Network Working Group
Internet-Draft
Intended status: Experimental
Expires: January 13, 2011

J. Manner
N. Varis
Aalto University
B. Briscoe
BT
July 12, 2010

Generic UDP Tunnelling (GUT)
draft-manner-tsvwg-gut-02.txt

Abstract

Deploying new transport protocols on the Internet is a well-known problem, as NATs and firewall drop packets with e.g. new protocol types or unidentified TCP options. Tunnelling over UDP is one way to make IP packets hide the actual payload and enable end-to-end delivery. This document proposes a simple UDP tunnelling encapsulation and end-host operation to enable new (and old) IP payloads, e.g., new transport protocols, to be deployed on the Internet.

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 http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 13, 2011.

Copyright Notice

Table of Contents

1. Terminology

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 BCP 14, RFC 2119 [RFC2119].

In addition, we use the following terms:

Native: the IP protocol that will, or has been, encapsulated by GUT, e.g., DCCP, SCTP, etc., even protocols like ICMP or RSVP. We also refer to native IP packet as the IP packet that carries the native IP protocol.

Initiator: the node that has sent the first packet belonging to an exchange by the native protocol.

Responder: the node that received the first packet of the native protocol.

Data packet (D-packet): a GUT packet that carries an encapsulated native protocol.

Control packet (C-packet): a GUT packet that carries only extension header(s).

Data-Control packet (DC-packet): a GUT packet that carries an encapsulated native protocol and an extension header.

2. Introduction

New IP paylods, e.g., transport layer technologies, such as SCTP [RFC4960] and DCCP [RFC4340], have well-known problems with deployment on the Internet. Firewalls drop IP packets with unknown (too new) transport protocol types or protocol extensions, e.g., TCP

options, and NAT boxes do not know how to translate these protocols.

Tunnelling over UDP has often been mentioned as a means to traverse middleboxes. Mostly the solutions are ad-hoc and protocol-specific. In order to make deployment of UDP tunnelling at least somewhat consistent, this document proposes a simple mechanism to realise the goal. The benefit is that with a generic solution we avoid the need to define tunneling specifications for each IP protocol separately. The fundamental goal of GUT is to mitigate the problem of existing NATs and firewalls, while still allowing middleboxes that deliberately want to block to do so.

The basic idea of GUT is to encapsulate the native transport protocol and its payload (in general the whole IP payload) within a UDP packet destined to the well-known port GUT_P. Between the outer UDP header and the inner transport header, we have a 4-byte GUT header that carries information about the encapsulated protocol to help decapsulation on the receiving side. GUT also introduces extension headers.

GUT does not specify, nor need, a specific tunnel setup protocol. It just encapsulates the native protocol and its payload - to any middlebox on the way this looks like a normal UDP flow to port GUT_P.

In other words, this specification, i.e, GUT, only defines

1. The encapsulation of the native IP payload,
2. The GUT header structure and content, and
3. The state machine on the initiator and responder sides.

If the native protocol has a handshake or any back-and-forth messaging, these are run automatically within the UDP-tunnel created by GUT: GUT is meant to be fully transparent to the native protocol. Note that GUT can also tunnel protocol types which do not have any port informations, such as RSVP or ICMP. The GUT encapsulation is agnostic to the IP protocol version being used (IPv4 or IPv6).

3. Basic operation

The basic idea of the protocol is to encapsulate the problematic

native IP payload within a UDP header and send the packet to a well-known UDP port GUT_P - any return packets from the responder will be tunneled to the UDP source port used by the initiator. The responder will get the UDP packets, check the encapsulated payload and evaluate if it wants to receive the packet, reconstruct the native IP packet, and forward it for further processing within the network stack; GUT is not meant to bypass explicit firewall rules and configuration. Figure 1 shows the encapsulation.

