Robust Header Compression
Internet-Draft
Expires: July 7, 2007

A. Surtees
M. A. West
Siemens/Roke Manor
A. B. Roach
Estacado Systems
January 3, 2007

Implementer's Guide for SigComp
draft-ietf-rohc-sigcomp-impl-guide-10

Status of this Memo

Copyright Notice

Abstract

This document describes common misinterpretations and some ambiguities in the Signaling Compression Protocol (SigComp), and offers guidance to developers to resolve any resultant problems. SigComp defines a scheme for compressing messages generated by application protocols such as the Session Initiation Protocol (SIP). This document (if approved) clarifies and corrects text in the

following updated RFCs: RFC 3320, RFC 3321, RFC 3485.

Table of Contents

1. Introduction

SigComp [1] defines the Universal Decompressor Virtual Machine (UDVM) for decompressing messages sent by a compliant compressor. SigComp further describes mechanisms to deal with state handling, message structure, and other details. While the behavior of the decompressor is specified in great detail, the behavior of the compressor is left as a choice for the implementer. During implementation and interoperability tests, some areas of SigComp that need clarification have been identified. The sections that follow enumerate the problem areas identified in the specification, and attempt to provide clarification.

Note that as this document refers to sections in several other documents the following notation is applied:

"in section 3.4" refers to section 3.4 of this document
"in section RFC-3320:3.4 refers to section 3.4 of RFC 3320 [1]

1.1. 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 RFC 2119 [5].

2. Decompression Memory Size

2.1. Bytecode within Decompression Memory Size

SigComp [1] states that the default Decompression Memory Size (DMS) is 2K. The UDVM memory size is defined in section RFC3320:7 to be (DMS - sizeof (sigcomp_msg)) for messages transported over UDP and (DMS / 2) for those transported over TCP. This means that when the message contains the bytecode (as it will for at least the first message) there will actually be two copies of the bytecode within the decompressor memory (see Figure 1). The presence of the second copy of bytecode in decompressor memory is correct in this case.

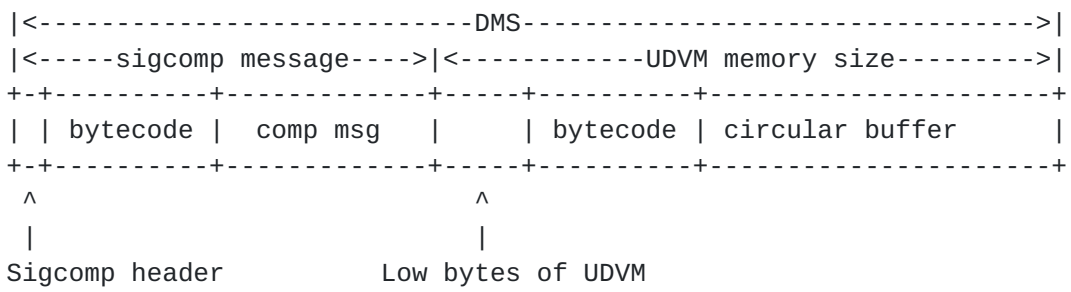


Figure 1: Bytecode and UDVM memory size within DMS

2.2. Default Decompression Memory Size

For many implementations, the length of decompression bytecode sent is in the range of three to four hundred bytes. Because SigComp specifies a default DMS of 2K, the described scheme seriously restricts the size of the circular buffer, and of the compressed message itself. In some cases, this set of circumstances has a damaging effect on the compression ratio; for others, it makes it completely impossible to send certain messages compressed.

To address this problem, those mandating the use of Sigcomp need to also provide further specification for their application that mandates the use of an appropriately sized DMS. Sizing of such a DMS should take into account (1) The size of bytecodes for algorithms likely to be employed in compressing the application messages, (2) the size of any buffers or structures necessary to execute such algorithms, (3) the size of application messages, and (4) the average entropy present within a single application message.

For example: assume a typical compression algorithm requiring approximately 400 bytes of bytecodes, plus about 2432 bytes of data structures. 400 + 2432 = 2832, which is the required UDVM memory size. For a TCP-based protocol, this means the DMS must be at least 5664 (2832 * 2) bytes, which is rounded up to 8k. For a UDP-based protocol, one must take into account the size of the SigComp messages themselves. Assuming a text-based protocol with sufficient average entropy to compress a single message by 50% (without any previous message history), and messages that are not expected to exceed 8192 bytes in size, the protocol itself will add 4096 bytes to the SigComp message size (on top of the 400 bytes of bytecodes plus a 3-byte header). 4096 + 400 + 3 = 4499. To calculate the DMS, one must add this to the required UDVM memory size: 2832 + 4499 = 6531, which is again rounded up to 8k of DMS.

3. UDVM Instructions

3.1. Data Input Instructions

When inputting data from the compressed message, the INPUT-BYTES (section RFC3320-9.4.2) and INPUT-BITS (section RFC3302-9.4.3) instructions both have the paragraph:

"If the instruction requests data that lies beyond the end of the SigComp message, no data is returned. Instead the UDVM moves program execution to the address specified by the address operand."

