TCP Maintenance and Minor Extensions
Internet-Draft
Intended status: Experimental
Expires: 25 October 2026

M. Boucadair
Orange
T. Reddy
Nokia
J. Xing
Tencent

23 April 2026

TCP RST Diagnostic Payload
draft-ietf-tcpm-rst-diagnostic-payload-01

Abstract

This document specifies an experimental diagnostic payload format returned in TCP RST segments. Such payloads are used to share with an endpoint the reasons for which a TCP connection has been reset. Sharing this information is meant to ease diagnostic and troubleshooting.

This specification builds on provisions that are already present in RFC 9293 "Transmission Control Protocol (TCP)". As such, this document does not require any change to RFC 9293.

Discussion Venues

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the TCP Maintenance and Minor Extensions mailing list (tcpm@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/tcpm/.

Source for this draft and an issue tracker can be found at https://github.com/boucadair/draft-boucadair-tcpm-rst-diagnostic-payload.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Copyright Notice

Table of Contents

1. Introduction

A TCP connection [RFC9293] can be reset by a peer for various reasons, e.g., received data does not correspond to an active connection. Also, a TCP connection can be reset by an on-path service function (e.g., Carrier Grade NAT (CGN) [RFC6888], NAT64 [RFC6146], or firewall) for several reasons. Typically, a Network Address Translator (NAT) function can generate an RST segment to notify an endpoint upon the expiry of the lifetime of the corresponding mapping entry or because an RST segment was received from a peer (Section 2.2 of [RFC7857]).

A TCP connection can also be closed by a user or an application at any time. However, the peer that receives an RST segment does not have any hint about the reason that led to terminating the connection. Likewise, the application that relies upon such a TCP connection may not easily identify the reason for the connection reset. Troubleshooting such events at the remote side of the connection that receives the RST segment may not be trivial.

This document fills this void by specifying an experimental format of the diagnostic payload returned in an RST segment. This design is backward compatible with TCP as further clarified in Section 6.1.

The generic procedure for processing an RST segment is specified in Section 3.5.3 of [RFC9293]. Only the deviations from that procedure to insert and validate a diagnostic payload is provided in Section 4.

This document specifies the format and the overall approach to ease maintaining the list of codes while allowing for adding new codes as needed in the future and accommodating any existing vendor-specific codes. An initial version of error codes is available in Table 2. However, the authoritative source to retrieve the full list of error codes is the IANA-maintained registry (Section 9.1).

Section 5 provides a set of examples to illustrate the use of TCP RST diagnostic payloads.

Section 7 provides an informative discussion of socket API considerations. Implementation and experimental validation are detailed in Appendix A.

Experiment goals are listed in Section 3.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This document makes use of the terms defined in Section 4 of [RFC9293].

SEG.LEN is defined in Section 3.3.1 of [RFC9293].

This document uses the following terms:

RST diagnostic payload: The payload of an RST message that conveys diagnostic data.

RST with diagnostic payload: An RST segment that includes diagnostic payload.

3. Experiment Description & Goals

The main objective of this experiment is to have a common format of RST diagnostic payload that would be used as basis for consistent testing and evaluation in a variety of deployment contexts (Internet, data centers, application clients/servers provided and managed by the same entity, clients and servers software owned by distinct entities, etc.).

Experiments reports are encouraged to share the main lessons learned in these experimentations. Specifically, the following items are of interest:

Delivery & on-path device interference: Identify and share issues (or lack thereof) related to the delivery of RST with diagnostic payload.

CPU/load impact: Assess CPU/load impact (or lack thereof) of handling RSTs, including when a mix of RSTs with and without diagnostic payload are sent by an endpoint.

Standard reset reasons: Assess whether the list of code reasons (Section 9.1) reflects observed reset cases.

Integration of socket API extensions: Exercise the socket API extensions and identify any required adjustement (Section 7).

Operational guidance: Strengthen the operational guidance for deploying RSTs with diagnostic payload.

4. RST Diagnostic Payload

4.1. Payload Format

This section defines the message format to convey RST diagnostic payload. This format is designed to minimize the length of the payload.

