Internet Area WG
Internet-Draft
Intended status: Standard track
Expires October 28, 2017

T. Herbert
Quantonium
L. Yong
Huawei USA
O. Zia
Microsoft

April 26, 2017

Generic UDP Encapsulation
draft-ietf-intarea-gue-02

Abstract

This specification describes Generic UDP Encapsulation (GUE), which is a scheme for using UDP to encapsulate packets of different IP protocols for transport across layer 3 networks. By encapsulating packets in UDP, specialized capabilities in networking hardware for efficient handling of UDP packets can be leveraged. GUE specifies basic encapsulation methods upon which higher level constructs, such as tunnels and overlay networks for network virtualization, can be constructed. GUE is extensible by allowing optional data fields as part of the encapsulation, and is generic in that it can encapsulate packets of various IP protocols.

Table of Contents

1. Introduction

This specification describes Generic UDP Encapsulation (GUE) which is a general method for encapsulating packets of arbitrary IP protocols within User Datagram Protocol (UDP) [RFC0768] packets. Encapsulating packets in UDP facilitates efficient transport across networks. Networking devices widely provide protocol specific processing and optimizations for UDP (as well as TCP) packets. Packets for atypical IP protocols (those not usually parsed by networking hardware) can be encapsulated in UDP packets to maximize deliverability and to leverage flow specific mechanisms for routing and packet steering.

GUE provides an extensible header format for including optional data in the encapsulation header. This data potentially covers items such as the virtual networking identifier, security data for validating or authenticating the GUE header, congestion control data, etc. GUE also allows private optional data in the encapsulation header. This feature can be used by a site or implementation to define local custom optional data, and allows experimentation of options that may eventually become standard.

This document does not define any specific GUE extensions. [GUEEXTENS] specifies a set of core extensions and [GUE4NVO3] defines an extension for using GUE with network virtualization.

The motivation for the GUE protocol is described in section 6.

1.1. Terminology and acronyms

GUE	Generic UDP Encapsulation
GUE Header	A variable length protocol header that is composed of a primary four byte header and zero or more four byte words for optional header data
GUE packet	A UDP/IP packet that contains a GUE header and GUE payload within the UDP payload
Encapsulator	A network node that encapsulates a packet in GUE
Decapsulator	A network node that decapsulates and processes packets encapsulated in GUE
Data message	An encapsulated packet in the GUE payload that is addressed to the protocol stack for an associated protocol
Control message	A formatted message in the GUE payload that is

	implicitly addressed to the decapsulator to monitor or control the state or behavior of a tunnel
Flags	A set of bit flags in the primary GUE header
Extension field	An optional field in a GUE header whose presence is indicated by corresponding flag(s)
C-bit	A single bit flag in the primary GUE header that indicates whether the GUE packet contains a control message or data message
Hlen	A field in the primary GUE header that gives the length of the GUE header
Proto/ctype	A field in the GUE header that holds either the IP protocol number for a data message or a type for a control message
Private data	Optional data in the GUE header that can be used for private purposes
Outer IP header	Refers to the outer most IP header or packet when encapsulating a packet over IP
Inner IP header	Refers to an encapsulated IP header when an IP packet is encapsulated
Outer packet	Refers to an encapsulating packet
Inner packet	Refers to a packet that is encapsulated

1.2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Base packet format

A GUE packet is comprised of a UDP packet whose payload is a GUE header followed by a payload which is either an encapsulated packet of some IP protocol or a control message such as an OAM (Operations, Administration, and Management) message. A GUE packet has the general format:

```
+-------------------------------+
|                               |
|        UDP/IP header          |
|                               |
|-------------------------------|
|                               |
|         GUE Header            |
|                               |
|-------------------------------|
|                               |
|      Encapsulated packet      |
|      or control message       |
|                               |
+-------------------------------+
```

The GUE header is variable length as determined by the presence of optional extension fields.

2.1. GUE version

The first two bits of the GUE header contain the GUE protocol version number. The rest of the fields after the GUE version number are defined based on the version number. Versions 0 and 1 are described in this specification; versions 2 and 3 are reserved.

3. Version 0

Version 0 of GUE defines a generic extensible format to encapsulate packets by Internet protocol number.

3.1. Header format

The header format for version 0 of GUE in UDP is:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+\
|        Source port            |      Destination port         | |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ UDP
|           Length              |          Checksum             | |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+/
| 0 |C|   Hlen  |  Proto/ctype  |             Flags             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                  Extensions Fields (optional)                 ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                    Private data (optional)                    ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The contents of the UDP header are:

- o Source port: If connection semantics (section 5.6.1) are applied to an encapsulation, this is set to the local source port for the connection. When connection semantics are not applied, this is set to a flow entropy value for use with ECMP (Equal-Cost Mulit-Path [RFC2992]); the properties of flow entropy are described in section 5.11.

- o Destination port: If connection semantics (section 5.6.1) are applied to an encapsulation, this is set to the destination port for the tuple. If connection semantics are not applied this is set to the GUE assigned port number, 6080.

- o Length: Canonical length of the UDP packet (length of UDP header and payload).

- o Checksum: Standard UDP checksum (handling is described in section 5.7).

The GUE header consists of:

- o Ver: GUE protocol version (0).

- o C: C-bit: When set indicates a control message, not set indicates a data message.

- o Hlen: Length in 32-bit words of the GUE header, including optional extension fields but not the first four bytes of the header. Computed as (header_len - 4) / 4 where header_len is the total header length in bytes. All GUE headers are a multiple of four bytes in length. Maximum header length is 128 bytes.

- o Proto/ctype: When the C-bit is set, this field contains a control message type for the payload (section 3.2.2). When C-bit is not set, the field holds the Internet protocol number for the encapsulated packet in the payload (section 3.2.1). The control message or encapsulated packet begins at the offset provided by Hlen.

- o Flags: Header flags that may be allocated for various purposes and may indicate presence of extension fields. Undefined header flag bits MUST be set to zero on transmission.

- o Extension Fields: Optional fields whose presence is indicated by corresponding flags.

- o Private data: Optional private data block (see section 3.4). If the private block is present, it immediately follows that last extension field present in the header. The private block is considered to be part of the GUE header. The length of this data is determined by subtracting the starting offset from the header length.

3.2. Proto/ctype field

The proto/ctype fields either contains an Internet protocol number (when the C-bit is not set) or GUE control message type (when the C-bit is set).

3.2.1 Proto field

When the C-bit is not set, the proto/ctype field MUST contain an IANA Internet Protocol Number. The protocol number is interpreted relative to the IP protocol that encapsulates the UDP packet (i.e. protocol of the outer IP header). The protocol number serves as an indication of the type of the next protocol header which is contained in the GUE payload at the offset indicated in Hlen. Intermediate devices MAY parse the GUE payload per the number in the proto/ctype field, and header flags cannot affect the interpretation of the proto/ctype field.

