. By checking whether the client certificate (*cert*) is signed by the CA (*ca*), we determine the CA model of authentication in the OpenVPN app. If the certificate of client and server is signed by the same CA, we then examine whether options for MITM prevention (*remote-cert-tls*, *ns-cert-type*) are applied.

2. **Code implementation** We focus on evaluating the implementation of custom features added for the purpose of obfuscating OpenVPN traffic. By selecting options that exist in these configuration profiles but not in the official manual page of OpenVPN [12], we are able to filter the custom features. In order to understand these custom obfuscation behaviors, we perform reverse engineering on the modified OpenVPN library. To target the obfuscation operations, we concentrate on the code implementation located after the original cryptographic procedure in OpenVPN, and before the actual network send/receive logic, e.g., *process_outgoing_link* or *read_incoming_link* functions in OpenVPN source code [17]. Utilizing the source code of OpenVPN and comparing it with the decompiled code generated by IDA Pro [8], we identify custom obfuscations added in these OpenVPN apps. After that, the obfuscation key is examined in the client profile.
3. **Permission management** File location of the management interface can be obtained from the client profile. While the OpenVPN service is running, We use a script to automatically examine the permission of its management interface. The permission management is vulnerable if the management interface is world-accessible.

5 Result and Security Analysis

We analyzed top 200 VPN apps collected from Google Play in May 2017. We utilize *google-play-scraper* [6] and *gplaycli* [5] to select and download VPN apps. *Google-play-scraper* provides the feature of searching popular apps matching a certain term, and *gplaycli* is able to automatically download a list of APK files from the Google Play store. Among the top 200 VPN apps on Google Play, 111 apps using OpenVPN were identified by our methods. We successfully analyzed the client deployment status of 84 OpenVPN-based apps. The remaining 27 apps were not evaluated due to the need of in-app purchase or the failure of server connection. All the apps were tested on a rooted Moto G with Android 7.1.

The main vulnerabilities we found are summarized in Table 1. 11 of the analyzed apps replace the standard OpenVPN encryption with custom obfuscation, thus the VPN traffic can be decrypted by network attackers. There are seven apps vulnerable to MITM attacks due to their lack of certificates usage validation or using the static key mode. Four of the OpenVPN apps leave the management interface unprotected, which may lead to Denial-of-Service attacks. The maximum number of installs among the apps belong to each misconfiguration type is also listed in Table 1.

Table 1: Main vulnerabilities we found in OpenVPN apps

Category	Vulnerability Type	# of Apps	Max Installs	Consequence
cipher	replacing encryption with obfuscation	11	1M	traffic decryption
	encryption disabled	2	1M	traffic in plain text
auth	lacking cert usage validation	6	1M	MITM
	static key mode	1	100K	traffic decryption, MITM
management	unprotected interface	4	1M	DoS

5.1 Insecure Encryption

Insecure Cipher Algorithm We found that 30 OpenVPN apps use the default cipher algorithm BF-CBC. As a 64-bit block cipher, Blowfish is vulnerable to the SWEET32 attack [23], thus it is no longer recommended. Despite the publication of this attack, Blowfish is still the default cipher algorithm of OpenVPN [12]. This *insecure-by-default* setting may influence the security of OpenVPN deployment.

Meanwhile, two of the analyzed apps explicitly set cipher to *none*, which disables encryption and transfers all traffic in plain text. When using option *cipher none* , OpenVPN has a warning in its standard output. However, since Android users cannot notice this warning, they will be unaware of this insecure setting.

Replace Encryption with Obfuscation The usage of custom obfuscation patches in the 84 OpenVPN apps is described in Table 2. Obfuscation is realized by adding extra encryptions of the OpenVPN packet data, and the key for obfuscation needs to be configured the same at both client and server side. We notice that 13 apps use RC4 to obfuscate the OpenVPN traffic, and the key of RC4 is set to the IP address of VPN server. Obfuscation itself does not weaken the security of OpenVPN. However, 11 of them **use custom obfuscation to replace standard encryption by setting cipher to *none***, which completely breaks the security of OpenVPN. Besides, nine apps use XOR-based obfuscation while four of them choose the same obfuscation key. There is only one app that uses two's complement to obfuscate the traffic.