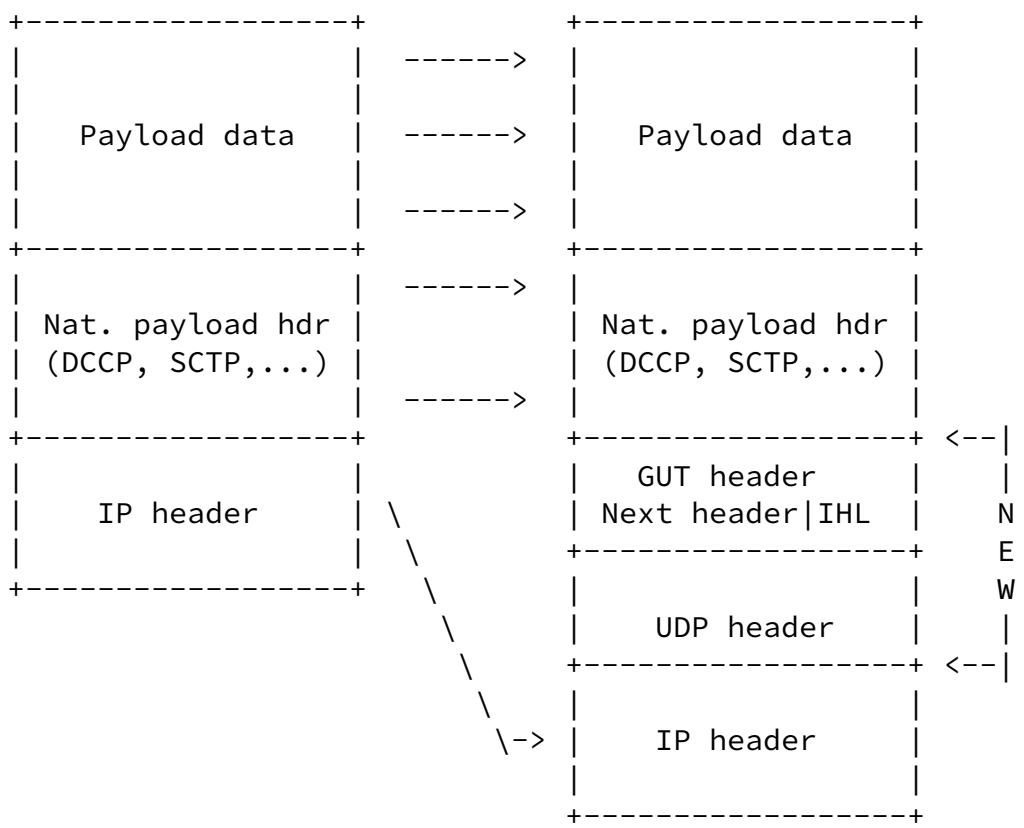

Figure 1: GUT encapsulation

The GUT header is 32 bits and carries three pieces of information that enable reconstructing the native IP packet and perform checksum calculations.

1. The GUT Header Length (12 bits) gives the length in bytes of the headers after the 4-byte GUT header and before the encapsulated payload. Any IP option headers encapsulated from the native IP packet are included. The GUT header itself is not included. Thus, a value of 0 indicates straight encapsulation, no headers between the GUT header and the encapsulated protocol.

2. The IPv4 Header Length (IHL, 4 bits) field from the native IPv4 packet.

3. The IP protocol number (next header, 8 bits) of the native IP packet (v4 and v6). The value 255 is used to indicate a GUT extension header (this value is Reserved in the IP protocol numbers registry). Thus, all extension headers carry the same Next header value but a different internal type.

The first 8-bit field is currently unused.