The intent is that if n bytes/bits are requested but only m are left in the message (where m < n) then the decompression dispatcher MUST NOT return any bytes/bits to the UDVM, and the m bytes/bits that are there MUST remain in the message unchanged.

For example, if the remaining bytes of a message are: 0x01 0x02 0x03 and the UDVM encounters an INPUT-BYTES (6, a, b) instruction. The decompressor dispatcher returns no bytes and jumps to the instruction specified by b. This contains an INPUT-BYTES (2, c, d) instruction so the decompressor dispatcher successfully returns the bytes 0x01 and 0x02.

In the case where an INPUT-BYTES instruction follows an INPUT-BITS instruction that has left a partial byte in the message, the partial byte should still be thrown away even if there are not enough bytes to input.

INPUT-BYTES (0, a, b) can be used to flush out a partial byte.

3.2. MULTILOAD

In order to make step-by-step implementation simpler, the MULTILOAD instruction is explicitly not allowed to write into any memory positions occupied by the MULTILOAD opcode or any of its parameters. Additionally, if there is any indirection of parameters, the indirection MUST be done at execution time.

Any implementation technique other than a step-by-step implementation (e.g. decode all operands then execute, which is the model of all other instructions) MUST yield the same result as a step-by-step implementation would.

For example:

```
at (64)

:location_a                    pad (2)
:location_b                    pad (2)
:location_c                    pad (2)
pad (30)
:udvm_memory_size              pad (2)
:circular_buffer               pad (2)

align (64)

MULTILOAD (location_a, 3, circular_buffer,
                udvm_memory_size, $location_a)
```

The step-by-step implementation would: write the address of circular_buffer into location_a (memory address 64); write the address of udvm_memory_size into location_a + 2 (memory address 66); write the value stored in location_a (accessed using indirection - that is now the address of circular_buffer) into location_a + 4 (memory address 68). Therefore, at the end of the execution by a correct implementation, location_c will contain the address of circular_buffer.

3.3. STATE-FREE

The STATE-FREE instruction does not check the minimum_access_length. This is correct because the state cannot be freed until the application has authenticated the message. The lack of checking does not pose a security risk because, if the sender has enough information to create authenticated messages, then sending messages that save state can push previous state out of storage anyway.

The STATE-FREE instruction can only free state in the compartment that corresponds to the message being decompressed. Attempting to free state that is either from another compartment, or that is not associated with any compartment, has no effect.

3.4. Using the stack

The instructions PUSH, POP, CALL and RETURN make use of a stack which is set up using the well known memory address stack_location to define where in memory the stack is located. Use of the stack is defined in section RFC3320-8.3, which states: '"Pushing" a value on the stack is an abbreviation for copying the value to stack[stack_fill] and then increasing stack_fill by 1.' and 'stack_fill is an abbreviation for the 2-byte word at stack_location and stack_location + 1'.

In the very rare case that the value of stack_fill is 0xFFFF when a value is pushed onto the stack, then the original stack_fill value MUST be increased by 1 to 0x0000 and written back to stack_location and stack_location + 1 (which will overwrite the value that has been pushed onto the stack).

The new value pushed onto the stack has, in theory, been written to stack [0xFFFF] = stack_location. Stack_fill would then be increased by 1; however, the value at stack_location and stack_location + 1 has just been updated. To maintain the integrity of the stack with regard to over and underflow, stack_fill cannot be re-read at this point, and the pushed value is overwritten.

4. Byte Copying Rules

Section RFC3320-8.4 states that "The string of bytes is copied in ascending order of memory address, respecting the bounds set by byte_copy_left and byte_copy_right." This is misleading in that it is perfectly legitimate to copy bytes outside of the bounds set by byte_copy_left and byte_copy_right. Byte_copy_left and byte_copy_right provide the ability to maintain a circular buffer as follows:

For moving to the right

```
if current_byte == ((byte_copy_right - 1) mod 2 ^ 16):
    next_byte = byte_copy_left
else:
    next_byte = (current_byte + 1) mod 2 ^ 16
```

which is equivalent to the algorithm given in section 8.4.

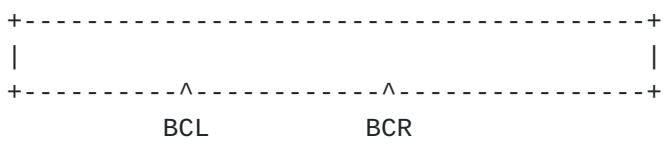
For moving to the left

```
if current_byte == byte_copy_left:
    previous_byte = (byte_copy_right - 1) mod 2 ^ 16
else:
    previous_byte = (current_byte - 1) mod 2 ^ 16
```

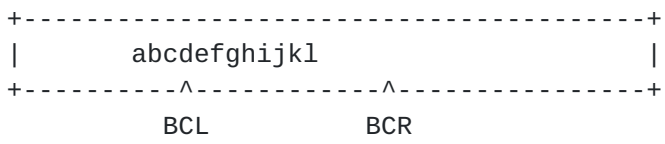
Moving to the left is only used for COPY_OFFSET.

Consequently, copying could begin to the left of byte_copy_left and continue across it (and jump back to it according to the given algorithm if necessary) and could begin at or to the right of byte_copy_right (though care must be taken to prevent decompression failure due to writing to / reading from beyond the UDVM memory).