The format of the RST diagnostic payload is shown in Figure 1.

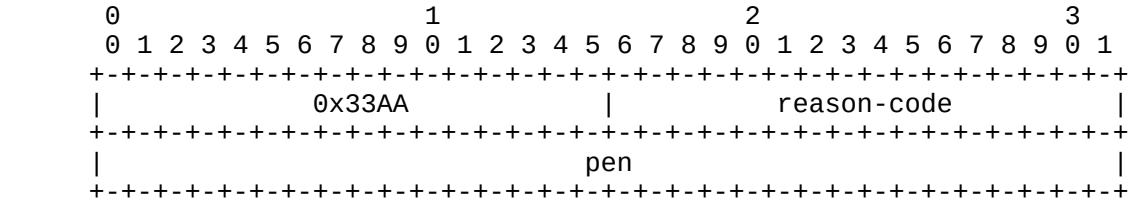

Figure 1: Structure of the RST Diagnostic Payload

The RST diagnostic payload comprises a magic number that is used to unambiguously identify an RST payload that follows this specification. It MUST be set to 0x33AA.

The descriptions of other fields shown in Figure 1 are as follows:

reason-code: This field takes a value from an available registry (IANA or vendor-specific).

Value 0 is reserved and MUST NOT be used.

The reason code is taken from the "TCP Failure Causes" registry (Section 9.1) if "pen" is set to 0.

It is RECOMMENDED that implementations support all codes defined in Table 2.

If the "pen" is not set to 0, then the reason code refers to the registry of the entity specified by the "pen" parameter.

pen: Includes a Private Enterprise Number (PEN) [Private-Enterprise-Numbers].

The reserved PEN value "0" is used to indicate that the reason code refers to the IANA-maintained registry (Section 9.1).

SEG.LEN MUST be 8 for an RST with diagnostic payload.

4.2. Behavior

Malformed RST diagnostic payloads that include the magic numbers 0x33AA MUST be silently ignored by the receiver. RSTs that carry such malformed diagnostic payloads are handled like an RST without payload.

A peer that receives a valid diagnostic payload may pass the reset reason information to the local application in addition to the information (MUST-12) described in Section 3.6 of [RFC9293]. That information may also be logged locally, unless a local policy specifies otherwise. How the information is passed to an application and how it is stored locally is implementation-specific.

Because new codes may be supported by a sender, receivers SHOULD NOT discard received RST diagnostic payloads with an unknown reason code unless configured otherwise.

5. Some Examples

Figure 2 depicts an example of an RST diagnostic payload that is generated to inform the peer that the TCP connection is reset because an ACK was received from that peer while the connection is still in the LISTEN state (Section 3.10.7.2 of [RFC9293]).

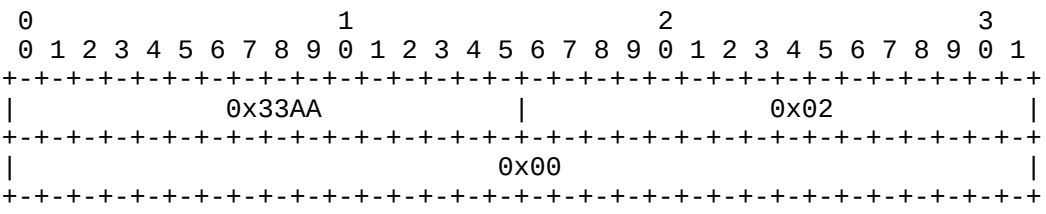

Figure 2: Example of an RST Diagnostic Payload with Reason Code

An RST diagnostic payload may also be sent by an on-path service function. For example, the following diagnostic payload is returned by a NAT function upon expiry of the mapping entry to which the TCP connection is bound (Figure 3).