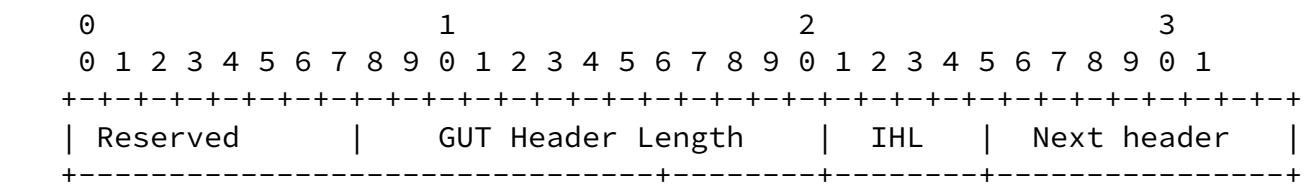

Figure 2: GUT header

3.1. Extension headers

In addition to the GUT main header, we define extendion headers, objects. An extension header can be added into a GUT packet carrying an encapsulated native protocol frame, or it can be sent standalone,

without an actual native protocol, i.e., below the GUT header we have one or more extension headers but no further payload. Any extension headers MUST be appended straight after the GUT main header and before any encapsulated native protocol.

Each extendion header begins with a fixed 32-bit part giving the object Type and object Length and Next header. This is followed by the object Value, which is a whole number of 32-bit words long.

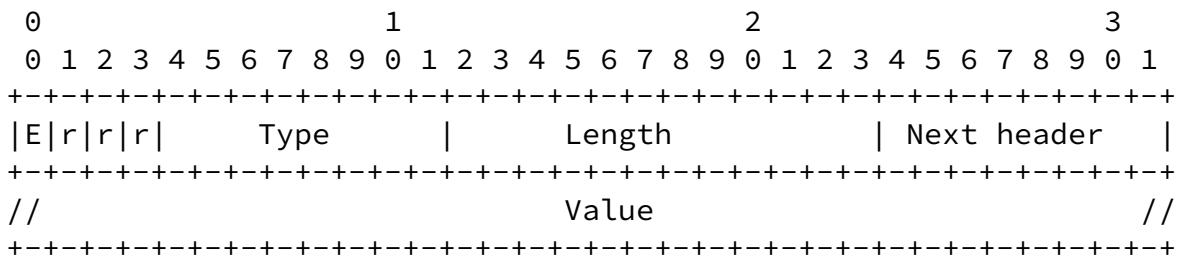

Figure 3: Extension header structure

E flag (extensibility): A value of "0" indicates that if the object is not understood, the whole GUT packet must be silently discarded. A value of "1" indicates that the unknown object can simply be ignored and the rest of the packet processed as usual.

Type (12 bits): An IANA-assigned identifier for the type of object.

Length (12 bits): Length has units of 32-bit words, and measures the length of Value. If there is no Value, Length equals 0. If the Length is not consistent with the contents of the object, the message MUST be discarded.

Value (variable): Value is (therefore) a whole number of 32 bit words. If there is any padding required, the length and location are be defined by the object-specific format information; objects which contain variable length (e.g. string) types may need to include additional length subfields to do so.

This specification introduces three extension headers. All these extension headers are sent as Control packets, i.e., without an

encapsulated native protocol.

- TEST: to test if the responder supports GUT.
- TEST-REPLY: a reply to a TEST.

A TEST object can be sent to check whether the receiving host supports GUT. The Responder SHOULD reply with a TEST-REPLY if it supports GUT. A TEST object MAY carry a Value field giving a nonce that should be send back in the TEST-REPLY. A nonce MUST be copied to the TEST-REPLY if the Responder is going to answer the TEST. The Responder MUST NOT store a flow state when receiving a TEST message. The TEST handshake is a similar function than the ICMP ECHO, i.e., both can be used to test if the other end point is alive (without assurance, though, since replying is voluntary).

- KEEPALIVE: to maintain a flow state on a middlebox.

A native protocol, e.g., a TCP flow, may be using keepalive messages to maintain middlebox states for a flow. Since GUT makes the native flow appear to be UDP, the timeouts of the native flow may be longer than what the middlebox is configured to use for a UDP flow. A GUT node MAY send KEEPALIVE headers in Control packets to maintain the state in middleboxes on the path. A GUT node that receives KEEPALIVE Control packets MUST discard them. The keepalives are always associated to a certain unique GUT flow. Therefore, the receiver MAY use the keepalives as an indication that the native protocol flow is still active and thus maintain it's own states for the flow.

