INTERNET-DRAFT
Intended Status: Informational
Expires: December 31, 2018

Hudson Ayers
Paul Crews
Hubert Teo
Conor McAvity
Amit Levy
Philip Levis
Stanford University

June 29, 2018

Design Considerations For Low Power Internet Protocols
draft-hudson-ayers-00

Abstract

This document discusses guidelines for specifying low-power Internet protocols in order to improve implementation interoperability. These guidelines are based around the importance of balancing memory usage and energy efficiency, and the importance of not relying on Postel's law when dealing with low resource devices. This document applies these guidelines to the IPv6 over low-power wireless personal area networks (6LoWPAN) Internet Standard, suggesting changes that would make it more likely for implementations to interoperate.

Status of this Memo

Copyright and License Notice

Table of Contents

1 Introduction

Interoperability is critical for the Internet. Not only do edge-devices need to interoperate with the broader Internet, they should also interoperate with other devices in the same network. Historically, though, embedded systems and sensor networks have been vertical silos of proprietary technologies, each using custom network protocols and homogeneous implementations. Networks typically require specialized gateways and cannot easily include devices from different vendors for a variety of applications.

Interoperability is just as important in the Internet of Things, but end hosts in the Internet of Things are less resourceful, more diverse in capability, and less well audited than typical Internet end hosts. Using IP allows devices from different manufacturers, running completely different software stacks, to interoperate, share services, and compose into larger, more complex applications. This interoperability should exist not only between IoT devices communicating with hosts across the broader Internet, but also between IoT devices in the same low power wireless network. The presence of such interoperability precludes the need for multiple gateways to support different devices, simplifies network management, and allows for efficient, logical communication between nearby devices.

To address this problem, the IETF published a series of RFCs detailing a standard format for transmitting IPv6 packets over low-power wireless link layers such as IEEE 802.15.4 {RFC 4919][RFC 4944][RFC 6282][RFC 6775]. The 6LoWPAN RFCs define a fragmentation format, a compression format, and more. These 6LoWPAN standards have been adopted by a number of popular embedded operating systems, including Contiki {CONTIKI], RiotOS {RIOT], OpenThread [OPENTHREAD], mbedOS [ARM], and TinyOS [TINYOS].

Unfortunately, none of these implementations are complete. Each implementation supports different subsets of 6LoWPAN. As a result, devices built using different embedded operating systems cannot interoperate. In fact, for every possible pairing, one implementation is likely to transmit 6LoWPAN packets which the other cannot process.

This paper explores the reasons behind the lack of interoperability in practice, and argues that this results from the protocol too heavily prioritizing radio efficiency over processor resources, and failing to consider the broad range of devices which embedded operating systems will attempt to support. This document proposes four guidelines for designing interoperable protocols for low-power wireless networks, and explain them through an example application to two 6LoWPAN standards - RFC 4944 and RFC 6282. These guidelines are

informed by an empirical analysis of existing 6LoWPAN implementations as well as experience implementing a full 6LoWPAN stack for the Tock operating system [TOCK].

1.1 Terminology

Readers are expected to be familiar with all terms and concepts discussed in "IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals", "Transmission of IPv6 Packets over IEEE 802.15.4 Networks", "Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks".

Readers would benefit from reading 6LoWPAN Neighbor Discovery (ND), 6LoWPAN routing requirements, and 6LoWPAN design/application spaces for additional details of 6LoWPAN work.

2. 6LoWPAN Interoperability Study

The IETF's 6LoWPAN working group has been concerned with interoperability between implementation since inception [6LO-CHART]. Indeed, members of the working group have organized ``Plugtests'', where vendors verified correct implementation of the 6LowPAN specifications and tested interoperability with other vendors. Unfortunately, detailed results of these plugtests are not publicly available.

A node sending IPv6 packets using 6LoWPAN may fragment the packet or compress headers in a large number of ways permitted by the specification. These choices depend both on the properties of the packet (e.g. whether it is a UDP packet or if the origin and destinations are in the same subnet) as well as on which compression and fragmentation options the sender chooses to use. For two nodes to interoperate, the 6LoWPAN implementation on each node must be able to receive any packet the other node might send.

This document investigates the interoperability of 6LoWPAN implementations from five common embedded software platforms: Riot OS, Contiki, OpenThread, TinyOS, and ARM Mbed. This study is concerned, specifically, with each implementation's ability to receive and decode 6LoWPAN packets sent from other implementations. This document does not explore whether devices built using these implementations could form a network in the first place, since 6LoWPAN leaves much of the network formation process (e.g. discovery and joining) unspecified.

The compatibility analysis in this document, between five common

embedded software frameworks with 6LoWPAN implementations, is based in three discoveries. First, a determination of how completely each implementation implements the 6LoWPAN specification was obtained by directly examining their source code. Second, this code analysis was extended to verify that under some circumstances, each implementation sends packets using compression or fragmentation options which another implementation cannot decode. Finally, it was discovered that even in cases where two implementations use compatible compression and fragmentation options, different implementation choices, such as header decompression bounds, limit their interoperability.

As a result, no pairing of these five implementations is fully interoperable.

2.1 Incomplete Implementations

The 6LoWPAN protocol consists of a large number of complex and mostly independent features which use the link-layer frame efficiently via compression and fragmentation optimizations. Examining the code and documentation for each of the aforementioned 6LoWPAN stacks reveals that the stacks do not uniformly implement the specification. In fact, each specification implements a different subset of the requirements in the 6LoWPAN specification, and none implements the entire specification to the letter. A visualization of the mismatched feature support across different 6LoWPAN implementations can be found in Table 1. Note that OT is an abbreviation used throughout this document to refer to OpenThread.