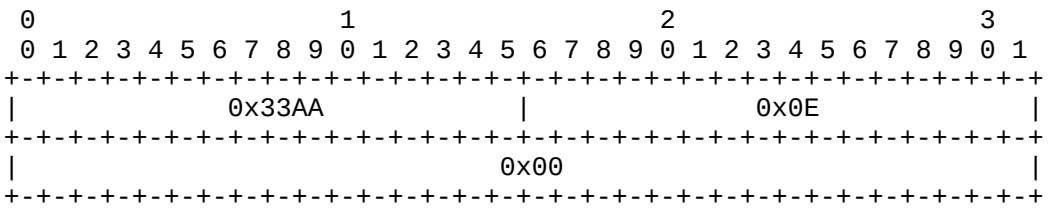

Figure 3: Example of an RST Diagnostic Payload to Report Connection Timeout

Figure 4 illustrates an RST diagnostic payload that is returned by a peer that resets a TCP connection for a reason code 1234 (0x4D2) defined by a vendor with the private enterprise number 32473 (0x7ED9).

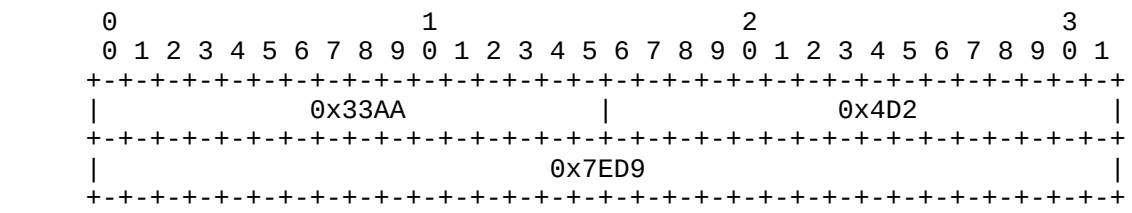

Figure 4: Example of an RST Diagnostic Payload to Report Vendor-Specific Reason Code

Figure 4 uses the Enterprise Number 32473 defined for documentation use [RFC5612].

6. Operational Considerations

6.1. Backward Compatibility

Returning diagnostic data in an RST segment is consistent with the provision in Section 3.5.3 of [RFC9293] for RST segments, especially:

> "TCP implementations SHOULD allow a received RST segment to include data (SHLD-2)."

Also, this document does not change the conditions under which an RST segment is generated (Section 3.5.2 of [RFC9293]).

6.2. Multiple RSTs

Per Section 3.6 of [RFC9293], one or more RST segments can be sent to reset a connection.

Sending more RST segments to reset a connection can be used to mitigate deployment contexts where some on-path devices may discard RSTs with payload data.

Whether a TCP endpoint elects to send more than one RST with only a subset of them that include the diagnostic payload is implementation-specific.

6.3. Manageability

TCP server implementations should support the following parameters:

* A parameter to control the activation of RSTs with diagnostic payload.

* A parameter to expose the set of reason codes that are supported by an implementation.

* A parameter to control whether "empty" RSTs are also sent together with an RST with diagnostic payload.

* A rate-limit of RSTs with diagnostic payload.

* Counters to track sent/received RSTs with diagnostic payload.

* Counters to track received invalid RSTs with diagnostic payload.

6.4. Maintenance

As new reason codes may be added to the IANA registry, there is a risk that the codes that are supported by an implementation do not match the latest version in the registry. Deviations can be detected using the exposure parameter in Section 6.3.

It is RECOMMENDED to proceed with regular software updates to align with the latest version in the registry.

7. Socket API Considerations (Informative)

This section describes how the socket API can be extended to provide a way for an application to use the functionality described in this document.

This section is informational only.

The API described in this section can change in a non-backwards compatible way during the evolution of this document due to changed functionality or gained experience during the implementation.

7.1. Socket Options

Table 1 provides an overview of the IPPROTO_TCP-level socket options defined in this section.

Option Name	Data Type	Set	Get
TCP_RST_REASON_ENABLE	uint32_t	X	
TCP_RST_REASON_CODE	struct tcp_rst_reason	X	X

Table 1: Socket Options

7.1.1. Enable the Sending of the Diagnostic Payload (TCP_RST_REASON_ENABLE)

Using setsockopt() with the IPPROTO_TCP-level socket option with the name TCP_RST_REASON_ENABLE enables or disables the sending of the diagnostic payload using a reason-code and pen. The option_value of type uint32_t specifies the pen in host byte order to use. When 0 is used (Section 3 of [RFC9371]), the reason-codes from the registry specified in Section 9.1 are used. When 0xffffffff is used (Section 3 of [RFC9371]), the sending is disabled. The default is that the sending of a diagnostic payload is disabled. An implementation might not support the use of PENs different from zero and 0xffffffff.