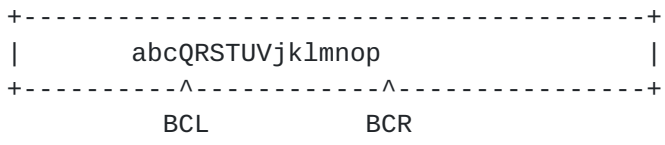
For further clarity: consider the UDVM memory laid out as follows, with byte_copy_left and byte_copy_right in the locations indicated by "BCL" and "BCR", respectively:



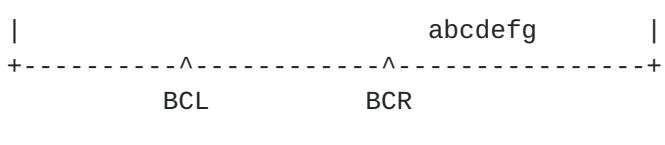
If an opcode read or wrote bytes starting to the left of byte_copy_left, it would do so in the following order:



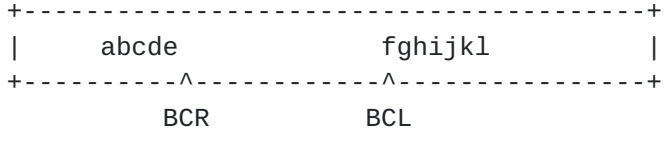
If the opcode continues to read or write until it reaches byte_copy_right, it would then wrap around to byte_copy_left and continue (letters after the wrap are capitalized for clarity):



Similarly, writing to the right of byte_copy_right is a perfectly valid operation for opcodes that honor byte copying rules:



A final, somewhat odd relic of the foregoing rules occurs when byte_copy_right is actually less than byte_copy_left. In this case, reads and writes will skip the memory between the pointers:



4.1. Instructions That Use Byte Copying Rules

This document amends the list of bytecodes that obey byte copying rules in section RFC3320-8.4 to include STATE-CREATE and CRC.

Section RFC3320-8.4 specifies the byte copying rules and includes a list of the instructions that obey them. STATE-CREATE is not in this list but END-MESSAGE is. This caused confusion due to the fact that neither instruction actually does any byte copying; rather, both instructions give information to the state-handler to create state. Logically, both instructions should have the same information about byte copying.

When state is created by the state-handler (whether the instruction was from END-MESSAGE or STATE-CREATE), the byte copying rules of section RFC3320-8.4 apply.

Note that, if the contents of the UDVM changes between the occurrence of the STATE-CREATE instruction and the state being created, the bytes that are stored are those in the buffer at the time of creation (i.e. when the message has been decompressed and authenticated).

CRC is not mentioned in section RFC3320-8.4 in the list of instructions that obey byte copying rules, but its description in section RFC3320-9.3.5 states that these rules are to be obeyed. When reading data over which to perform the CRC check, byte copying rules apply as specified in section RFC3320-8.4.

When the partial identifier for a STATE-FREE instruction is read, (during the execution of END-MESSAGE) byte copying rules as per section RFC3320-8.4 apply.

Given that reading the buffer for creating and freeing state within the END-MESSAGE instruction obeys byte copying rules, there may be some confusion as to whether reading feedback items should also obey byte copying rules. Byte copying rules do not apply for reading feedback items.

5. State Retention Priority

5.1. Priority Values

For state_retention_priority, 65535 < 0 < 1 < ... < 65534. This is slightly counter intuitive, but is correct.

5.2. Multiple State Retention Priorities

There may be confusion when the same piece of state is created at two different retention priorities. The following clarifies this:

The retention priority MUST be associated with the compartment and

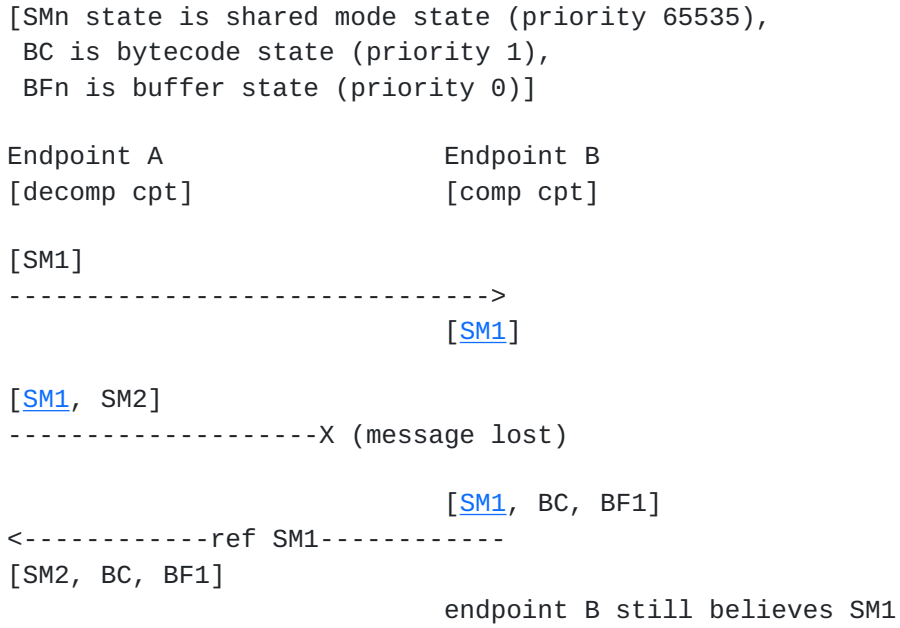
not with the piece of state. For example, if endpoint A creates a piece of state with retention priority 1 and endpoint B creates exactly the same state with retention priority 2, there should be one copy (assuming the model of state management suggested in SigComp [1]) of the actual state but each compartment should keep a record of this piece of state with its own priority. (If this does not happen then the state could be kept for longer than A anticipated or less time than B anticipated depending on which priority is used. This could cause Decompression Failure to occur.)