When the outer IP protocol is IPv4, the proto field MUST be set to a valid IP protocol number usable with IPv4; it MUST NOT be set to a number for IPv6 extension headers or ICMPv6 options (number 58). An

exception is that the destination options extension header using the PadN option MAY be used with IPv4 as described in section 3.6. The "no next header" protocol number (59) also MAY be used with IPv4 as described below.

When the outer IP protocol is IPv6, the proto field can be set to any defined protocol number except that it MUST NOT be set to Hop-by-hop options (number 0). If a received GUE packet in IPv6 contains a protocol number that is an extension header (e.g. Destination Options) then the extension header is processed after the GUE header is processed as though the GUE header is an extension header.

IP protocol number 59 ("No next header") can be set to indicate that the GUE payload does not begin with the header of an IP protocol. This would be the case, for instance, if the GUE payload were a fragment when performing GUE level fragmentation. The interpretation of the payload is performed through other means (such as flags and extension fields), and intermediate devices MUST NOT parse packets based on the IP protocol number in this case.

3.2.2 Ctype field

When the C-bit is set, the proto/ctype field MUST be set to a valid control message type. A value of zero indicates that the GUE payload requires further interpretation to deduce the control type. This might be the case when the payload is a fragment of a control message, where only the reassembled packet can be interpreted as a control message.

Control messages will be defined in an IANA registry. Control message types 1 through 127 may be defined in standards. Types 128 through 255 are reserved to be user defined for experimentation or private control messages.

This document does not specify any standard control message types other than type 0.

3.3. Flags and extension fields

Flags and associated extension fields are the primary mechanism of extensibility in GUE. As mentioned in section 3.1, GUE header flags indicate the presence of optional extension fields in the GUE header. [GUEXTENS] defines a basic set of GUE extensions.

3.3.1. Requirements

There are sixteen flag bits in the GUE header. Some flags indicate presence of an extension fields. The size of an extension field

indicated by a flag MUST be fixed.

Flags can be paired together to allow different lengths for an extension field. For example, if two flag bits are paired, a field can possibly be three different lengths-- that is bit value of 00 indicates no field present; 01, 10, and 11 indicate three possible lengths for the field. Regardless of how flag bits are paired, the lengths and offsets of optional fields corresponding to a set of flags MUST be well defined.

Extension fields are placed in order of the flags. New flags are to be allocated from high to low order bit contiguously without holes. Flags allow random access, for instance to inspect the field corresponding to the Nth flag bit, an implementation only considers the previous N-1 flags to determine the offset. Flags after the Nth flag are not pertinent in calculating the offset of the Nth flag. Random access of flags and fields permits processing of optional extensions in an order that is independent of their position in the packet. The processing order of extensions defined in [GUEEXTENS] demonstrates this property.

Flags (or paired flags) are idempotent such that new flags MUST NOT cause reinterpretation of old flags. Also, new flags MUST NOT alter interpretation of other elements in the GUE header nor how the message is parsed (for instance, in a data message the proto/ctype field always holds an IP protocol number as an invariant).

The set of available flags can be extended in the future by defining a "flag extensions bit" that refers to a field containing a new set of flags.

3.3.2. Example GUE header with extension fields

An example GUE header for a data message encapsulating an IPv4 packet and containing the VNID and Security extension fields (both defined in [GUEXTENS]) is shown below:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 0 |0|    3    |      94       |1|0 0 1|          0            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             VNID                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                           Security                            +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

In the above example, the first flag bit is set which indicates that the VNID extension is present which is a 32 bit field. The second through fourth bits of the flags are paired flags that indicate the presence of a security field with seven possible sizes. In this example 001 indicates a sixty-four bit security field.

3.4. Private data

An implementation MAY use private data for its own use. The private data immediately follows the last field in the GUE header and is not a fixed length. This data is considered part of the GUE header and MUST be accounted for in header length (Hlen). The length of the private data MUST be a multiple of four and is determined by subtracting the offset of private data in the GUE header from the header length. Specifically:

```
Private_length = (Hlen * 4) - Length(flags)
```

where "Length(flags)" returns the sum of lengths of all the extension fields present in the GUE header. When there is no private data present, the length of the private data is zero.

The semantics and interpretation of private data are implementation specific. The private data may be structured as necessary, for instance it might contain its own set of flags and extension fields.

An encapsulator and decapsulator MUST agree on the meaning of private data before using it. The mechanism to achieve this agreement is outside the scope of this document but could include implementation-defined behavior, coordinated configuration, in-band communication using GUE control messages, or out-of-band messages.

If a decapsulator receives a GUE packet with private data, it MUST validate the private data appropriately. If a decapsulator does not expect private data from an encapsulator, the packet MUST be dropped. If a decapsulator cannot validate the contents of private data per the provided semantics, the packet MUST also be dropped. An implementation MAY place security data in GUE private data which if present MUST be verified for packet acceptance.

3.5. Message types

3.5.1. Control messages

Control messages carry formatted data that are implicitly addressed to the decapsulator to monitor or control the state or behavior of a tunnel (OAM). For instance, an echo request and corresponding echo reply message can be defined to test for liveness.

Control messages are indicated in the GUE header when the C-bit is set. The payload is interpreted as a control message with type specified in the proto/ctype field. The format and contents of the control message are indicated by the type and can be variable length.

Other than interpreting the proto/ctype field as a control message type, the meaning and semantics of the rest of the elements in the GUE header are the same as that of data messages. Forwarding and routing of control messages should be the same as that of a data message with the same outer IP and UDP header and GUE flags; this ensures that control messages can be created that follow the same path as data messages.

3.5.2. Data messages

Data messages carry encapsulated packets that are addressed to the protocol stack for the associated protocol. Data messages are a primary means of encapsulation and can be used to create tunnels for overlay networks.

Data messages are indicated in GUE header when the C-bit is not set. The payload of a data message is interpreted as an encapsulated packet of an Internet protocol indicated in the proto/ctype field. The packet immediately follows the GUE header.

3.6. Hiding the transport layer protocol number

The GUE header indicates the Internet protocol of the encapsulated packet. A protocol number is either contained in the Proto/ctype field of the primary GUE header or in the Payload Type field of a GUE Transform extension field (used to encrypt the payload with DTLS, [GUEEXTENS]). If the transport protocol number needs to be hidden from the network, then a trivial destination options can be used.

The PadN destination option [RFC2460] can be used to encode the transport protocol as a next header of an extension header (and maintain alignment of encapsulated transport headers). The Proto/ctype field or Payload Type field of the GUE Transform field is set to 60 to indicate that the first encapsulated header is a destination options extension header.

