[CVE-2019-14899] Inferring and hijacking VPN-tunneled TCP connections.

From: "William J. Tolley" <william () breakpointingbad com>
Date: Wed, 04 Dec 2019 19:37:07 -0700

Hi all,

I am reporting a vulnerability that exists on most Linux distros, and other *nix operating systems which allows a network adjacent attacker to determine if another user is connected to a VPN, the virtual IP address they have been assigned by the VPN server, and whether or not there is an active connection to a given website. Additionally, we are able to determine the exact seq and ack numbers by counting encrypted packets and/or examining their size. This allows us to inject data into the TCP stream and hijack connections.

Most of the Linux distributions we tested were vulnerable, especially Linux distributions that use a version of systemd pulled after November 28th of last year which turned reverse path filtering off. However, we recently discovered that the attack also works against IPv6, so turning reverse path filtering on isn't a reasonable solution, but this was how we discovered that the attack worked on Linux.

Adding a prerouting rule to drop packets destined for the client's virtual IP address is effective on some systems, but I have only tested this on my machines (Manjaro 5.3.12-1, Ubuntu 19.10 5.3.0-23). This rule was proposed by Jason Donenfeld, and an analagous rule on the output chain was proposed by Ruoyu "Fish" Wang of ASU. We have some concerns that inferences can still be made using slightly different methods, but this suggestion does prevent this particular attack.

There are other potential solutions being considered by the kernel maintainers, but I can't speak to their current status. I will provide updates as I receive them.

I have attached the original disclosure I provided to distros () vs openwall org and security () kernel org below, with at least one critical correction: I orignally listed CentOS as being vulnerable to the attack, but this was incorrect, at least regarding IPv4. We didn't know the attack worked against IPv6 at the time we tested CentOS, and I haven't been able to test it yet.

William J. Tolley
Beau Kujath
Jedidiah R. Crandall

Breakpointing Bad &
University of New Mexico

****

**General Disclosure:

We have discovered a vulnerability in Linux, FreeBSD, OpenBSD, MacOS, iOS, and Android which allows a malicious access point, or an adjacent user, to determine if a connected user is using a VPN, make positive inferences about the websites they are visiting, and determine the correct sequence and acknowledgement numbers in use, allowing the bad actor to inject data into the TCP stream. This provides everything that is needed for an attacker to hijack active connections inside the VPN tunnel.

This vulnerability works against OpenVPN, WireGuard, and IKEv2/IPSec, but has not been thoroughly tested against tor, but we believe it is not vulnerable since it operates in a SOCKS layer and includes authentication and encryption that happens in userspace. It should be noted, however, that the VPN technology used does not seem to matter and we are able to make all of our inferences even though the responses from the victim are encrypted, using the size of the packets and number of packets sent (in the case of challenge ACKs, for example) to determine what kind of packets are being sent through the encrypted VPN tunnel.

We have already reported a related vulnerability to Android earlier this year related to the issue, which resulted in the assignment of CVE-2019-9461, however, the CVE strictly applies to the fact that the Android devices would respond to unsolicited packets sent to the user's virtual IP address over the wireless interface, but this does not address the fundamental issue of the attack and did not result in a change of the reverse path settings of Android as of the most recent security update.

This attack did not work against any Linux distribution we tested until the release of Ubuntu 19.10, and we noticed that the rp_filter settings were set to "loose" mode. We see that the default settings in sysctl.d/50-default.conf in the systemd repository were changed from "strict" to "loose" mode on November 28, 2018, so distributions using a version of systemd without modified configurations after this date are now vulnerable. Most Linux distributions we tested which use other init systems leave the value as 0, the default for the Linux kernel.

We have described the procedure for reproducing the vulnerability with Linux and included a section illustrating the differences in architecture.

There are 3 steps to this attack:

1. Determining the VPN client's virtual IP address
2. Using the virtual IP address to make inferences about active connections
3. Using the encrypted replies to unsolicited packets to determine the sequence and acknowledgment numbers of the active connection to hijack the TCP session

There are 4 components to the reproduction:

1. The Victim Device (connected to AP, 192.168.12.x, 10.8.0.8)
2. AP (controlled by attacker, 192.168.12.1)
3. VPN Server (not controlled by attacker, 10.8.0.1)
4. A Web Server (not controlled by the attacker, public IP in a real-world scenario)

The victim device connects to the access point, which for most of our testing was a laptop running create_ap. The victim device then establishes a connection with their VPN provider.

The access point can then determine the virtual IP of the victim by sending SYN-ACK packets to the victim device across the entire virtual IP space (the default for OpenVPN is 10.8.0.0/24). When a SYN-ACK is sent to the correct virtual IP on the victim device, the device responds with a RST; when the SYN-ACK is sent to the incorrect virtual IP, nothing is received by the attacker.

To quickly demonstrate this difference, we use the nping commands on the AP device running create_ap. The source IP is the gateway of our AP, the destination IP is the virtual IP assigned to the tun interface by the VPN client, ap0 is the interface create_ap created on the attacker device, and the destination MAC is the victim's wireless MAC address.