7.1.2. Get or Set the Diagnostic Payload as Code (TCP_RST_REASON_CODE)

Using getsockopt() with the IPPROTO_TCP-level socket option with the name TCP_RST_REASON_CODE allows the caller to retrieve the reason-code and the pen of the diagnostic payload in the received RST segment, which terminated the corresponding TCP connection.

Using setsockopt() with this socket option allows the caller to provide reason-code and pen to be sent as part of the diagnostic payload when the application triggers the sending of a RST segment by using close(). In addition to using close() in combination with the SOL_SOCKET-level socket option with name SO_LINGER, the application can just provide the TCP_RR_RST_ON_CLOSE flag in trr_flags. This way the application can trigger the sending of a RST segment by calling setsockopt() once followed by close().

For accepted sockets, this socket option is inherited from the listening socket.

The following structure is used as the option_value:

```
struct tcp_rst_reason {
        uint16_t trr_flags;
        uint16_t trr_code;
        uint32_t trr_pen;
};
```

trr_flags: This field is reported as 0 for getsockopt() calls. For setsockopt()

calls, the following flag can be used:

TCP_RR_RST_ON_CLOSE: When this flag is set, calling close() triggers the sending of a RST segment similar to case, where the SOL_SOCKET-level socket option with name SO_LINGER is used to enable lingering with the linger time of 0. When this flag is cleared, the corresponding functionality is disabled.

trr_code: The reason-code in host byte order to be interpreted in combination with the PEN provided in trr_pen. In case of trr_pen being zero, trr_code refers to a value in the registry defined in Section 9.1.

trr_pen: The PEN in host byte order to is used in combination with the reason-code specified in trr_code. When this socket option is used with setsockopt(), it is an error to use zero as a value for trr_pen as long as trr_code is not zero.

When getsockopt() with this socket option is performed on a socket, which has not received a RST with a diagnostic payload containing a reason-code and pen, zero is provided as the trr_code and trr_pen. When setsockopt() with a trr_code and trr_pen of zero is performed, the special handling of RST segments sent during the ungraceful termination of the TCP connection is disabled.

8. Security Considerations

[RFC9293] discusses TCP-related security considerations. In particular, RST-specific attacks and their mitigations are discussed in Section 3.10.7.3 of [RFC9293].

The presence of vendor-specific reason codes may be used to fingerprint hosts. Such a concern does not apply if the reason codes are taken from the IANA-maintained registry. Implementers are, thus, encouraged to register new codes within IANA instead of maintaining specific registries.

9. IANA Considerations

9.1. New Registry for TCP Failure Causes

This document requests IANA to create a new registry entitled "TCP Failure Causes" under the "Transmission Control Protocol (TCP) Parameters" registry group [IANA-TCP].

Values are taken from the 1-65535 range.

The assignment policy for this registry is "Expert Review" (Section 4.5 of [RFC8126]). See more guidance at Section 9.2.

The registry is initially populated with the values listed in Table 2.

Value	Description	Specification (if available)
0	Reserved	ThisDocument
1	Illegal option length	Section 3.1 of [RFC9293]
2	Desynchronized state	Section 3.5.1 of [RFC9293]
3	New data is received after CLOSE is called	Sections 3.6.1 and 3.10.7.1 of [RFC9293]
4	ABORT process	Section 3.10.5 of [RFC9293]
5	Unexpected ACK received by non-synchronized state connection	Section 3.10.7 of [RFC9293]
6	Unexpected SYN in the window	Section 3.10.7 of [RFC9293]
7	Unexpected security compartment	Appendix A.1 of [RFC9293]
8	Malformed message	ThisDocument
9	Not authorized	ThisDocument
10	Resource exceeded	ThisDocument

11	Network failure	ThisDocument
12	Reset received from the peer	ThisDocument
13	Destination unreachable	ThisDocument
14	Connection timeout	ThisDocument
15	Too much outstanding data	Section 3.6 of [RFC8684]
16	Unacceptable performance	Section 3.6 of [RFC8684]
17	Middlebox interference	Section 3.6 of [RFC8684]

Table 2: Initial TCP Failure Causes

Note that codes in the 8-14 range can be used by service functions (CGN, firewall, proxy, etc.).