If the same piece of state is created within a compartment with a different priority, then one copy of it should be stored with the new priority and it MUST count only once against SMS. That is, the state creation updates the priority rather than creates a new piece of state.

5.3. Retention Priority 65535 (or -1)

There is potentially a problem with storing multiple pieces of state with the minimum retention priority (65535) as defined in SigComp [1]. This can be shown by considering the following examples, which are of shared mode which is documented in SigComp Extended [2]. The key thing about state with retention priority 65535 is that it can be created by an endpoint in the decompressor compartment without the knowledge of the remote compressor (which controls state creation in the decompressor compartment).

Example 1:



```
                                is at endpoint A

                                [BC, BF1, BF2]
     <------------ref SM1------------

     decompression failure at A
     because SM1 has already been deleted
```

Example 2:

```
     Endpoint A                 Endpoint B
     [decomp cpt]               [comp cpt]

     [SM1]
     ------------------------------->
                                [SM1]

                                [SM1, BC, BF1]
     (message lost)X------ref SM1-----

     [SM1, SM2]
     ------------------------------->
                                endpoint B does not create SM2
                                because there is no space
                                [SM1, BC, BF1]

                                [SM1, BC, BF1, BF2]
     <------------ref SM1------------
     [SM2, BC, BF2]
                                endpoint B still believes SM1
                                is at endpoint A

                                [BC, BF1, BF2, BF3]
     <------------ref SM1------------

     decompression failure at A
     because SM1 has already been deleted
```

Figure 2: Retention priority 65535 examples

Once there is more than one piece of state of minimum priority state created in a decompressor compartment, the corresponding compressor cannot be certain about which pieces of state are present in that (decompressor) compartment. If there is only one piece of state, then no such ambiguity exists.

The problem is a consequence of the different rules for the creation of minimum priority state. In particular, the creation of the second

piece of state without the knowledge of the compressor could mean that the first piece is pushed out earlier than the compressor expects (despite the fact that the state processing rules from SigComp [1] are being implemented correctly).

SigComp [1] also states that a compressor MUST be certain that all of the data needed to decompress a SigComp message is available at the receiving endpoint. Thus it SHOULD NOT reference any state unless it can be sure that the state exists. The fact that the compressor at B has no way of knowing how much state has been created at A can lead to loss of synchronization between the endpoints which is not acceptable.

One observation is that it is always safe to reference a piece of minimum priority state following receipt of the advertisement of the state.

If it is known that both endpoints are running SigComp version 2 as defined in NACK [3], then an endpoint MAY assume that the likelihood of loss of synchronization is very small and rely on the NACK mechanism for recovery.

However, for a compressor to try and avoid causing the generation of NACKs, it has to be able to make some assumptions about the behavior of the peer compressor. Also if one of the endpoints does not support NACK, then some other solution is needed.

Consequently, where NACK is not supported or for NACK averse compressors, the recommendation is that only one piece of minimum priority state SHOULD be present in a compartment at any one time. If both endpoints support NACK [3], then this recommendation MAY be relaxed, but implementers need to think carefully about the consequences of creating multiple pieces of minimum priority state. In either case, if the behavior of the application restricts the message flow, this fact could be exploited to allow safe creation of multiple minimum priority states; however, care must still be taken.

Note that if a compressor wishes the remote endpoint to be able to create a new piece of minimum priority state, it can use the STATE-FREE instruction to remove the existing piece of state.

6. Duplicate State

If a piece of state is created in a compartment in which it already exists, the time of its creation SHOULD be updated as if it had just been created, irrespective of the new state retention priority.

7. State Identifier Clashes

Section RFC3320-6.2 of SigComp [1] states that when creating a piece of state, the full 20 byte hash should be checked to see whether or not another piece of state with this identifier exists. If it does, and the state item is not identical, then the new creation MUST fail. It is stated that the probability of this occurring is vanishingly small (and so it is, see below).

However, when state is accessed, only the first n bytes of the state identifier are used, where n could be as low as 6. At this point, if there are two pieces of state with the same first n bytes of state identifier, the STATE-ACCESS instruction will cause decompression failure. The compressor referencing the state will not expect this failure mode because the state creation succeeded without a clash. At a server endpoint where there could be thousands or millions of pieces of state, how likely is this to actually happen?