For example:

The correct address generates a RST from the victim:

```
nping --tcp --flags SA --source-ip 192.168.12.1 --dest-ip 10.8.0.8 --
rate 3 -c 3 -e ap0 --dest-mac 08:00:27:9c:53:12
```

The incorrect address does not elicit a response from the victim:

```
nping --tcp --flags SA --source-ip 192.168.12.1 --dest-ip 10.8.0.9 --
rate 3 -c 3 -e ap0 --dest-mac 08:00:27:9c:53:12
```

Similarly, to test if there is an active connection for any given website, such as 64.106.46.56, for example, we send SYN or SYN-ACKs from 64.106.46.56 on port 80 (or 443) to the virtual IP of the victim across the entire ephemeral port space of the victim. The correct four-tuple will elicit no more than 2 challenge ACKs per second from the victim, whereas the victim will respond to the incorrect four-tuple with a RST for each packet sent to it.

To quickly test this, we suggest creating a netcat connection on the victim device, such as this:

```
Netcat 64.106.46.56 80 -p 40404
```

The correct four-tuple generates challenge ACKs

```
nping --tcp --flags SA --source-ip 64.106.46.56 -g 80 --dest-ip
10.8.0.8 -p 40404 --rate 10 -c 10 -e ap0 --dest-mac 08:00:27:9c:53:12
```

The incorrect four-tuple generates a single RST for each packet sent:

```
nping --tcp --flags SA --source-ip 64.106.46.56 -g 80 --dest-ip
10.8.0.8 -p 40405 --rate 10 -c 10 -e ap0 --dest-mac 08:00:27:9c:53:12
```

Finally, once the attacker determined that the user has an active TCP connection to an external server, we will attempt to infer the exact next sequence number and in-window acknowledgment number needed to inject forged packets into the connection. To find the appropriate sequence and ACK numbers, we will trigger responses from the client in the encrypted connection found in part 2. The attacker will continually spoof reset packets into the inferred connection until it sniffs challenge ACKs. The attacker can reliably determine if the packets flowing from the client to the VPN server are challenge ACKs by looking at the size and timing of the encrypted responses in relation to the attacker's spoofed packets. The victim's device will trigger a TCP challenge ACK on each reset it receives that has an in-window sequence number for an existing connection. For example, if the client is using OpenVPN to exchange encrypted packets with the VPN server, then the client will always respond with an SSL packet of length 79 when a challenge ACK is triggered.

The attacker must spoof resets to different blocks across the entire sequence number space until one triggers an encrypted challenge ACK. The size of the spoof block plays a significant role in how long the sequence inference takes, but should be conservative as to not skip over the receive window of the client. In practice, when the attacker thinks it sniffs an encrypted challenge-ACK, it can verify this is true by spoofing X packets with the same sequence number. If there were X encrypted responses with size 79 triggered, then the attacker knows for certain it is triggering challenge ACKs (at most 2 packets of size 79 per second).

After the attacker has inferred the in-window sequence number for the client's connection, they can quickly determine the exact sequence number and in-window ACK needed to inject. First, they spoof empty push-ACKs with the in-window sequence while guessing in-window ACK numbers. Once the spoofed packets trigger another challenge-ACK, an in-window ACK number is found. Finally, the attacker continually spoofs empty TCP data packets with the in-window ACK and sequence numbers as it decrements the sequence number after each send. The victim will respond with another challenge ACK once the attacker spoofs the exact sequence number minus one. The attacker can now inject arbitrary payloads into the ongoing encrypted connection using the inferred ACK and next sequence number.

This can be tested by observing the behavior from this sequence of commands, continuing with the same four-tuple:

Using the four-tuple from the previous steps, we send RSTs in the sequence number range in blocks of 50,000 until we trigger a challenge ACK.

```
nping --tcp --flags R --source-ip 64.106.46.56 -g 80 --dest-ip 10.8.0.8
-p 40404 --rate 10 -c 10 -e ap0 --dest-mac 08:00:27:9c:53:12 --seq [SEQ
RANGE]
```

If the packet lands in-window, the victim will respond with at most 2 challenge ACKs per second. These packets are still encrypted and originate from the virtual interface, unlike with Android, but we can still determine the contents of these packets by their size. The encrypted challenge ACK packets are larger than the encrypted RST packets. You can run tcpdump on the victim machine to accelerate the testing of his process by viewing the actual sequence and acknowledgement numbers.

After we have found an in-window sequence number, we locate an in-window acknowledgement by spoofing empty PSH-ACKs with the in-window sequence number and guessing the acknowledgement number by dividing the acknowledgement number space into eight blocks. In most instances, seven of these blocks will trigger challenge ACKs, but one of them will not, which allows us to quickly determine which block falls within the acknowledgement window. We are interested in the block that does not respond with a challenge ACK. This behavior can be observed by using an in-window sequence number and an acknowledgement number in the block containing the correct acknowledgement number.

```
nping --tcp --flags PA --source-ip 64.106.46.56 -g 80 --dest-ip
10.8.0.8 -p 40404 --rate 10 -c 10 -e ap0 --dest-mac 08:00:27:9c:53:12
-seq 12345678 --ack [ACK RANGE]
```

Finally, using the in-window sequence and acknowledgement numbers, we spoof empty PSH-ACKs using the same in-windows acknowledgement number and decrementing the sequence number until we trigger another challenge ACK. This sequence number is one fewer than the next expected sequence number. We can then arbitrarily inject data into the active TCP connection.