Furthermore, we perform a thorough analysis of the most commonly used custom obfuscation option *antidpi*. Before OpenVPN's network send/receive logic, a custom RC4 encryption/decryption of the whole OpenVPN packet is added. The key of RC4 is determined by the argument of *antidpi_remote*. The crux deployment security problem is that 11 apps disable the standard encryption of OpenVPN and set a poor key for the RC4 encryption. OpenVPN's own encryption is disabled by setting *cipher none* and the key of RC4 is set to server's IP

Table 2: The usage of custom obfuscation patches

Obfuscation Type	**Option Name**	**Obfuscation Key**	**# of Apps**
RC4	*antidpi,antidpi_remote*	Server's IP address	13
XOR	*obsecure_key,scramble,link-key*	Random string	9
Two's complement	*ob-key*	-	1

address by setting *antidpi_remote*. In this case, users and developers who believe network data is protected will be in fact fully exposed to threats. Any passive attacker that obtains network traffic of these apps can completely decrypt the data and recover users' online activity, e.g., the attacker can learn all the HTTP traffic.

We investigated several forked OpenVPN projects to analyze potential causes of the misconfiguration of custom obfuscation. A custom obfuscation patch called *xorpatch* in OpenVPN community forum claims that encryption of the scramble patch is secure thus *'it is OK to use cipher none'* [19]. Due to neglecting the importance of encryption, the patch author gives an insecure demonstration of configuration, which may mislead other developers to make the same mistake. In addition, we audited some OpenVPN patches on GitHub [14,15,20]. All of them are in lack of guide about how to set the key for obfuscation. Users may not know how to set the correct patched options with OpenVPN's original features, thus weak arguments like using IP address as the key may occur.

5.2 Weak Authentication

Table 3: The usage of client authentication methods

Password	**Certificate**	**TLS-auth**	**# of apps**
$\surd$	$\surd$	$\surd$	18
$\times$	$\surd$	$\surd$	4
$\surd$	$\times$	$\surd$	8
$\surd$	$\surd$	$\times$	29
$\times$	$\surd$	$\times$	10
$\surd$	$\times$	$\times$	14

Table 3 presents the usage of different client authentication methods of OpenVPN apps. Different security levels are provided by these authentication methods. *Password+Certificate+TLS-auth* is the most secure method, which is adopted by 18 apps, while 14 apps use less secure *Password only* authentication. Besides, one app is found to use static key mode. In this mode OpenVPN connection can be decrypted and MITM-ed as the key is shared by different users.

We observe that *Password+Certificate* is the most used type. In our results, 61 apps use certificate-based authentication and 39 of them use the same CA trust model. It is convenient to use the CA trust model in Figure 4a since the CAs deployed at client and server side are the same. However, six apps lack the validation of server certificate usage, which means a rogue client can conduct MITM attacks on other clients.

The trust model in Figure 4b is immune to this attack due to different CAs are used. To prevent this attack against the same CA model, extra validation of server's certificate usage (i.e., certificate for server only) is needed. OpenVPN has provided several options like *remote-cert-tls* to require an explicit key usage of peer certificates. These security protections are not enabled by default since in most case private keys are not leaked. While for most free OpenVPN apps, client certificates and private keys are publicly available, thus developers must apply these options to prevent MITM attacks.

Due to lack of security awareness, developers of these free VPN apps usually make their VPN client profiles public, or even provide the same client credential for different users. While providing convenient VPN services, these VPN apps leak the client profiles at the same time.

The *insecure-by-default* policy in OpenVPN may also cause this misconfiguration. OpenVPN provides various of options, some are required for enabling the basic VPN function, some are for security hardening purpose. Developers may omit these complicated security protection options, leaving their OpenVPN service insecure by default.

5.3 Unprotected Management Interface

In the circumstance of OpenVPN on Android, the management interface handles the communication between native and Java layer. The problem is that, the management interface itself does not have any authentication mechanism, thus the file permission of the interface must be correctly set. Otherwise it can can be exploited by other apps on the same device.

In our experiment, four of the analyzed apps set the permission of management interface world-accessible, i.e., *srwxrwxrwx*. Because the file location of management interface can be inferred from the client profile, a malicious app on the same device can exploit the insecure permission of the interface, and access to it before the normal connection. The implementation of OpenVPN management interface does not support multiplex, thus the first connection will block others from accessing the interface. After the malicious app connects to the management interface, the normal connection is blocked and this eventually leads to a Denial-of-Service attack.