Consider the birthday paradox (where there only have to be 23 people in a room to have a greater than 50% chance that two of them will have the same birthday (Birthday [8]).

The naive calculation using factorials gives:

$$Pd(N,s) = 1 - \frac{N!}{(N - s)!\ N^s}$$

where N is the number of possible values and s is the sample size.

However, due to dealing with large numbers an approximation is needed:

$$Pd(N,s) = 1 - e^{(LnFact(N) - LnFact(N-s) - s\ Ln(N))}$$

where LnFact (x) is the log of x!, which can be approximated by:

$$LnFact(x) \sim (x + 1/2)\ Ln(x) - x + Ln(2*Pi)/2 + \frac{1}{12x} - \frac{1}{360\ x^3} + \frac{1}{1260\ x^5} - \frac{1}{1680\ x^7}$$

which using N = 2^48 [6 octet partial state identifier] gives:

s = 1 000 000: Pd (N,s) = 0.018%
s = 10 000 000: Pd (N,s) = 16.28%

s = 100 000 000: Pd (N,s) = 100.00%

so when implementing, thought should be given as to whether or not 6 octets of state identifier is enough to ensure that state access will be successful (particularly at a server).

The likelihood of a clash when using the full 20 octets of state identifier, does indeed have a vanishingly small probability: using N = 2^160 [full 20 octet state identifier] gives:

s = 1 000 000: Pd (N,s) = 3.42E-35%
s = 10 000 000: Pd (N,s) = 3.42E-33%
s = 100 000 000: Pd (N,s) = 3.42E-31%

Consequently, care must be taken when deciding how many octets of state identifier to use to access state at the server.

8. Message re-ordering

SigComp [1] makes only one reference to the possibility of misordered messages. However, the statement that the 'compressor MUST ensure that the message can be decompressed using the resources available at the remote endpoint' puts the onus on the compressor to take account of the possibility of misordering occurring.

Whether misordering can occur and whether that would have an impact depends on the compartment definition and the transport protocol in use. Therefore, it is up to the implementer of the compressor to take these factors into account.

9. Requested Feedback

9.1. Feedback when SMS is zero

If an endpoint receives a request for feedback then it SHOULD return the feedback even if its SMS is zero. The storage overhead of the requested feedback is NOT part of the SMS.

9.2. Updating feedback requests

When an endpoint receives a valid message it updates the requested feedback data for that compartment. Section RFC3320-5 states that there is no need to transmit any requested feedback item more than once. However, there are cases where it would be beneficial for the feedback to be sent more than once (e.g. a retransmitted 200 OK SIP message [9] to an INVITE SIP message implies that the original 200

OK, and the feedback it carried, might not have reached the remote endpoint). Therefore, an endpoint SHOULD transmit feedback repeatedly until it receives another valid message that updates the feedback.

Section RFC3320-9.4.9 states that when requested_feedback_location equals zero, no feedback request is made. However, there is no indication of whether this means that the existing feedback data is left untouched or this means that the existing feedback data SHOULD be overwritten to be 'no feedback data'. If requested_feedback_location equals zero, the existing feedback data SHOULD be left untouched and returned in any subsequent messages as before.

Section RFC3320-9.4.9 also makes no statement about what happens to existing feedback data when requested_feedback_location does not equal zero but the Q flag indicating the presence/absence of a requested_feedback_item is zero. In this case, the existing feedback data SHOULD be overwritten to be 'no feedback data'.

10. Advertising resources

10.1. The I-bit and local state items

The I-bit in requested feedback is a mechanism by which a compressor can tell a remote endpoint that it is not going to access any local state items. By doing so, it gives the remote endpoint the option of not advertising them in subsequent messages. Setting the I-bit does not obligate the remote endpoint to cease sending advertisements.

The remote endpoint SHOULD still advertise its parameters such as DMS and state memory size (SMS). (This is particularly important; if the sender of the first message sets the I-bit, it will still want the advertisement of parameters from the receiver. If it doesn't receive these, it has to assume the default parameters which will affect compression efficiency.)

The endpoint receiving an I-bit of 1 can reclaim the memory used to store the locally available state items. However, this has NO impact on any state that has been created by the sender using END-MESSAGE or STATE-CREATE instructions.

10.2. Dynamic Update of Resources

Decompressor resources such as SMS and DMS can be dynamically updated at the compressor by use of the SMS and DMS bits in returned parameters feedback (see section RFC3320-9.4.9). Changing resources

dynamically (apart from initial advertisements for each compartment) is not expected to happen very often.

If additional resources are advertised to a compressor then it is up to the implementation at the compressor whether or not to make use of these resources. For example, if the decompressor advertises 8k SMS but the compressor only has 4k SMS then the compressor MAY choose not to use the extra 4k (e.g. in order to monitor state saved at the decompressor). In this case, there is no synchronization problem. The compressor MUST NOT use more than the most recently advertised resources. Note that the compressor SMS is unofficial (enables compressor to monitor decompressor state) and is separate from the SMS advertised by the decompressor.