3.2. Sender operation

In implementation terms, the GUT process running on a host or router (proxy) receives the native IP packet, the whole packet including the IP header, grabs the bytes immediately after the native IP static header, adds a UDP and GUT header, and sends the packet to the destination indicated in the received IP packet. Header checksums are recalculated and IP options are encapsulated.

The sender MAY limit the UDP checksum coverage to just the octets outside the native transport header, i.e., not include the encapsulated payload. This is purely for efficiency - to avoid

running the whole payload through the checksum calculation twice -

with all the memory shifts that will entail.

The source port in the UDP header MAY be chosen freely, although if the native encapsulated protocol had a notion of port numbers, the sender SHOULD choose the same source port (note that the source port may already be used by another process, thus the processing may have to choose another port number). The IP header indicates a UDP transport, the GUT header is the first 4 bytes of the UDP payload. The Total Length field in the IP header gives the length of the whole datagram including the encapsulated transport protocol packet. The GUT IP header length gives the length of any encapsulated IPv4 Options; this value is actually copied directly from the native IP packet.

The current value of GUT_P is 4887 (rule of thumb 1-800-GUTP)

The GUT daemon is not considered as an IP hop, thus, when the sender builds the IP header, it MUST use the TTL from the native IP packet. Similarly any ECN and DSCP bits are copied from the native IP packet to the outgoing IP packet.

GUT adds 12 octets of headers (UDP+GUT) which may cause fragmentation to happen. GUT is meant as transparent a functionality as possible. Thus, in principle GUT relies on the network stack to do IP packet fragmentation and reassembly.

If the native IP packet had IP options, those are preserved within the GUT encapsulation. Here the processing must store the original IHL-field from the native IPv4 packet to be used on the responder side for building the native IP packet properly.

It is possible, that the sender gets fragmented IP packets from the network stack to be GUT-encapsulated. In such case, the GUT process MUST reassemble the whole IP packet before adding the UDP and GUT header. The subsequent packet is given back to the network stack for transmission, and may be fragmented at that point to fit the MTU of the link. The initiator may decide to use Path MTU Discovery when the first packet of a flow is received for encapsulation, yet, this will add a delay to the transmission of the first packet and may result in the native protocol to react accordingly.

The initiator has to store state, the 5-tuple or 3-tuple (or any information that enables tracking a particular flow), for each initiated GUT encapsulation. This state is needed for properly catching potential return packets of the native IP protocol from the responder (e.g. DCCP, SCTP). This state can be made to expire after a certain timeout, or an implementation can decide to monitor open

sockets in the operating system, and remove state of encapsulated native protocols that have their socket closed.

3.3. Receiver operation

On the receiver side (responder and initiator), the GUT service receives UDP packets, verifies the checksums, does further analysis about whether it wants to process the packet further, and either drops the packet or continues processing.

GUT must be able to receive packets with two distinct destination port ranges:

GUT_P port: this port number is seen when the packet was sent by a flow initiator.
Any other UDP port: the flow initiator has chosen a source port number and if the encapsulated IP protocol included two-way messaging (e.g. a handshake, acknowledgement packets, etc.), it will receive return packets to this UDP port.

When the host receives packets to port GUT_P, i.e., it is the destination of the flow, the responder, it MUST store the 5-tuple or 3-tuple (or any information that enables tracking a particular flow) of the encapsulated IP packet flow. This state information is needed to send back packets belonging to the same flow. An implementation may optimize the overall resource consumption of the state in GUT by omitting state information for flows where the source ports of the native- and UDP transport protocols match.

After decapsulation of the 32-bit GUT header and the UDP header, the GUT processing reconstructs the native IP packet by using the received IP header fields but exchanges the encapsulated next header and IHL fields found in the GUT header. Then the rebuilt packet is injected into the network stack for further processing. Any checksums are recalculated. Any IP options will now be visible to the network stack.