The format of the extension header is below:

```
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   | Next Header |    2      |     1     |      0    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

For IPv4, it is permitted in GUE to used this precise destination

option to contain the obfuscated protocol number. In this case next header MUST refer to a valid IP protocol for IPv4. No other extension headers or destination options are permitted with IPv4.

4. Version 1

Version 1 of GUE allows direct encapsulation of IPv4 and IPv6 in UDP. In this version there is no GUE header; a UDP packet carries an IP packet. The first two bits of the UDP payload for GUE are the GUE version and coincide with the first two bits of the version number in the IP header. The first two version bits of IPv4 and IPv6 are 01, so we use GUE version 1 for direct IP encapsulation which makes two bits of GUE version to also be 01.

This technique is effectively a means to compress out the GUE header when encapsulating IPv4 or IPv6 packets and there are no flags or extension fields present. This method is compatible to use on the same port number as packets with the GUE header (GUE version 0 packets). This technique saves encapsulation overhead on costly links for the common use of IP encapsulation, and also obviates the need to allocate a separate port number for IP-over-UDP encapsulation.

4.1. Direct encapsulation of IPv4

The format for encapsulating IPv4 directly in UDP is:

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+\
   |        Source port            |      Destination port         | |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ UDP
   |           Length              |          Checksum             | |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+/
   |0|1|0|0|  IHL  |Type of Service|          Total Length         |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |         Identification        |Flags|      Fragment Offset    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |  Time to Live |   Protocol    |   Header Checksum             |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                       Source IPv4 Address                     |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                     Destination IPv4 Address                  |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Note that 0100 value IP version field express the GUE version as 1 (bits 01) and IP version as 4 (bits 0100).

4.2. Direct encapsulation of IPv6

The format for encapsulating IPv6 directly in UDP is demonstrated below:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+\
|        Source port            |      Destination port         | |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ UDP
|           Length              |          Checksum             | |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+/
|0|1|1|0| Traffic Class |           Flow Label                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Payload Length        |     NextHdr   |   Hop Limit   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                        Source IPv6 Address                    +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                      Destination IPv6 Address                 +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Note that 0110 value IP version field expresses the GUE version as 1 (bits 01) and IP version as 6 (bits 0110).

5. Operation

The figure below illustrates the use of GUE encapsulation between two hosts. Host 1 is sending packets to Host 2. An encapsulator performs encapsulation of packets from Host 1. These encapsulated packets traverse the network as UDP packets. At the decapsulator, packets are decapsulated and sent on to Host 2. Packet flow in the reverse direction need not be symmetric; GUE encapsulation is not required in the reverse path.

```
   +---------------+                       +---------------+
   |               |                       |               |
   |    Host 1     |                       |     Host 2    |
   |               |                       |               |
   +---------------+                       +---------------+
          |                                        ^
          V                                        |
   +---------------+   +---------------+   +---------------+
   |               |   |               |   |               |
   | Encapsulator  |-->|    Layer 3    |-->| Decapsulator  |
   |               |   |    Network    |   |               |
   +---------------+   +---------------+   +---------------+
```

The encapsulator and decapsulator may be co-resident with the corresponding hosts, or may be on separate nodes in the network.

5.1. Network tunnel encapsulation

Network tunneling can be achieved by encapsulating layer 2 or layer 3 packets. In this case the encapsulator and decapsulator nodes are the tunnel endpoints. These could be routers that provide network tunnels on behalf of communicating hosts.

5.2. Transport layer encapsulation

When encapsulating layer 4 packets, the encapsulator and decapsulator should be co-resident with the hosts. In this case, the encapsulation headers are inserted between the IP header and the transport packet. The addresses in the IP header refer to both the endpoints of the encapsulation and the endpoints for terminating the the transport protocol. Note that the transport layer ports in the encapsulated packet are independent of the UDP ports in the outer packet.

Details about performing transport layer encapsulation are discussed in [TOU].

5.3. Encapsulator operation

Encapsulators create GUE data messages, set the fields of the UDP header, set flags and optional extension fields in the GUE header, and forward packets to a decapsulator.

An encapsulator can be an end host originating the packets of a flow, or can be a network device performing encapsulation on behalf of hosts (routers implementing tunnels for instance). In either case, the intended target (decapsulator) is indicated by the outer destination IP address and destination port in the UDP header.

If an encapsulator is tunneling packets -- that is encapsulating packets of layer 2 or layer 3 protocols (e.g. EtherIP, IPIP, ESP tunnel mode) -- it SHOULD follow standard conventions for tunneling of one protocol over another. For instance, if an IP packet is being encapsualated in GUE then diffserv interaction [RFC2983] and ECN propagation for tunnels [RFC6040] SHOULD be followed.

5.4. Decapsulator operation

A decapsulator performs decapsulation of GUE packets. A decapsulator is addressed by the outer destination IP address of a GUE packet. The decapsulator validates packets, including fields of the GUE header.

If a decapsulator receives a GUE packet with an unsupported version, unknown flag, bad header length (too small for included extension fields), unknown control message type, bad protocol number, an unsupported payload type, or an otherwise malformed header, it MUST drop the packet. Such events MAY be logged subject to configuration and rate limiting of logging messages. No error message is returned back to the encapsulator. Note that set flags in a GUE header that are unknown to a decapsulator MUST NOT be ignored. If a GUE packet is received by a decapsulator with unknown flags, the packet MUST be dropped.

5.4.1. Processing a received data message

If a valid data message is received, the UDP headers are removed from the packet. The outer IP header remains intact and the next protocol in the IP header is set to the protocol from the proto field in the GUE header. The resulting packet is then resubmitted into the protocol stack to process that packet as though it was received with the protocol in the GUE header.

As an example, consider that a data message is received where GUE encapsulates an IP packet. In this case proto field in the GUE header is set 94 for IPIP:

```
+-------------------------------------+
|   IP header (next proto = 17,UDP)   |
|-------------------------------------|
|                 UDP                 |
|-------------------------------------|
|         GUE (proto = 94,IPIP)       |
|-------------------------------------|
|         IP header and packet        |
+-------------------------------------+
```

The receiver removes the UDP and GUE headers and sets the next protocol field in the IP packet to IPIP, which is derived from the GUE proto field. The resultant packet would have the format:

```
+-------------------------------------+
|   IP header (next proto = 94,IPIP)  |
|-------------------------------------|
|         IP header and packet        |
+-------------------------------------+
```

This packet is then resubmitted into the protocol stack to be processed as an IPIP packet.

5.4.2. Processing a received control message

If a valid control message is received, the packet MUST be processed as a control message. The specific processing to be performed depends on the ctype in the GUE header.

5.5. Router and switch operation

Routers and switches SHOULD forward GUE packets as standard UDP/IP packets. The outer five-tuple should contain sufficient information to perform flow classification corresponding to the flow of the inner packet. A router does not normally need to parse a GUE header, and none of the flags or extension fields in the GUE header are expected to affect routing. In cases where the outer five-tuple does not provide sufficient entropy for flow classification, for instance UDP ports are fixed to provide connection semantics (section 5.6.1), then the encapsulated packet MAY be parsed to determine flow entropy.

A router MUST NOT modify a GUE header when forwarding a packet. It MAY encapsulate a GUE packet in another GUE packet, for instance to implement a network tunnel (i.e. by encapsulating an IP packet with a GUE payload in another IP packet as a GUE payload). In this case, the router takes the role of an encapsulator, and the corresponding decapsulator is the logical endpoint of the tunnel. When encapsulating a GUE packet within another GUE packet, there are no provisions to automatically GUE copy flags or fields to the outer GUE header. Each layer of encapsulation is considered independent.

5.6. Middlebox interactions

A middle box MAY interpret some flags and extension fields of the GUE header for classification purposes, but is not required to understand any of the flags or extension fields in GUE packets. A middle box MUST NOT drop a GUE packet merely because there are flags unknown to it. The header length in the GUE header allows a middlebox to inspect

the payload packet without needing to parse the flags or extension fields.

5.6.1. Inferring connection semantics

A middlebox might infer bidirectional connection semantics for a UDP flow. For instance, a stateful firewall might create a five-tuple rule to match flows on egress, and a corresponding five-tuple rule for matching ingress packets where the roles of source and destination are reversed for the IP addresses and UDP port numbers. To operate in this environment, a GUE tunnel should be configured to assume connected semantics defined by the UDP five tuple and the use of GUE encapsulation needs to be symmetric between both endpoints. The source port set in the UDP header MUST be the destination port the peer would set for replies. In this case the UDP source port for a tunnel would be a fixed value and not set to be flow entropy as described in section 5.11.

The selection of whether to make the UDP source port fixed or set to a flow entropy value for each packet sent SHOULD be configurable for a tunnel. The default MUST be to set the flow entropy value in the UDP source port.

5.6.2. NAT

IP address and port translation can be performed on the UDP/IP headers adhering to the requirements for NAT with UDP [RFC4787]. In the case of stateful NAT, connection semantics MUST be applied to a GUE tunnel as described in section 5.6.1. GUE endpoints MAY also invoke STUN [RFC5389] or ICE [RFC5245] to manage NAT port mappings for encapsulations.

5.7. Checksum Handling

The potential for mis-delivery of packets due to corruption of IP, UDP, or GUE headers needs to be considered. Historically, the UDP checksum would be considered sufficient as a check against corruption of either the UDP header and payload or the IP addresses. Encapsulation protocols, such as GUE, can be originated or terminated on devices incapable of computing the UDP checksum for packet. This section discusses the requirements around checksum and alternatives that might be used when an endpoint does not support UDP checksum.

5.7.1. Requirements

One of the following requirements MUST be met:

o UDP checksums are enabled (for IPv4 or IPv6).

o The GUE header checksum is used (defined in [GUEEXTENS]).

o Use zero UDP checksums. This is always permissible with IPv4; in IPv6, they can only be used in accordance with applicable requirements in [RFC8086], [RFC6935], and [RFC6936].

5.7.2. UDP Checksum with IPv4

For UDP in IPv4, the UDP checksum MUST be processed as specified in [RFC768] and [RFC1122] for both transmit and receive. An encapsulator MAY set the UDP checksum to zero for performance or implementation considerations. The IPv4 header includes a checksum that protects against mis-delivery of the packet due to corruption of IP addresses. The UDP checksum potentially provides protection against corruption of the UDP header, GUE header, and GUE payload. Enabling or disabling the use of checksums is a deployment consideration that should take into account the risk and effects of packet corruption, and whether the packets in the network are already adequately protected by other, possibly stronger mechanisms such as the Ethernet CRC. If an encapsulator sets a zero UDP checksum for IPv4, it SHOULD use the GUE header checksum as described in [GUEEXTENS] assuming there are no other mechanisms used to protect the GUE packet.

When a decapsulator receives a packet, the UDP checksum field MUST be processed. If the UDP checksum is non-zero, the decapsulator MUST verify the checksum before accepting the packet. By default, a decapsulator SHOULD accept UDP packets with a zero checksum. A node MAY be configured to disallow zero checksums per [RFC1122]. Configuration of zero checksums can be selective. For instance, zero checksums might be disallowed from certain hosts that are known to be sending over paths subject to packet corruption. If verification of a non-zero checksum fails, a decapsulator lacks the capability to verify a non-zero checksum, or a packet with a zero-checksum was received and the decapsulator is configured to disallow, the packet MUST be dropped.

5.7.3. UDP Checksum with IPv6

In IPv6, there is no checksum in the IPv6 header that protects against mis-delivery due to address corruption. Therefore, when GUE is used over IPv6, either the UDP checksum or the GUE header checksum SHOULD be used unless there are alternative mechanisms in use that protect against misdelivery. The UDP checksum and GUE header checksum SHOULD not be used at the same time since that would be mostly redundant.

If neither the UDP checksum or the GUE header checksum is used, then

the requirements for using zero IPv6 UDP checksums in [RFC6935] and [RFC6936] MUST be met.

When a decapsulator receives a packet, the UDP checksum field MUST be processed. If the UDP checksum is non-zero, the decapsulator MUST verify the checksum before accepting the packet. By default a decapsulator MUST only accept UDP packets with a zero checksum if the GUE header checksum is used and is verified. If verification of a non-zero checksum fails, a decapsulator lacks the capability to verify a non-zero checksum, or a packet with a zero-checksum and no GUE header checksum was received, the packet MUST be dropped.

5.8. MTU and fragmentation

Standard conventions for handling of MTU (Maximum Transmission Unit) and fragmentation in conjunction with networking tunnels (encapsulation of layer 2 or layer 3 packets) SHOULD be followed. Details are described in MTU and Fragmentation Issues with In-the-Network Tunneling [RFC4459].

If a packet is fragmented before encapsulation in GUE, all the related fragments MUST be encapsulated using the same UDP source port. An operator SHOULD set MTU to account for encapsulation overhead and reduce the likelihood of fragmentation.

Alternative to IP fragmentation, the GUE fragmentation extension can be used. GUE fragmentation is described in [GUEEXTENS].

5.9. Congestion control

Per requirements of [RFC5405], if the IP traffic encapsulated with GUE implements proper congestion control no additional mechanisms should be required.

In the case that the encapsulated traffic does not implement any or sufficient control, or it is not known whether a transmitter will consistently implement proper congestion control, then congestion control at the encapsulation layer MUST be provided per [RFC5405]. Note that this case applies to a significant use case in network virtualization in which guests run third party networking stacks that cannot be implicitly trusted to implement conformant congestion control.

Out of band mechanisms such as rate limiting, Managed Circuit Breaker [CIRCBRK], or traffic isolation MAY be used to provide rudimentary congestion control. For finer-grained congestion control that allows alternate congestion control algorithms, reaction time within an RTT, and interaction with ECN, in-band mechanisms might be

warranted.

5.10. Multicast

GUE packets can be multicast to decapsulators using a multicast destination address in the encapsulating IP headers. Each receiving host will decapsulate the packet independently following normal decapsulator operations. The receiving decapsulators need to agree on the same set of GUE parameters and properties; how such an agreement is reached is outside the scope of this document.