Reducing the resources has potential synchronization issues and so SHOULD NOT be done unless absolutely necessary. If this is the case then the memory MUST NOT be reclaimed until the remote endpoint has acknowledged the message sent with the advertisement. If state is to be deleted to accommodate a reduction in SMS then both endpoints MUST delete it according to the state retention priority (see section RFC3320-6.2). The compressor SHOULD use up to the amount of resources most recently advertised.

10.3. Advertisement of locally available state items

Section RFC3320-3.3.3 defines locally available state items to be the pieces of state that an endpoint has available but that have not been uploaded by the SigComp message. The examples given are dictionaries and well known pieces of bytecode; and the advertisement mechanism discussed in section RFC3320-9.4.9 provides a way for the endpoint to advertise the pieces of locally available state that it has.

However, SigComp [1] does not (nor was it ever intended to) fully define the use of locally available state items, in particular, the length of time for which they will be available. The use of locally available state items is left for definition in other documents. However, this fact, coupled with the fact that SigComp does contain some hooks for uses of locally available state items and the fact that some of the definitions of such uses (in SigComp Extended [2]) are incomplete has caused some confusion. Therefore, this section clarifies the situation.

Note that any definitions of uses of locally available state items MUST NOT conflict with any other uses.

10.3.1. Basic SigComp

SigComp provides a mechanism for an endpoint to advertise locally

available state (section RFC3320-9.4.9). If the endpoint receiving the advertisement does not 'recognize' it and therefore know the properties of the state e.g. its length and lifetime, the compressor needs to consider very carefully whether or not to access the state; especially if NACK [3] is not available.

SigComp provides the following hooks for use in conjunction with locally available state items. Without further definition locally available state SHOULD NOT be used.

Section RFC3320-6.2 allows for the possibility to map locally available state items to a compartment and states that, if this is done, the state items MUST have state retention priority 65535 in order to not interfere with state created at the request of the remote compressor. Note that Section 5.3 also recommends that only one such piece of state SHOULD be created per compartment.

The I-bit in the requested_feedback_location (see section RFC3320-9.4.9) allows a compressor to indicate to the remote endpoint that it will not reference any of the previously advertised locally available state. Depending on the implementation model for state handling at the remote endpoint, this could allow the remote endpoint to reclaim the memory being used by such state items.

10.3.2. Dictionaries

The most basic use of the local state advertisement is the advertisement of a dictionary (e.g. the dictionary specified by SIP/SDP Static Dictionary [4]) or a piece of bytecode. In general, these pieces of state:

- are not mapped to compartments
- are local to the endpoint
- are available for at least the duration of the compartment
- do not have any impact on the compartment SMS

However, for a given piece of state the exact lifetime needs to be defined e.g. in public specifications such as SigComp for SIP [7] or the 3GPP IMS specification [10]. Such a specification should also indicate whether or not advertisement of the state is needed.

10.3.3. SigComp Extended Mechanisms

SigComp Extended [2] defines some uses of local state advertisements for which additional clarification is provided here.

Shared-mode (section RFC3321-5.2) is well-defined (when combined with the clarification in Section 5.3). In particular, the states that

are created and advertised are mapped into the compartment, have the minimum retention priority and persist only until they are deleted by the creation of new (non minimum retention priority) state.

The definition of endpoint initiated acknowledgments (section RFC3321-5.1.2) requires clarification in order to ensure that the definition does not preclude advertisements being used to indicate that state will be kept beyond the lifetime of the compartment (as discussed in SigComp for SIP [7]). Thus the clarification is:

Where Endpoint A requests state creation at Endpoint B, Endpoint B MAY subsequently advertise the hash of the created state item to Endpoint A. This conveys to Endpoint A (i) that the state has been successfully created within the compartment; and (ii) that the state will be available for at least the lifetime of the state as defined by the state deletion rules according to age and retention priority of SigComp [1]. If the state is available at Endpoint B after it would be deleted from the compartment according to [1], then the state no longer counts towards the SMS of the compartment. Since there is no guarantee of such state being available beyond its normally defined lifetime, endpoints SHOULD only attempt to access the state after this time where it is known that NACK [3] is available.

11. Uncompressed bytecode

It is possible to write bytecode that simply instructs the decompressor to output the entire message (effectively sending it uncompressed but within a SigComp message). This is particularly useful if the bytecode is well known (so that decompressors can recognize and output the bytes without running a VM if they wish), it is documented here.

The mnemonic code is:

```
at (0)
:udvm_memory_size         pad (2)
:cycles_per_bit           pad (2)
:sigcomp_version          pad (2)
:partial_state_id_length  pad (2)
:state_length             pad (2)
:reserved                 pad (2)
at (64)
:byte_copy_left           pad (2)
:byte_copy_right          pad (2)
:input_bit_order          pad (2)
:stack_location           pad (2)

; Simple loop
;       Read a byte
;       Output a byte
; Until there are no more bytes!

at (128)
:start
INPUT-BYTES (1, byte_copy_left, end)
OUTPUT (byte_copy_left, 1)
JUMP (start)

:end
END-MESSAGE (0,0,0,0,0,0,0)
```

which translates to give the following SigComp message:

```
0xf8, 0x00, 0xa1, 0x1c, 0x01, 0x86, 0x09, 0x22, 0x86, 0x01, 0x16,
0xf9, 0x23
```

12. RFC 3485 SIP/SDP Static Dictionary

SIP/SDP Static Dictionary [4] provides a dictionary of strings frequently used in SIP and SDP messages. The format of the dictionary is the list of strings followed by a table of offset references to the strings so that a compressor can choose to reference the address of the string or the entry in the table. Both parts of the dictionary are divided into 5 prioritized sections to allow compressors to choose how much of it they use (which is particularly useful in the case where it has to be downloaded). If only part of the dictionary is used, then the corresponding sections of both parts (strings and offset table) are used.