Note to the RFC Editor: Please replace ThisDocument with the RFC number assigned to this document.

9.2. Guidelines for the Designated Experts

Criteria that should be applied by the designated experts include determining whether the proposed registration duplicates existing entries and whether the registration description is clear and fits the purpose of this registry.

The designated experts may approve registration once they checked that the new requested code is not covered by an existing code and if the provided reasoning to register the new code is acceptable. A registration request may supply a pointer to a specification where that code is defined. However, a registration may be accepted even if no permanent and readily available public specification is available.

Registration requests are to be sent to rst-diag-review@ietf.org (mailto:rst-diag-review@ietf.org) and are evaluated within a three-week review period on the advice of one or more designated experts. Within the review period, the designated experts will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/rfc/rfc8126>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8684] Ford, A., Raiciu, C., Handley, M., Bonaventure, O., and C. Paasch, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 8684, DOI 10.17487/RFC8684, March 2020, <https://www.rfc-editor.org/rfc/rfc8684>.

[RFC9293] Eddy, W., Ed., "Transmission Control Protocol (TCP)", STD 7, RFC 9293, DOI 10.17487/RFC9293, August 2022, <https://www.rfc-editor.org/rfc/rfc9293>.

10.2. Informative References

[IANA-TCP] "Transmission Control Protocol (TCP) Parameters", <https://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml#>.

[Private-Enterprise-Numbers] "Private Enterprise Numbers", <https://www.iana.org/assignments/enterprise-numbers>.

[RFC5612] Eronen, P. and D. Harrington, "Enterprise Number for Documentation Use", RFC 5612, DOI 10.17487/RFC5612, August 2009, <https://www.rfc-editor.org/rfc/rfc5612>.

[RFC6146] Bagnulo, M., Matthews, P., and I. van Beijnum, "Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers", RFC 6146, DOI 10.17487/RFC6146, April 2011, <https://www.rfc-editor.org/rfc/rfc6146>.

[RFC6888] Perreault, S., Ed., Yamagata, I., Miyakawa, S., Nakagawa, A., and H. Ashida, "Common Requirements for Carrier-Grade NATs (CGNs)", BCP 127, RFC 6888, DOI 10.17487/RFC6888, April 2013, <https://www.rfc-editor.org/rfc/rfc6888>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <https://www.rfc-editor.org/rfc/rfc7252>.

[RFC7857] Penno, R., Perreault, S., Boucadair, M., Ed., Sivakumar, S., and K. Naito, "Updates to Network Address Translation (NAT) Behavioral Requirements", BCP 127, RFC 7857, DOI 10.17487/RFC7857, April 2016, <https://www.rfc-editor.org/rfc/rfc7857>.

[RFC9371] Baber, A. and P. Hoffman, "Registration Procedures for Private Enterprise Numbers (PENs)", RFC 9371, DOI 10.17487/RFC9371, March 2023, <https://www.rfc-editor.org/rfc/rfc9371>.

Appendix A. Implementation and Experimental Validation in Linux

Questions and concerns have been raised regarding whether RST with payload affects the normal termination of flows across different software platforms, operating systems, middleboxes, etc. Even though Section 3.5.3 of [RFC9293] explicitly allows this behavior, a full implementation is needed to widely verify if unexpected cases can happen in the real world.

The overall design in Linux is to pre-allocate a large enough zeroed buffer, put a reset reason code in the first byte and sent it out to verify whether the RST with payload can be possibly declined by any equipment in between two sides and the other side successfully parses the RST with payload.

A.1. Implementation

The following implementation is accomplished on top of Linux 6.16:

Payload Attachment: Allocate a 1000-byte data payload attached to all generated RST packets.