A responder should never receive fragmented IP packets since the operating system IP stack will take care of rebuilding the fragments into a full IP packet.

3.4. Example with one NAT-PT between the initiator and responder

The following figure describes how various protocol fields are mapped on a two-way IP packet flow. The example shows a DCCP-transfer going from A to B. The figure presents the content of IP packets as they

are sent out from a component on the path. Note that if the encapsulated protocol does not have port numbers, the GUT processing

is even simpler.

```
[Source, IP A]  [GUT@A] [NAT, ext IP C] [GUT@B] [Dest, IP B]

------------- Source A to destination B -------------------
1. [IP: A->B, DCCP]
2. [DCCP: E->F]

3.              [IP: A->B, UDP]
4.              [UDP: E->GUT]
5.              [GUT-hdr, DCCP]
6.              [DCCP: E->F]

7.                      [IP: C->B, UDP]
8.                      [UDP: P->GUT]
                        [GUT-hdr, DCCP]
                        ...

9.                                      [IP: C->B, DCCP]
10.                                     [DCCP: E->F]

------------- Destination B to source A -------------------
11.                                             [IP: B->C,DCCP]
12.                                             [DCCP: F->E]

13.                                     [IP: B->C, UDP]
14.                                     [UDP: GUT->P ]
15.                                     [GUT-hdr, DCCP]
16.                                     [DCCP: F->E]

17.                     [IP: B->A, UDP]
18.                     [UDP: GUT->E]
                        ...

19.             [IP: B->A, DCCP]
20.             [DCCP: F->E]
```

Figure 4: GUT encapsulation example

A few details from the figure above:

- Line 4: the GUT process takes GUT_P as the destination port, and chooses a source port based on the DCCP header.
- Line 8: the NAT may choose a new source port P, instead of E, and rewrite the UDP header.

- Line 10: before giving the packet to the local IP stack, the GUT process takes note of the source IP and port numbers, and the encapsulated protocol.
- Line 11-12: the tunneled protocol has not seen the GUT encapsulation, thus, it will use the native port numbers in the reverse traffic.
- - Lines 13-16: the GUT process has earlier stored state about the flow, knows now that the packet is for an existing stream, and can direct the flow to the right destination port "P", instead of sending it to GUT_P, as if the packet belonged to a new stream.

4. Deployment Considerations

The basic goal of GUT is to look like generic UDP messaging to any middlebox on the path. If the native transport protocol has support for congestion control, GUT encapsulated packets that are lost will trigger the native transport to react.

As GUT-encapsulated traffic looks like an ordinary stream of UDP packets, existing NAT traversal protocols and techniques work out of the box. For example, a responder GUT-daemon can, when needed, maintain the GUT_P open at the NAT using any suitable NAT-traversal protocol. The KEEPALIVE extension header in Control packets may also be used to maintain the state on middleboxes.

GUT was originally designed to be used for host-to-host communication. Yet, nothing actually prohibits to have a network node that takes the IP packets coming from a host, and tunnels them through GUT. Similarly, a network node on the receiving side of the connection can decapsulate the packets before they actually hit the receiving end-host, so essentially making a GUT-proxy service.

There is yet one critical issue to consider, namely when to encapsulate a transport protocol in GUT, and when not. GUT introduces the TEST/TEST-REPLY handshake that can be used to verify if the receiver has support for GUT; the TEST extension header could be sent at the same time as the initial packet of a native protocol. When using a TEST Control packet, the Initiator may get back an ICMP port unreachable ICMP message. This is a clear indication that GUT is not supported by the receiver.

A further option is to trigger GUT when replies to a transport protocol Y's connection initiation are not received within a given timeout. Using GUT can also be a configuration parameter, say, e.g., the host always encapsulates DCCP packets into GUT; this operation is fully transparent to the inner transport protocol.