The misuse of Android UNIX domain socket has been analyzed by [28], here we focus on OpenVPN and explore the causes of unprotected file permission. We observe that *management-client* option is not used by the four apps, while all other apps enable this option. The typical execution flow of OpenVPN is modified by disabling this option. When *management-client* is disabled, the native process, not the Java process, acts as the server of UNIX domain socket. The management interface is created at the native layer and the OpenVPN process listens on the UNIX domain socket. We find that the default file permission of the UNIX domain socket in OpenVPN is world-accessible because *umask(0)* is used by OpenVPN [17]. To protect the UNIX domain socket and only allow specific user to access the interface, developers need to enable the *management-client-user* option, which specifies the file permission of the management interface.

On the other side, if *management-client* is enabled, the Java layer is responsible for creating the management interface. At Java layer the security model is supplied by Android Java VM and file is created with correctly protected default permission. In a word, the different default file permissions of native layer and Java layer result in this vulnerability.

6 Recommendations

Don't use custom obfuscation to replace encryption. Custom obfuscation is commonly implemented by simple scramble operations thus it is not secure enough to replace the standard encryption of OpenVPN. The purpose of obfuscation is to hide protocol metadata, not to protect the payload. For bypassing network censorship, the OpenVPN team disapproves of custom patches and suggests to use *obfsproxy* [18]. Another approach is to tunnel the VPN traffic in common secure protocols like TLS or SSH.

Deploy countermeasures against MITM. OpenVPN provides different ways to avoid the Man-in-the-Middle attack from an authorized client. Certificates can be assigned with specific key usage and extended key usage. Options like *remote-cert-tls server* or *ns-cert-type server* make OpenVPN clients accept server-only certificates. Signing certificates for server and client with different CAs can also prevent this MITM attack.

Set secure file permissions on Android. Since Android is a multi-app platform, developers should protect their own files from being tempered by other apps on the same device. File permission at Java layer is correctly protected by default. However, at the native layer, developers should take their own responsibility and use *umask* and *chmod* to securely protect their files.

Securely distribute client profiles. Developers should harden the client configuration, protect the distribution of client profiles (e.g., transmit them via email) and securely store them at the client side. Unique client credentials should be generated for different users to prevent the abuse of public client profiles. To achieve a better security level, VPN profiles can be encrypted or stored in Android *Keystore* [1].

7 Related Work

Several studies have been working on the privacy and security of VPN services. Appelbaum et al. [22] are the first to uncover the VPN traffic leakage problem caused by misconfiguration of route tables. Perta et al. [26] extend their work and analyze popular commercial VPN services. Their results reveal that the majority of VPN services suffer from IPv6 leakage and DNS hijack attacks. Ikram et al. [25] conduct a comprehensive privacy and security analysis of Android VPN permission-enabled apps. Their study mainly focuses on investigating VPN apps' manipulation of TLS traffic and behavior of tracking user privacy. Instead of concentrating on network analysis, our work evaluates the security of VPN apps from the aspects of security related configuration and code implementation at the client side. Recently OpenVPN 2.4.0 has been audited and several security issues have been found [27]. Our work reveals that, in addition to the flaws in official implementations, developers' custom modification and configuration in VPN applications can also lead to severe security vulnerabilities.

There are some studies about the security of custom VPN protocols and misuses of UNIX domain sockets on Android. Gutmann [24] gives a classic cryptographic audience of the weakness of some custom VPN protocols. He suggests to use standard-protocol-based VPN, such as OpenVPN and IPsec, while we demonstrate that misuses of OpenVPN can still threaten the VPN communication. Shao et al. [28] conduct a systematic study of the misuses of Android UNIX domain sockets. We analyze the causes of insecure permission of OpenVPN management interface based on their work.