Reason Code Encoding: The first byte of the payload is used to store a predefined reset reason code that is listed in include/net/rstreason.h file, while the remainder of the payload is zero-padded. The reason code is generated by the existing mechanism called TCP reset reasons (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d5115a55ffb52).

Handling of Reset Types: The implementation distinguishes between the two primary reset scenarios in tcp_send_active_reset() and tcp_v4_send_reset() respectively:

* For an *Active Reset*, initiated proactively by the local system, the payload is placed in the linear area of the socket buffer (sk_buff).

* For a *Passive Reset*, sent in response to an unexpected or invalid incoming packet, the payload is stored in the non-linear (paged) area of the sk_buff.

Complete patch is shown in Figure 5.

```
diff --git a/include/net/tcp.h b/include/net/tcp.h
index b3815d104340..0b32257774c8 100644
--- a/include/net/tcp.h
+++ b/include/net/tcp.h
@@ -62,6 +62,7 @@ void tcp_time_wait(struct sock *sk, int state, int timeo);
 #define MAX_TCP_OPTION_SPACE 40
 #define TCP_MIN_SND_MSS                48
 #define TCP_MIN_GSO_SIZE       (TCP_MIN_SND_MSS - MAX_TCP_OPTION_SPACE)
+#define PAYLOAD_LEN 1000

 /*
  * Never offer a window over 32767 without using window scaling. Some
diff --git a/net/ipv4/tcp_ipv4.c b/net/ipv4/tcp_ipv4.c
index 84d3d556ed80..49250e6bd6a1 100644
--- a/net/ipv4/tcp_ipv4.c
+++ b/net/ipv4/tcp_ipv4.c
@@ -741,6 +741,7 @@ static bool tcp_v4_ao_sign_reset(const struct sock *sk, struct sk_buf
f *skb,
 static void tcp_v4_send_reset(const struct sock *sk, struct sk_buff *skb,
                               enum sk_rst_reason reason)
 {
+       u32 len = sizeof(struct tcphdr) + REPLY_OPTIONS_LEN + PAYLOAD_LEN;
        const struct tcphdr *th = tcp_hdr(skb);
        struct {
                struct tcphdr th;
@@ -757,6 +758,7 @@ static void tcp_v4_send_reset(const struct sock *sk, struct sk_buff *
skb,
 #endif
```

```
        u64 transmit_time = 0;
        struct sock *ctl_sk;
+       char buffer[len];
        struct net *net;
        u32 txhash = 0;

@@ -786,7 +788,8 @@ static void tcp_v4_send_reset(const struct sock *sk, struct sk_buff *
skb,
        }

        memset(&arg, 0, sizeof(arg));
-       arg.iov[0].iov_base = (unsigned char *)&rep;
+       memset(&buffer, 0, len);
+       arg.iov[0].iov_base = (unsigned char *)buffer;
        arg.iov[0].iov_len  = sizeof(rep.th);

        net = sk ? sock_net(sk) : skb_dst_dev_net_rcu(skb);
@@ -911,6 +914,10 @@ static void tcp_v4_send_reset(const struct sock *sk, struct sk_buff
*skb,
                ctl_sk->sk_mark = 0;
                ctl_sk->sk_priority = 0;
        }
+       memcpy(buffer, (char *)&rep, arg.iov[0].iov_len);
+       /* put rst reason into the first byte in payload */
+       buffer[arg.iov[0].iov_len] = reason;
+       arg.iov[0].iov_len += PAYLOAD_LEN;
        ip_send_unicast_reply(ctl_sk, sk,
                              skb, &TCP_SKB_CB(skb)->header.h4.opt,
                              ip_hdr(skb)->saddr, ip_hdr(skb)->daddr,
diff --git a/net/ipv4/tcp_output.c b/net/ipv4/tcp_output.c
index b616776e3354..c07dd009a0de 100644
--- a/net/ipv4/tcp_output.c
+++ b/net/ipv4/tcp_output.c
@@ -3628,12 +3628,14 @@ void tcp_send_fin(struct sock *sk)
 void tcp_send_active_reset(struct sock *sk, gfp_t priority,
                            enum sk_rst_reason reason)
 {
+       u32 len = MAX_TCP_HEADER + PAYLOAD_LEN;
+       char payload[PAYLOAD_LEN];
        struct sk_buff *skb;

        TCP_INC_STATS(sock_net(sk), TCP_MIB_OUTRSTS);

        /* NOTE: No TCP options attached and we never retransmit this. */
-       skb = alloc_skb(MAX_TCP_HEADER, priority);
+       skb = alloc_skb(len, priority);
        if (!skb) {
                NET_INC_STATS(sock_net(sk), LINUX_MIB_TCPABORTFAILED);
                return;
@@ -3641,8 +3643,13 @@ void tcp_send_active_reset(struct sock *sk, gfp_t priority,
```

```
        /* Reserve space for headers and prepare control bits. */
        skb_reserve(skb, MAX_TCP_HEADER);
+       skb_put(skb, PAYLOAD_LEN);
        tcp_init_nondata_skb(skb, tcp_acceptable_seq(sk),
                             TCPHDR_ACK | TCPHDR_RST);
+       memset(payload, 0, PAYLOAD_LEN);
+       payload[0] = reason;
+       skb_store_bits(skb, 0, payload, PAYLOAD_LEN);
+       TCP_SKB_CB(skb)->end_seq += PAYLOAD_LEN;
        tcp_mstamp_refresh(tcp_sk(sk));
        /* Send it off. */
        if (tcp_transmit_skb(sk, skb, 0, priority))
```