GUE allows encapsulation of unicast, broadcast, or multicast traffic. Flow entropy (the value in the UDP source port) can be generated from the header of encapsulated unicast or broadcast/multicast packets at an encapsulator. The mapping mechanism between the encapsulated multicast traffic and the multicast capability in the IP network is transparent and independent of the encapsulation and is otherwise outside the scope of this document.

5.11. Flow entropy for ECMP

5.11.1. Flow classification

A major objective of using GUE is that a network device can perform flow classification corresponding to the flow of the inner encapsulated packet based on the contents in the outer headers.

Hardware devices commonly perform hash computations on packet headers to classify packets into flows or flow buckets. Flow classification is done to support load balancing of flows across a set of networking resources. Examples of such load balancing techniques are Equal Cost Multipath routing (ECMP), port selection in Link Aggregation, and NIC device Receive Side Scaling (RSS). Hashes are usually either a three-tuple hash of IP protocol, source address, and destination address; or a five-tuple hash consisting of IP protocol, source address, destination address, source port, and destination port. Typically, networking hardware will compute five-tuple hashes for TCP and UDP, but only three-tuple hashes for other IP protocols. Since the five-tuple hash provides more granularity, load balancing can be finer-grained with better distribution. When a packet is encapsulated with GUE and connection semantics are not applied, the source port in the outer UDP packet is set to a flow entropy value that corresponds to the flow of the inner packet. When a device computes a five-tuple hash on the outer UDP/IP header of a GUE packet, the resultant value classifies the packet per its inner flow.

Examples of deriving flow entropy for encapsulation are:

- o If the encapsulated packet is a layer 4 packet, TCP/IPv4 for instance, the flow entropy could be based on the canonical five-tuple hash of the inner packet.

- o If the encapsulated packet is an AH transport mode packet with TCP as next header, the flow entropy could be a hash over a three-tuple: TCP protocol and TCP ports of the encapsulated packet.

- o If a node is encrypting a packet using ESP tunnel mode and GUE encapsulation, the flow entropy could be based on the contents of the clear-text packet. For instance, a canonical five-tuple hash for a TCP/IP packet could be used.

[RFC6438] discusses methods to compute and set flow entropy value for IPv6 flow labels. Such methods can also be used to create flow entropy values for GUE.

5.11.2. Flow entropy properties

The flow entropy is the value set in the UDP source port of a GUE packet. Flow entropy in the UDP source port SHOULD adhere to the following properties:

- o The value set in the source port is within the ephemeral port range (49152 to 65535 [RFC6335]). Since the high order two bits of the port are set to one, this provides fourteen bits of entropy for the value.

- o The flow entropy has a uniform distribution across encapsulated flows.

- o An encapsulator MAY occasionally change the flow entropy used for an inner flow per its discretion (for security, route selection, etc). To avoid thrashing or flapping the value, the flow entropy used for a flow SHOULD NOT change more than once every thirty seconds (or a configurable value).

- o Decapsulators, or any networking devices, SHOULD NOT attempt to interpret flow entropy as anything more than an opaque value. Neither should they attempt to reproduce the hash calculation used by an encapasulator in creating a flow entropy value. They MAY use the value to match further receive packets for steering decisions, but MUST NOT assume that the hash uniquely or permanently identifies a flow.

- o Input to the flow entropy calculation is not restricted to ports and addresses; input could include flow label from an IPv6 packet, SPI from an ESP packet, or other flow related state in the encapsulator that is not necessarily conveyed in the packet.

- o The assignment function for flow entropy SHOULD be randomly seeded to mitigate denial of service attacks. The seed SHOULD be changed periodically.

5.12 Negotiation of acceptable flags and extension fields

An encapsulator and decapsulator need to achieve agreement about GUE parameters will be used in communications. Parameters include GUE version, flags and extension fields that can be used, security algorithms and keys, supported protocols and control messages, etc. This document proposes different general methods to accomplish this, however the details of implementing these are considered out of scope.

General methods for this are:

- o Configuration. The parameters used for a tunnel are configured at each endpoint.

- o Negotiation. A tunnel negotiation can be performed. This could be accomplished in-band of GUE using control messages or private data.

- o Via a control plane. Parameters for communicating with a tunnel endpoint can be set in a control plane protocol (such as that needed for nvo3).

- o Via security negotiation. Use of security typically implies a key exchange between endpoints. Other GUE parameters may be conveyed as part of that process.

6. Motivation for GUE

This section presents the motivation for GUE with respect to other encapsulation methods.

6.1. Benefits of GUE

* GUE is a generic encapsulation protocol. GUE can encapsulate protocols that are represented by an IP protocol number. This includes layer 2, layer 3, and layer 4 protocols.

* GUE is an extensible encapsulation protocol. Standardized

optional data such as security, virtual networking identifiers, fragmentation are being defined.

* For extensilbity, GUE uses flag fields as opposed to TLVs as some other encapsulation protocols do. Flag fields are strictly ordered, allow random access, and are efficient in use of header space.

* GUE allows private data to be sent as part of the encapsulation. This permits experimentation or customization in deployment.

* GUE allows sending of control messages such as OAM using the same GUE header format (for routing purposes) as normal data messages.

* GUE maximizes deliverability of non-UDP and non-TCP protocols.

* GUE provides a means for exposing per flow entropy for ECMP for atypical protocols such as SCTP, DCCP, ESP, etc.

6.2 Comparison of GUE to other encapsulations

A number of different encapsulation techniques have been proposed for the encapsulation of one protocol over another. EtherIP [RFC3378] provides layer 2 tunneling of Ethernet frames over IP. GRE [RFC2784], MPLS [RFC4023], and L2TP [RFC2661] provide methods for tunneling layer 2 and layer 3 packets over IP. NVGRE [RFC7637] and VXLAN [RFC7348] are proposals for encapsulation of layer 2 packets for network virtualization. IPIP [RFC2003] and Generic packet tunneling in IPv6 [RFC2473] provide methods for tunneling IP packets over IP.

Several proposals exist for encapsulating packets over UDP including ESP over UDP [RFC3948], TCP directly over UDP [TCPUDP], VXLAN [RFC7348], LISP [RFC6830] which encapsulates layer 3 packets, MPLS/UDP [7510], and Generic UDP Encapsulation for IP Tunneling (GRE over UDP)[RFC8086]. Generic UDP tunneling [GUT] is a proposal similar to GUE in that it aims to tunnel packets of IP protocols over UDP.

GUE has the following discriminating features:

o UDP encapsulation leverages specialized network device processing for efficient transport. The semantics for using the UDP source port for flow entropy as input to ECMP are defined in section 5.11.

o GUE permits encapsulation of arbitrary IP protocols, which includes layer 2 3, and 4 protocols.

- o Multiple protocols can be multiplexed over a single UDP port number. This is in contrast to techniques to encapsulate protocols over UDP using a protocol specific port number (such as ESP/UDP, GRE/UDP, SCTP/UDP). GUE provides a uniform and extensible mechanism for encapsulating all IP protocols in UDP with minimal overhead (four bytes of additional header).

- o GUE is extensible. New flags and extension fields can be defined.

- o The GUE header includes a header length field. This allows a network node to inspect an encapsulated packet without needing to parse the full encapsulation header.

- o Private data in the encapsulation header allows local customization and experimentation while being compatible with processing in network nodes (routers and middleboxes).

- o GUE includes both data messages (encapsulation of packets) and control messages (such as OAM).

- o The flags-field model facilitates efficient implementation of extensibility in hardware.

  For instance a TCAM can be use to parse a known set of N flags where the number of entries in the TCAM is 2^N.

  For comparison, the number of TCAM entries needed to parse a set of N arbitrarily ordered TLVS is approximately e*N!.

7. Security Considerations

There are two important considerations of security with respect to GUE.

- o Authentication and integrity of the GUE header.

- o Authentication, integrity, and confidentiality of the GUE payload.

GUE security is provided by extensions for security defined in [GUEEXTENS]. These extensions include methods to authenticate the GUE header and encrypt the GUE payload.

The GUE header can be authenticated using a security extension for an HMAC. Securing the GUE payload can be accomplished use of the GUE Payload Transform. This extension can be used to perform DTLS in the payload of a GUE packet to encrypt the payload.

A hash function for computing flow entropy (section 5.11) SHOULD be randomly seeded to mitigate some possible denial service attacks.

8. IANA Consideration

8.1. UDP source port

A user UDP port number assignment for GUE has been assigned:

```
       Service Name: gue
       Transport Protocol(s): UDP
       Assignee: Tom Herbert <tom@herbertland.com>
       Contact: Tom Herbert <tom@herbertland.com>
       Description: Generic UDP Encapsulation
       Reference: draft-herbert-gue
       Port Number: 6080
       Service Code: N/A
       Known Unauthorized Uses: N/A
       Assignment Notes: N/A
```

8.2. GUE version number

IANA is requested to set up a registry for the GUE version number. The GUE version number is 2 bits containing four possible values. This document defines version 0 and 1. New values are assigned in accordance with RFC Required policy [RFC5226].

Version number	Description	Reference
0	Version 0	This document
1	Version 1	This document
2..3	Unassigned	

8.3. Control types

IANA is requested to set up a registry for the GUE control types. Control types are 8 bit values. New values for control types 1-127 are assigned in accordance with RFC Required policy [RFC5226].

Control type	Description	Reference
0	Need further interpretation	This document
1..127	Unassigned	
128..255	User defined	This document

8.4. Flag-fields

IANA is requested to create a "GUE flag-fields" registry to allocate flags and extension fields used with GUE. This shall be a registry of bit assignments for flags, length of extension fields for corresponding flags, and descriptive strings. There are sixteen bits for primary GUE header flags (bit number 0-15). New values are assigned in accordance with RFC Required policy [RFC5226].

Flags bits	Field size	Description	Reference
Bit 0	4 bytes	VNID	[GUE4NVO3]
Bit 1..3	001->8 bytes 010->16 bytes 011->32 bytes	Security	[GUEEXTENS]
Bit 4	8 bytes	Fragmen- tation	[GUEEXTENS]
Bit 5	4 bytes	Payload transform	[GUEEXTENS]
Bit 6	4 bytes	Remote checksum offload	[GUEEXTENS]
Bit 7	4 bytes	Checksum	[GUEEXTENS]
Bit 8..15		Unassigned	

New flags are to be allocated from high to low order bit contiguously without holes.

9. Acknowledgements

The authors would like to thank David Liu, Erik Nordmark, Fred Templin, Adrian Farrel, Bob Briscoe, and Murray Kucherawy for valuable input on this draft.

10. References

10.1. Normative References

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <http://www.rfc-editor.org/info/rfc768>.

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <http://www.rfc-editor.org/info/rfc1122>.

[RFC2434] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", RFC 2434, DOI 10.17487/RFC2434, October 1998, <http://www.rfc-editor.org/info/rfc2434>.

[RFC2983] Black, D., "Differentiated Services and Tunnels", RFC 2983, DOI 10.17487/RFC2983, October 2000, <http://www.rfc-editor.org/info/rfc2983>.

[RFC6040] Briscoe, B., "Tunnelling of Explicit Congestion Notification", RFC 6040, DOI 10.17487/RFC6040, November 2010, <http://www.rfc-editor.org/info/rfc6040>.

[RFC6935] Eubanks, M., Chimento, P., and M. Westerlund, "IPv6 and UDP Checksums for Tunneled Packets", RFC 6935, DOI 10.17487/RFC6935, April 2013, <http://www.rfc-editor.org/info/rfc6935>.

[RFC6936] Fairhurst, G. and M. Westerlund, "Applicability Statement for the Use of IPv6 UDP Datagrams with Zero Checksums", RFC 6936, DOI 10.17487/RFC6936, April 2013, <http://www.rfc-editor.org/info/rfc6936>.

[RFC4459] Savola, P., "MTU and Fragmentation Issues with In-the-Network Tunneling", RFC 4459, DOI 10.17487/RFC4459, April 2006, <http://www.rfc-editor.org/info/rfc4459>.

10.2. Informative References

[RFC3828] Larzon, L-A., Degermark, M., Pink, S., Jonsson, L-E., Ed., and G. Fairhurst, Ed., "The Lightweight User Datagram Protocol (UDP-Lite)", RFC 3828, July 2004, <http://www.rfc-editor.org/info/rfc3828>.

[RFC7348] Mahalingam, M., Dutt, D., Duda, K., Agarwal, P., Kreeger, L., Sridhar, T., Bursell, M., and C. Wright, "Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks", RFC 7348, August 2014, <http://www.rfc-editor.org/info/rfc7348>.

[RFC7605] Touch, J., "Recommendations on Using Assigned Transport Port Numbers", BCP 165, RFC 7605, DOI 10.17487/RFC7605, August 2015, <http://www.rfc-editor.org/info/rfc7605>.

[RFC7637] Garg, P., Ed., and Y. Wang, Ed., "NVGRE: Network Virtualization Using Generic Routing Encapsulation", RFC 7637, DOI 10.17487/RFC7637, September 2015,

<http://www.rfc-editor.org/info/rfc7637>.

[RFC8086] Yong, L., Ed., Crabbe, E., Xu, X., and T. Herbert, "GRE-in-UDP Encapsulation", RFC 8086, DOI 10.17487/RFC8086, March 2017, <http://www.rfc-editor.org/info/rfc8086>.

[RFC7510] Xu, X., Sheth, N., Yong, L., Callon, R., and D. Black, "Encapsulating MPLS in UDP", RFC 7510, DOI 10.17487/RFC7510, April 2015, <http://www.rfc-editor.org/info/rfc7510>.

[RFC4340] Kohler, E., Handley, M., and S. Floyd, "Datagram Congestion Control Protocol (DCCP)", RFC 4340, DOI 10.17487/RFC4340, March 2006, <http://www.rfc-editor.org/info/rfc4340>.

[RFC4787] Audet, F., Ed., and C. Jennings, "Network Address Translation (NAT) Behavioral Requirements for Unicast UDP", BCP 127, RFC 4787, DOI 10.17487/RFC4787, January 2007, <http://www.rfc-editor.org/info/rfc4787>.

[RFC5389] Rosenberg, J., Mahy, R., Matthews, P., and D. Wing, "Session Traversal Utilities for NAT (STUN)", RFC 5389, DOI 10.17487/RFC5389, October 2008, <http://www.rfc-editor.org/info/rfc5389>.

[RFC5285] Rosenberg, J., "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols", RFC 5245, DOI 10.17487/RFC5245, April 2010, <http://www.rfc-editor.org/info/rfc5245>.

[RFC5405] Eggert, L. and G. Fairhurst, "Unicast UDP Usage Guidelines for Application Designers", BCP 145, RFC 5405, DOI 10.17487/RFC5405, November 2008, <http://www.rfc-editor.org/info/rfc5405>.

[RFC6438] Carpenter, B. and S. Amante, "Using the IPv6 Flow Label for Equal Cost Multipath Routing and Link Aggregation in Tunnels", RFC 6438, DOI 10.17487/RFC6438, November 2011, <http://www.rfc-editor.org/info/rfc6438>.

[RFC2003] Perkins, C., "IP Encapsulation within IP", RFC 2003, DOI 10.17487/RFC2003, October 1996, <http://www.rfc-editor.org/info/rfc2003>.

[RFC3948] Huttunen, A., Swander, B., Volpe, V., DiBurro, L., and M. Stenberg, "UDP Encapsulation of IPsec ESP Packets", RFC

3948, DOI 10.17487/RFC3948, January 2005, <http://www.rfc-editor.org/info/rfc3948>.

[RFC6830] Farinacci, D., Fuller, V., Meyer, D., and D. Lewis, "The Locator/ID Separation Protocol (LISP)", RFC 6830, DOI 10.17487/RFC6830, January 2013, <http://www.rfc-editor.org/info/rfc6830>.