However, there are some minor bugs in the dictionary. In a number of places, the entry in the offset table refers to an address that is

not in the corresponding priority section in the list of strings. Consequently, if the bytecode uses the offset table and limits use of the dictionary to priorities less than 4, then care must be taken not to use the following strings in the dictionary:

'application' at 0x0334 is not at priority 2 (it's priority 4)
'sdp' at 0x064b is not at priority 2 (it's priority 4)
'send' at 0x089d is not at priority 2 (it's priority 3)
'recv' at 0x0553 is not at priority 2 (it's priority 4)
'phone' at 0x00f2 is not at priority 3 (it's priority 4)

These are seen to be relatively low cost bugs as only these 5 strings are affected and they are only affected under certain conditions.

13. Security Considerations

This document updates SigComp [1], SigComp Extended [2] and the SigComp Static Dictionary [4]. The security considerations for [2] and [4] are the same as for [1]; therefore, this section discusses only how the security considerations for [1] are affected by the updates.

Several security risks are discussed in [1]. These are discussed briefly here; however, this update does not change the security considerations of SigComp:

Snooping into state of other users - this is mitigated by using at least 48 bits from the hash. This update does not reduce the minimum and recommends use of more bits under certain circumstances.

Faking state or making unauthorized changes - this is mitigated by the fact that the application layer has to authorize state manipulation. This update does not change that mechanism.

Use of Sigcomp as a tool in a DoS attack - this is mitigated by the fact that SigComp only generates one decompressed message per incoming compressed message. That is not changed by this update.

Attacking SigComp as the DoS target by filling with state - this is mitigated by the fact that the application layer having to authorize state manipulation. This update does not change that mechanism.

Attacking the UDVM by sending it looping code - this is mitigated by the upper limit of "UDVM cycles" which is unchanged by this update.

14. IANA Considerations

This document updates SigComp [1] but does not change the version. Consequently the IANA considerations are the same as those for [1].

This document updates SigComp Extended [2] but does not change the version. Consequently the IANA considerations are the same as those for [2].

This document updates Static Dictionary [4] but does not change the version. Consequently the IANA considerations are the same as those for [4].

15. Acknowledgements

We would like to thank the following people who, largely through being foolish enough to be authors or implementors of SigComp, have provided us their confusion, suggestions, and comments:

Richard Price
Lajos Zaccomer
Timo Forsman
Tor-Erik Malen
Jan Christoffersson
Kwang Mien Chan
William Kembery
Pekka Pessi

16. References

16.1. Normative References

[1] Price, R., Borman, C., Christoffersson, J., Hannu, H., Liu, Z., and J. Rosenberg, "Signaling Compression (SigComp)", RFC 3320, January 2003.

[2] Hannu, H., Christoffersson, J., Forsgren, S., Leung, K., Liu, Z., and R. Price, "Signaling Compression (SigComp) - Extended Operations", RFC 3321, January 2003.

[3] Roach, A., "A Negative Acknowledgement Mechanism for Signaling Compression)", RFC 4077, October 2004.

[4] Garcia-Martin, M., Borman, C., Ott, J., Price, R., and A. Roach, "The Session Initiation Protocol (SIP) and Session Description Protocol (SDP) Static Dictionary for Signaling Compression (SigComp)", RFC 3485, February 2003.

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

16.2. Informative References

[6] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications (ABNF)", RFC 2234, November 1997.

[7] Borman, C., Liu, Z., Price, R., and G. Camarillo, "Applying Signaling Compression (SigComp) to the Session Initiation Protocol (SIP)", draft-ietf-rohc-sigcomp-sip-04.txt , November 2006.

[8] Ritter, T., "Estimating Population from Repetitions in Accumulated Random Samples", 1994, <http://www.ciphersbyritter.com/ARTS/BIRTHDAY.HTM>.

[9] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, June 2002.

[10] "IP Multimedia Call Control Protocol based on Session Initiation Protocol (SIP)", October 2006.

Appendix A. Dummy Application Protocol (DAP)

A.1. Introduction

This appendix defines a simple dummy application protocol (DAP) that can be used for SigComp interoperability testing. This is handy for SigComp implementations that are not integrated with SIP stack. It also provides some features that facilitate the tests of SigComp internal operations.

The message format is quite simple. Each message consists of a 8-line message-header, an empty line, and an OPTIONAL message-body. You can see the style resembles that of SIP and HTTP.

The exact message format is given later in augmented Backus-Naur Form (ABNF) [6]. Here are a few notes:

Each line of message-header MUST be terminated with CR LF.

The empty line MUST be present even if the message-body is not.