Continuing with our toy example:

```
nping --tcp --flags PA --source-ip 64.106.46.56 -g 80 --dest-ip
10.8.0.8 -p 40404 --rate 10 -c 10 -e ap0 --dest-mac 08:00:27:9c:53:12
-seq [EXACT] --ack [IN-WINDOW] --data-string “hello,world.”
```

**Operating Systems Affected:

Here is a list of the operating systems we have tested which are vulnerable to this attack:

Ubuntu 19.10 (systemd)
Fedora (systemd)
Debian 10.2 (systemd)
Arch 2019.05 (systemd)
Manjaro 18.1.1 (systemd)

Devuan (sysV init)
MX Linux 19 (Mepis+antiX)
Void Linux (runit)

Slackware 14.2 (rc.d)
Deepin (rc.d)
FreeBSD (rc.d)
OpenBSD (rc.d)

This list isn’t exhaustive, and we are continuing to test other distributions, but made usere to cover a variety of init systems to show this is not limited to systemd.

**Operating System Variations:

The behavior is slightly different on other operating systems. Here is a summary of the differences:

Android: In the first phase of the attack, Android responds with unencrypted RSTs to unsolicited SYN-ACKs for the correct port and ICMP packets for the incorrect one. For the second phase, it will respond with RSTs on the correct four-tuple.

MacOS/iOS: The first phase of the attack does not work as described here, but you can use an open port on the Apple machine to determine the virtual IP address. We use port 5223, which is used for iCloud, iMessage, FaceTime, Game Center, Photo Stream, and push notifications etc.

We know the phone will communicate with one of the push notification servers on port 5223, and have observed that on MacOS, the port used on the victim device is not the same as the port used to connect to the VPN server, but is very close (in our testing it has always been within 10).

```
nping --tcp --flags SA --source-ip 17.57.144.[84-87] -g 5223 --dest-ip
10.8.0.8 -p [X] --rate 3 -c 3 -e ap0 --dest-mac 08:00:27:9c:53:12
```

For iOS devices, it does not follow this convention for choosing the client’s source port, but always choose a port between ~48000-50000 (our testing on iOS 13.1 was between 48162-49555).

FreeBSD: The first two phases work essentially the same as Linux, however, for the last phase, the ACK number is not needed at all, so that piece of phase three can be skipped.

OpenBSD: OpenBSD responds to spoofed SYN packets to the correct virtual IP with unencrypted RST packets, and the incorrect virtual IP elicits unencrypted NTP packets or nothing at all for the first part of the attack. For the second part, the responses are encrypted, but we can still determine which packets are challenge ACKs from the packet size, as with Linux. Connections can be reset by sending a RST with the correct sequence number.

**Possible Mitigations:

1. Turning reverse path filtering on

Potential problem: Asynchronous routing not reliable on mobile devices, etc. Also, it isn’t clear that this is actually a solution since it appears to work in other OSes with different networking stacks. Also, even with reverse path filtering on strict mode, the first two parts of the attack can be completed, allowing the AP to make inferences about active connections, and we believe it may be possible to carry out the entire attack, but haven’t accomplished this yet.

2. Bogon filtering

Potential problem: Local network addresses used for vpns and local networks, and some nations, including Iran, use the reserved private IP

```
space as part of the public space.

3. Encrypted packet size and timing

Since the size and number of packets allows the attacker to bypass the
encryption provided by the VPN service, perhaps some sort of padding
could be added to the encrypted packets to make them the same size.
Also, since the challenge ACK per process limit allows us to determine
if the encrypted packets are challenge ACKs, allowing the host to
respond with equivalent-sized packets after exhausting this limit could
prevent the attacker from making this inference.


We have prepared a paper for publication concerning this
vulnerability and the related implications, but intend to keep it
embargoed until we have found a satisfactory workaround. Then we will
report the vulnerability to oss-security () lists openwall com. We are
also reporting this vulnerability to the other services affected, which
also includes: Systemd, Google, Apple, OpenVPN, and WireGuard, in
addition to distros () vs openwall org for the operating systems affected.

Thanks,

William J. Tolley
Beau Kujath
Jedidiah R. Crandall

Breakpointing Bad &
University of New Mexico
```

Current thread:

- **[CVE-2019-14899] Inferring and hijacking VPN-tunneled TCP connections.** *William J. Tolley (Dec 04)*
  - Re: [CVE-2019-14899] Inferring and hijacking VPN-tunneled TCP connections. *Noel Kuntze (Dec 05)*
    - Re: [CVE-2019-14899] Inferring and hijacking VPN-tunneled TCP connections. *Noel Kuntze (Dec 08)*