[RFC3378] Housley, R. and S. Hollenbeck, "EtherIP: Tunneling Ethernet Frames in IP Datagrams", RFC 3378, DOI 10.17487/RFC3378, September 2002, <http://www.rfc-editor.org/info/rfc3378>.

[RFC2784] Farinacci, D., Li, T., Hanks, S., Meyer, D., and P. Traina, "Generic Routing Encapsulation (GRE)", RFC 2784, DOI 10.17487/RFC2784, March 2000, <http://www.rfc-editor.org/info/rfc2784>.

[RFC4023] Worster, T., Rekhter, Y., and E. Rosen, Ed., "Encapsulating MPLS in IP or Generic Routing Encapsulation (GRE)", RFC 4023, DOI 10.17487/RFC4023, March 2005, <http://www.rfc-editor.org/info/rfc4023>.

[RFC2661] Townsley, W., Valencia, A., Rubens, A., Pall, G., Zorn, G., and B. Palter, "Layer Two Tunneling Protocol "L2TP"", RFC 2661, DOI 10.17487/RFC2661, August 1999, <http://www.rfc-editor.org/info/rfc2661>.

[GUEEXTENS] Herbert, T., Yong, L., and Templin, F., "Extensions for Generic UDP Encapsulation" draft-herbert-gue-extensions-00

[GUE4NVO3] Yong, L., Herbert, T., Zia, O., "Generic UDP Encapsulation (GUE) for Network Virtualization Overlay" draft-hy-nvo3-gue-4-nvo-03

[GUESEC] Yong, L., Herbert, T., "Generic UDP Encapsulation (GUE) for Secure Transport" draft-hy-gue-4-secure-transport-03

[TCPUDP] Chesire, S., Graessley, J., and McGuire, R., "Encapsulation of TCP and other Transport Protocols over UDP" draft-cheshire-tcp-over-udp-00

[TOU] Herbert, T., "Transport layer protocols over UDP" draft-herbert-transports-over-udp-00

[GUT] Manner, J., Varia, N., and Briscoe, B., "Generic UDP Tunnelling (GUT) draft-manner-tsvwg-gut-02.txt"

[CIRCBRK] Fairhurst, G., "Network Transport Circuit Breakers", draft-ietf-tsvwg-circuit-breaker-15

[LCO]   Cree, E., https://www.kernel.org/doc/Documentation/networking/checksum-offloads.txt

Appendix A: NIC processing for GUE

This appendix provides some guidelines for Network Interface Cards (NICs) to implement common offloads and accelerations to support GUE. Note that most of this discussion is generally applicable to other methods of UDP based encapsulation.

A.1. Receive multi-queue

Contemporary NICs support multiple receive descriptor queues (multi-queue). Multi-queue enables load balancing of network processing for a NIC across multiple CPUs. On packet reception, a NIC selects the appropriate queue for host processing. Receive Side Scaling is a common method which uses the flow hash for a packet to index an indirection table where each entry stores a queue number. Flow Director and Accelerated Receive Flow Steering (aRFS) allow a host to program the queue that is used for a given flow which is identified either by an explicit five-tuple or by the flow's hash.

GUE encapsulation is compatible with multi-queue NICs that support five-tuple hash calculation for UDP/IP packets as input to RSS. The flow entropy in the UDP source port ensures classification of the encapsulated flow even in the case that the outer source and destination addresses are the same for all flows (e.g. all flows are going over a single tunnel).

By default, UDP RSS support is often disabled in NICs to avoid out-of-order reception that can occur when UDP packets are fragmented. As discussed above, fragmentation of GUE packets is be mostly avoided by fragmenting packets before entering a tunnel, GUE fragmentation, path MTU discovery in higher layer protocols, or operator adjusting MTUs. Other UDP traffic might not implement such procedures to avoid fragmentation, so enabling UDP RSS support in the NIC might be a considered tradeoff during configuration.

A.2. Checksum offload

Many NICs provide capabilities to calculate standard ones complement payload checksum for packets in transmit or receive. When using GUE encapsulation, there are at least two checksums that are of interest: the encapsulated packet's transport checksum, and the UDP checksum in the outer header.

A.2.1. Transmit checksum offload

NICs can provide a protocol agnostic method to offload transmit checksum (NETIF_F_HW_CSUM in Linux parlance) that can be used with GUE. In this method, the host provides checksum related parameters in a transmit descriptor for a packet. These parameters include the starting offset of data to checksum, the length of data to checksum, and the offset in the packet where the computed checksum is to be written. The host initializes the checksum field to pseudo header checksum.

In the case of GUE, the checksum for an encapsulated transport layer packet, a TCP packet for instance, can be offloaded by setting the appropriate checksum parameters.

NICs typically can offload only one transmit checksum per packet, so simultaneously offloading both an inner transport packet's checksum and the outer UDP checksum is likely not possible.

If an encapsulator is co-resident with a host, then checksum offload may be performed using remote checksum offload (described in [GUEEXTENS]). Remote checksum offload relies on NIC offload of the simple UDP/IP checksum which is commonly supported even in legacy devices. In remote checksum offload, the outer UDP checksum is set and the GUE header includes an option indicating the start and offset of the inner "offloaded" checksum. The inner checksum is initialized to the pseudo header checksum. When a decapsulator receives a GUE packet with the remote checksum offload option, it completes the offload operation by determining the packet checksum from the indicated start point to the end of the packet, and then adds this into the checksum field at the offset given in the option. Computing the checksum from the start to end of packet is efficient if checksum-complete is provided on the receiver.

Another alternative when an encapsulator is co-resident with a host is to perform Local Checksum Offload [LCO]. In this method, the inner transport layer checksum is offloaded and the outer UDP checksum can be deduced based on the fact that the portion of the packet covered by the inner transport checksum will sum to zero (or at least the bit wise "not" of the inner pseudo header).

A.2.2. Receive checksum offload

GUE is compatible with NICs that perform a protocol agnostic receive checksum (CHECKSUM_COMPLETE in Linux parlance). In this technique, a NIC computes a ones complement checksum over all (or some predefined portion) of a packet. The computed value is provided to the host stack in the packet's receive descriptor. The host driver can use

this checksum to "patch up" and validate any inner packet transport checksum, as well as the outer UDP checksum if it is non-zero.