This document does not try to define all potential cases and triggers that might result in an initiator to employ GUT for a certain flow. The actual triggers will emerge as GUT is experimented with.

5. Encapsulation of protocols without port numbers

GUT is originally designed to counter the problems of deploying relatively new transport protocols on existing Internet. Yet, GUT can also be used to encapsulate any other protocol, e.g., RSVP or HIP.

Note that some protocols may not involve port numbers, e.g., RSVP. In such cases, GUT is free to choose a random port for the initiator's port number; the responder's port is always GUT_P.

6. Security Considerations

In general GUT has the same security concerns as the IP protocol it encapsulates. For example, if the encapsulated protocol can be harmed by injecting false packets into the stream, GUT can not prohibit this.

The main additional concern GUT introduces is the increased state

needed to properly tunnel packets back-and-forth. Yet, here an implementation SHOULD analyze the encapsulated IP protocol and drop the packet, without storing state, if it does not match the expectations. For example, if the host does not have a transport port open at the indicated destination port, GUT SHOULD drop the packet silently. Also, in case the native IP packet flow does not have a notion of port numbers to enable more accurate matching of pakcets, an implementation may consider storing more information about the flow that just the 3-tuple. This has the downside that GUT must be more aware of each individual native IP protocol - currently GUT basically only needs to know if the native protocol has a notion of port numbers or not; thus, a GUT implementation only needs special treatment of native UDP, TCP, SCTP and DCCP packet flows.

Packets belonging to a GUT encapsulated flow will go through a firewall processing twice, (1) once when the IP packet arrives, either locally or from the network, and before it is given to GUT, and (2) when GUT sends out the encapsulated IP protocol inside UDP.

GUT itself does not employ any security functions for content protection. Yet, one could use any one-way mechanism, or purely rely on the security functions of the inner payload. If security measures are used on GUT, it should be a one-way scheme, which does not rely

on back-and-forth signalling; we don't want to force two-way signaling within GUT, this may or may not happen due to the inner protocol being tunneled.

The TEST/TEST-REPLY hanshake can be used for DoS attacks. Therefore, the Responder SHOULD employ rate limitation when answering TEST Control packets.

7. IANA Considerations

This document requests IANA to

1. Allocate a new UDP port number GUT_P as referred to in the document.

2. Create a GUT extension headers repository and allocate three values for TEST, TEST-REPLY and KEEPALIVE. The values are 8 bits

long.

8. Summary

Essentially this document define a generic mechanism for tunneling any IP payload over a UDP tunnel. The benefits are:

1. Existing IP protocols, with or without port information, work without changes.
2. Deployment can be done on the end-host or a network proxy.
3. No changes are required for existing NAT and firewall devices.

9. Acknowledgements

This work was partly performed and funded by Trilogy, a research project supported by the European Commission under its Seventh Framework Program (INFSOICT-216372).

The authors thank Robert Hancock for various ideas behind the generic UDP tunneling concepts.

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

10.2. Informative References

[RFC4340] Kohler, E., Handley, M., and S. Floyd, "Datagram Congestion Control Protocol (DCCP)", RFC 4340, March 2006.

[RFC4960] Stewart, R., "Stream Control Transmission Protocol", RFC 4960, September 2007.

Authors' Addresses

Jukka Manner
Aalto University
Department of Communications and Networking (Comnet)
P.O. Box 13000
FIN-00076 Aalto
Finland

Phone: +358 9 470 22481
Email: jukka.manner@tkk.fi
URI: http://www.netlab.tkk.fi/~jmanner/

Nuutti Varis
Aalto University
P.O. Box 13000
Espoo FIN-00076 Aalto
Finland

Email: nvaris@cc.hut.fi

Bob Briscoe
BT
B54/77, Adastral Park
Martlesham Heath
Ipswich IP5 3RE
UK

Phone: +44 1473 645196
Email: bob.briscoe@bt.com
URI: http://bobbriscoe.net/