8 Conclusion

In this work, we focus on the client side deployment security of Android OpenVPN apps. After summarizing the procedure of client deployment and VPN connection, we present a security assessment methodology by evaluating the security of client profile, code implementation and permission management. The configuration status of 84 popular OpenVPN-based apps on Google Play are analyzed. To our best knowledge, we are the first to identify a typical misuse of insecure custom obfuscation in several OpenVPN apps. Our experiment also shows that MITM vulnerability and Denial-of-Service problem due to misconfigurations still exists in these apps. The misconfigurations are either due to patch authors' wrong advices and lacking of document, the 'insecure-by-default' OpenVPN configuration, or due to developers' incorrect file permission setting on Android. Finally we develop some practical recommendations for securing the OpenVPN deployment.

References

1. Android keystore system. `https://developer.android.com/reference/java/security/KeyStore.html`.
2. Android vpn service documentation. `https://developer.android.com/reference/android/net/VpnService.html`.
3. Bro network security monitor. `https://www.bro.org`.
4. Detailed vpn comparison chart. `https://thatoneprivacysite.net/vpn-comparison-chart/`.
5. Google play downloader via command line. `https://github.com/matlink/gplaycli`.

6. Google-play-scraper. https://github.com/facundoolano/google-play-scraper.
7. The heartbleed bug. http://heartbleed.com/.
8. Ida pro. https://www.hex-rays.com/products/ida/.
9. Nvpn antidpi. http://www.nvpn.net/. Accessed: 2017-07-21.
10. Openvpn for android source code. https://github.com/schwabe/ics-openvpn.
11. Openvpn management interface. https://openvpn.net/index.php/open-source/documentation/miscellaneous/79-management-interface.html.
12. Openvpn manual page. https://community.openvpn.net/openvpn/wiki/Openvpn24ManPage.
13. Openvpn mitm protection. https://openvpn.net/index.php/open-source/documentation/howto.html#mitm.
14. Openvpn obfuscation patch. https://github.com/siren1117/openvpn-obfuscation-release/.
15. Openvpn patch from tunnelblick. https://github.com/Tunnelblick/Tunnelblick/tree/master/third_party/sources/openvpn/openvpn-2.4.3/patches.
16. Openvpn security overview. https://openvpn.net/index.php/open-source/documentation/security-overview.html.
17. Openvpn source code. https://github.com/OpenVPN/openvpn.
18. Openvpn traffic obfuscation guide. https://community.openvpn.net/openvpn/wiki/TrafficObfuscation.
19. Xorpatch in openvpn forum. https://forums.openvpn.net/viewtopic.php?f=15&t=12605&hilit=openvpn_xorpatch.
20. Xorpatch source code. https://github.com/clayface/openvpn_xorpatch.
21. Zpn antidpi. https://zpn.im/blog/total-anonymity-connectivity-antidpi. Accessed: 2017-07-21.
22. Jacob Appelbaum, Marsh Ray, Karl Koscher, and Ian Finder. vpwns: Virtual pwned networks. In *2nd USENIX Workshop on Free and Open Communications on the Internet. USENIX Association*, 2012.
23. Karthikeyan Bhargavan and Gaëtan Leurent. On the practical (in-) security of 64-bit block ciphers: Collision attacks on http over tls and openvpn. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 456–467. ACM, 2016.
24. Peter Gutmann. Linux's answer to ms-pptp. https://www.cs.auckland.ac.nz/~pgut001/pubs/linux_vpn.txt.
25. Muhammad Ikram, Narseo Vallina-Rodriguez, Suranga Seneviratne, Mohamed Ali Kaafar, and Vern 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*, pages 349–364. ACM, 2016.
26. Vasile C Perta, Marco V Barbera, Gareth Tyson, Hamed Haddadi, and Alessandro Mei. A glance through the vpn looking glass: Ipv6 leakage and dns hijacking in commercial vpn clients. *Proceedings on Privacy Enhancing Technologies*, 2015(1):77–91, 2015.
27. Quarkslab. Security assessment of openvpn. https://blog.quarkslab.com/security-assessment-of-openvpn.html. Accessed: 2017-07-21.
28. Yuru Shao, Jason Ott, Yunhan Jack Jia, Zhiyun Qian, and Z Morley Mao. The misuse of android unix domain sockets and security implications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 80–91. ACM, 2016.
29. Kenneth White. Most vpn services are terrible. https://gist.github.com/kennwhite/1f3bc4d889b02b35d8aa. Accessed: 2017-07-21.