Many legacy NICs don't provide checksum-complete but instead provide an indication that a checksum has been verified (CHECKSUM_UNNECESSARY in Linux). Usually, such validation is only done for simple TCP/IP or UDP/IP packets. If a NIC indicates that a UDP checksum is valid, the checksum-complete value for the UDP packet is the "not" of the pseudo header checksum. In this way, checksum-unnecessary can be converted to checksum-complete. So, if the NIC provides checksum-unnecessary for the outer UDP header in an encapsulation, checksum conversion can be done so that the checksum-complete value is derived and can be used by the stack to validate checksums in the encapsulated packet.

A.3. Transmit Segmentation Offload

Transmit Segmentation Offload (TSO) is a NIC feature where a host provides a large (>MTU size) TCP packet to the NIC, which in turn splits the packet into separate segments and transmits each one. This is useful to reduce CPU load on the host.

The process of TSO can be generalized as:

- Split the TCP payload into segments which allow packets with size less than or equal to MTU.

- For each created segment:

  1. Replicate the TCP header and all preceding headers of the original packet.

  2. Set payload length fields in any headers to reflect the length of the segment.

  3. Set TCP sequence number to correctly reflect the offset of the TCP data in the stream.

  4. Recompute and set any checksums that either cover the payload of the packet or cover header which was changed by setting a payload length.

Following this general process, TSO can be extended to support TCP encapsulation in GUE. For each segment the Ethernet, outer IP, UDP header, GUE header, inner IP header (if tunneling), and TCP headers are replicated. Any packet length header fields need to be set properly (including the length in the outer UDP header), and checksums need to be set correctly (including the outer UDP checksum if being used).

To facilitate TSO with GUE, it is recommended that extension fields do not contain values that need to be updated on a per segment basis. For example, extension fields should not include checksums, lengths, or sequence numbers that refer to the payload. If the GUE header does not contain such fields then the TSO engine only needs to copy the bits in the GUE header when creating each segment and does not need to parse the GUE header.

A.4. Large Receive Offload

Large Receive Offload (LRO) is a NIC feature where packets of a TCP connection are reassembled, or coalesced, in the NIC and delivered to the host as one large packet. This feature can reduce CPU utilization in the host.

LRO requires significant protocol awareness to be implemented correctly and is difficult to generalize. Packets in the same flow need to be unambiguously identified. In the presence of tunnels or network virtualization, this may require more than a five-tuple match (for instance packets for flows in two different virtual networks may have identical five-tuples). Additionally, a NIC needs to perform validation over packets that are being coalesced, and needs to fabricate a single meaningful header from all the coalesced packets.

The conservative approach to supporting LRO for GUE would be to assign packets to the same flow only if they have identical five-tuple and were encapsulated the same way. That is the outer IP addresses, the outer UDP ports, GUE protocol, GUE flags and fields, and inner five tuple are all identical.

Appendix B: Implementation considerations

This appendix is informational and does not constitute a normative part of this document.

B.1. Priveleged ports

Using the source port to contain a flow entropy value disallows the security method of a receiver enforcing that the source port be a privileged port. Privileged ports are defined by some operating systems to restrict source port binding. Unix, for instance, considered port number less than 1024 to be privileged.

Enforcing that packets are sent from a privileged port is widely considered an inadequate security mechanism and has been mostly deprecated. To approximate this behavior, an implementation could restrict a user from sending a packet destined to the GUE port without proper credentials.

B.2. Setting flow entropy as a route selector

An encapsulator generating flow entropy in the UDP source port could modulate the value to perform a type of multipath source routing. Assuming that networking switches perform ECMP based on the flow hash, a sender can affect the path by altering the flow entropy. For instance, a host can store a flow hash in its PCB for an inner flow, and might alter the value upon detecting that packets are traversing a lossy path. Changing the flow entropy for a flow SHOULD be subject to hysteresis (at most once every thirty seconds) to limit the number of out of order packets.

B.3. Hardware protocol implementation considerations

Low level data path protocol, such is GUE, are often supported in high speed network device hardware. Variable length header (VLH) protocols like GUE are often considered difficult to efficiently implement in hardware. In order to retain the important characteristics of an extensible and robust protocol, hardware vendors may practice "constrained flexibility". In this model, only certain combinations or protocol header parameterizations are implemented in hardware fast path. Each such parameterization is fixed length so that the particular instance can be optimized as a fixed length protocol. In the case of GUE this constitutes specific combinations of GUE flags, fields, and next protocol. The selected combinations would naturally be the most common cases which form the "fast path", and other combinations are assumed to take the "slow path".

In time, needs and requirements of the protocol may change which may manifest themselves as new parameterizations to be supported in the fast path. To allow allow this extensibility, a device practicing constrained flexibility should allow the fast path parameterizations to be programmable.

Authors' Addresses

Tom Herbert
Quantonium
4701 Patrick Henry
Santa Clara, CA 95054
US

Email: tom@herbertland.com

Lucy Yong
Huawei USA
5340 Legacy Dr.

Plano, TX 75024
US

Email: lucy.yong@huawei.com

Osama Zia
Microsoft
1 Microsoft Way
Redmond, WA 98029
US

Email: osamaz@microsoft.com