Figure 5: Complete Patch

A.2. Experimental Validation

To ensure a thorough evaluation, a multi-layered experimental methodology was designed, progressing from basic functional checks to complex, real-world compatibility and stability tests. The whole implementation has been deployed in Tencent’s production environment for almost six months.

A.2.1. Functional Verification

The basic functionality test is using iperf or iperf3 to construct a normal termination senario. The tcpdump tool with -X option effectively helps to show the [RST+] flag and the 1000-byte payload, confirming that the kernel correctly generated and transmitted the augmented RST packets.

Two servers, designated as Client A and Server B. The test is conducted as following:

1. Start the iperf3 server on Server B (iperf3 -s).

2. Initiate a connection from Client A to Server B (iperf3 -c [IP_of_B]).

3. After the connection is established, one of the iperf3 processes is terminated using the kill command, triggering the kernel to send an RST packet.

4. Simultaneously, tcpdump is run on either host to capture the reset packet using the filter: ’tcp[tcpflags] & tcp-rst != 0’ -X -nn -vv -S.

A.2.2. Compatibility Verification

Hardwares and Kernels: Tests were conducted on various Linux distributions (e.g., Ubuntu or CentOS) with different kernel versions. The physical hosts were equipped with a range of network interface cards (NICs), including Intel i40e, ixgbe, and Mellanox mlx5.

Virtualization: The mechanism was tested in a virtualized environment where the VM used a virtio_net driver and the host employed DPDK to redirect packets in the host.

Middleboxes: Tests were performed with Layer 4 (L4) and Layer 7 (L7) gateways placed between the client and server to verify correct packet parsing and forwarding.

Wide Area Network (WAN): The setup was tested over long-haul international links to simulate complex conditions, including China-to-Singapore (RTT > 30ms) and China-to-Germany (RTT > 200ms).

In conclusion, across all complex environment tests, the RST packets with payloads were successfully received by the peer. No instances of packets being dropped or mishandled by intermediate middleboxes, gateways, or diverse hardware and software configurations were observed.

Acknowledgments

The "diagnostic payload" name is inspired by Section 5.5.2 of [RFC7252] that was cited by Carsten Bormann in the tcpm mailing list.

Thanks to Jon Shallow, Neal Cardwell, Lars Eggert, Eric Dumazet, Rick Jones, Yoshifumi Nishida, Li Jinghui, and Gleb Smirnoff for the review and comments.

Contributors

Michael T端xen
Email: tuexen@fh-muenster.de

Authors' Addresses

Mohamed Boucadair
Orange
Email: mohamed.boucadair@orange.com

Tirumaleswar Reddy
Nokia
India
Email: kondtir@gmail.com

Jason Xing
Tencent
Email: kerneljasonxing@gmail.com