Body-length is the length of the message-body, excluding the CRLF which separates the message-body from the message-header.

All strings in message-header are case-insensitive.

For implementation according to this appendix, the DAP-version MUST be set to 1.

A.2. Processing a DAP message

A message with invalid format will be discarded by a DAP receiver

For testing purpose, a message with valid format will be returned to the original sender (IP address, port number) in clear text, i.e., without compression. This is the case even if the sender requests this receiver to reject the message. Note that the entire DAP message (message-header + CRLF + message-body) is returned. This allows the sender to compare what it sent with what the receiver decompressed.

Endpoint-ID is the global identifier of the sending endpoint. It can be used to test the case where multiple SigComp endpoints communicate with the same remote SigComp endpoint. For simplicity, IPv4 address is used for this purpose.

Compartment-ID is the identifier of the *compressor* compartment that the *sending* endpoint used to compress this message. It is assigned by the sender and therefore only unique per sending endpoint. I.e., DAP messages sent by different endpoints MAY carry same compartment-ID. Therefore, the receiver SHOULD use the (endpoint-ID, compartment-ID) pair carried in a message to determine the decompressor compartment identifier for that message. The exact local representation of the derived compartment identifier is an implementation choice.

To test SigComp feedback [1], peer compartments between two endpoints are defined in DAP as those with the same compartment-ID. For example, (endpoint-A, 1) and (endpoint-B, 1) are peer compartments. That means, SigComp feedback for a DAP message sent from compartment 1 of endpoint-A to endpoint-B will be piggybacked on a DAP message sent from compartment 1 of endpoint-B to endpoint-A.

A DAP receiver will follow the instruction carried in header line-5 to either accept or reject a DAP message. Note: line-6 and line-7 will be ignored if the message is rejected.

A DAP receiver will follow the instruction in line-6 to create or close the decompressor compartment that is associated with the received DAP message (see above).

If the header line-7 of a received DAP message carries "TRUE", the

receiver will send back a response message to the sender. This allows the test of SigComp feedback. As mentioned above, the response message MUST be compressed by, and sent from, the local compressor compartment that is peer of the remote compressor compartment. Other than this constraint, the response message is just a regular DAP message that can carry arbitrary message-header and message-body. For example, the "need-response" field of the response can also be set to TRUE, which will trigger a response to response, and so on. Note that since either endpoint has control over the "need-response" field of its own messages, this does not lead to a dead loop. A sensible implementation of a DAP sender SHOULD NOT blindly set this field to TRUE unless a response is desired. For testing, the message-body of a response MAY contain the message-header of the original message that triggered the response.

Message-seq can be used by a DAP sender to track each message it sends, e.g. in case of losses. Message loss can happen either on the path or at the receiving endpoint (i.e. due to decompression failure). The assignment of message-seq is up to the sender. For example, it could be either assigned per compartment or per endpoint. This has no impact on the receiving side.

A.3. DAP message format in ABNF

(Note: see (ABNF) [6] for basic rules.)

```
DAP-message = message-header CRLF [ message-body ]

message-body = *OCTET

message-header = line-1 line-2 line-3 line-4 line-5 line-6 line-7
line-8

line-1 = "DAP-version" ":" 1*DIGIT CRLF
line-2 = "endpoint-ID" ":" IPv4address CRLF
line-3 = "compartment-ID" ":" 1*DIGIT CRLF
line-4 = "message-seq" ":" 1*DIGIT CRLF
line-5 = "message-auth" ":" ( "ACCEPT" / "REJECT" ) CRLF
line-6 = "compartment-op" ":" ( "CREATE" / "CLOSE" / "NONE" ) CRLF
line-7 = "need-response" ":" ( "TRUE" / "FALSE" )
line-8 = "body-length" ":" 1*DIGIT CRLF

IPv4address = 1*3DIGIT "." 1*3DIGIT "." 1*3DIGIT "." 1*3DIGIT
```

A.4. An example of a DAP message

```
DAP-version: 1
endpoint-ID: 123.45.67.89
compartment-ID: 2
message-seq: 0
```

```
      message-auth: ACCEPT
      compartment-op: CREATE
      need-response: TRUE
      body-length: 228

   This is a DAP message sent from SigComp endpoint at IP address
   123.45.67.89.  This is the first message sent from compartment 2.
   Please accept the message, create the associated compartment, and
   send back a response message.
```

Authors' Addresses

Abigail Surtees
Siemens/Roke Manor
Roke Manor Research Ltd.
Romsey, Hants SO51 0ZN
UK

Phone: +44 (0)1794 833131
Email: abigail.surtees@roke.co.uk
URI: http://www.roke.co.uk

Mark A. West
Siemens/Roke Manor
Roke Manor Research Ltd.
Romsey, Hants SO51 0ZN
UK

Phone: +44 (0)1794 833311
Email: mark.a.west@roke.co.uk
URI: http://www.roke.co.uk

Adam Roach
Estacado Systems
17210 Campbell Rd.
Suite 250
Dallas, TX 75252
US

Phone: sip:adam@estacado.net
Email: adam@estacado.net

Intellectual Property Statement

Disclaimer of Validity

Copyright Statement

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.