Feature	Stack				
	Contiki	OT	Riot	Arm	TinyOS
Uncompressed IPv6	o		o	o	o
6LoWPAN Fragmentation	o	o	o	o	o
1280 byte packets	o	o	o	o	o
Dispatch_IPHC header prefix	o	o	o	o	o
IPv6 Stateless Address Compression	o	o	o	o	o
Stateless multicast address compression	o	o	o	o	o
802.15.4 16 bit short address support		o	o	o	o
IPv6 Address Autoconfiguration	o	o	o	o	o
IPv6 Stateful Address Compression	o	o	o	o	o
Stateful multicast address compression		o	o	o	
IPv6 TC and Flow label compression	o	o	o	o	o
IPv6 NH Compression: Tunneled IPv6		o		o	o
IPv6 NH Compression: UDP	o	o	o	o	o
UDP port compression	o	o	o	o	o
UDP checksum elision					o

Compression + headers past first frag					

Table 1: 6LoWPAN Interoperability Matrix

2.2 Unrealistic Bounds

Beyond the variation in what portions of the 6LoWPAN specification each stack implements, there also exists significant variation in how each stack handles certain implementation-specific details. Some of these details have little impact on interoperability, such as decisions regarding how many fragments a stack holds for a given packet before dropping all of them, whether to allow for reconstruction of multiple packets simultaneously, and how long to hold onto fragments for which the rest of the packet has not yet arrived. Other details, however, differ in ways that significantly affect interoperability between stacks. A discussion of two such details follows:

2.2.1 Maximum Header Decompression

Each of the stacks analyzed imposes some limit on the maximum amount of header decompression possible for a received packet. Such a limit is necessary to ensure that packet and fragment buffers within a stack are large enough for received packets. The maximum amount of header decompression allowed by the 6LoWPAN specification is about 1200 bytes, basically, if an entire MSS IPv6 packet was sent containing only compressed headers. Some of the stacks analyzed decompress fragments directly into the MSS buffer which will eventually contain the entire IPv6 packet, and thus support this bound. Other stacks impose significantly lower limits - limits low enough that packets could easily be constructed within the 6LoWPAN specification that would exceed these limits. For example, Contiki's limit of 38 bytes of header decompression is exceeded by any packet for which the IP header is maximally compressed (38 bytes) and the UDP header is compressed at all. Accordingly, certain stacks would send packets with a significant amount of header compression, but other stacks would silently drop these packets due to lacking buffer space for fragments requiring that much decompression. Furthermore, these stacks do not given any indication back to the sender that a packet has been dropped for this reason, making it difficult for the sending stack to identify how to adjust its transmission to successfully deliver data.

2.2.2 Arbitrary Next Header Compression

Several of the 6LoWPAN stacks also impose limits on the arbitrary

compression/decompression of IPv6 extension headers and next headers required by the specification. The headers which must be handled are as follows:

- IPv6 Hop-By-Hop Options Header

- IPv6 Routing Header

- IPv6 Fragment Header

- IPv6 Destination Options Header

- IPv6 Mobility Header

- IPv6 Next Header

- UDP Next Header

Further, 6LoWPAN implementations are expected to be able to decompress at least one of each of these headers, and up to two Destination Options headers, in almost any order. Handling all of these possible cases can result in complex state machines, convoluted code, and increase in code size and RAM use. Therefore, several of the stacks examined impose a limit on this arbitrary next header decompression - namely, Contiki and Riot. Both of these stacks only check for the UDP Next Header. This greatly simplifies the code required for decompression of next headers in these stacks as compared to the others, which require recursion to handle this arbitrary compression. The offshoot of this simplified code, however, is that these stacks will drop packets with certain compressed extension header configurations when other stacks send such messages.

2.3 No Pairing Interoperates

These interoperability concerns are more than theoretical: existing 6LoWPAN stacks generate valid packets that other stacks discard. This proves that missing receive functionality is not simply a case of limited 6LoWPAN stacks abstaining from handling packets which no existing stacks ever generate.

What follows is a listing of each of the 10 possible combinations of 6LoWPAN stacks, accompanied by a single example packet which can be generated by one of the stacks in the pairing which the other stack would not receive.

Contiki, OpenThread : Contiki generated message using uncompressed IPv6

Contiki, Riot: Riot generated message using stateful multicast address compression

Contiki, Mbed: Mbed generated message using compresesd, tunneled IPv6

Contiki, TinyOS: TinyOS generated message containing compressed IPv6 extension headers

OpenThread, Riot: OpenThread generated message containing any of the IPv6 extension headers, which the OpenThread stack automatically compresses

OpenThread, Mbed: Mbed generated IPv6 packet containing the IPv6 mobility header

OpenThread, TinyOS: OpenThread generated message for which the destination address is compressed using stateful multicast compression

Riot, Mbed: Mbed generated IPv6 message containing any compressed next header other than the UDP header

Riot, TinyOS: Riot generated message for which the destination address is compressed using stateful multicast compression

Mbed, TinyOS: Mbed generated Neighbor Discovery message using the 6LoWPAN context option as specified in RFC 6775.

This is a non-exhaustive listing, and for most of these pairings several message formats exist which could be generated by one that would be dropped by the other. Each instance for which a claim is made that packets could be easily generated has been verified via code analysis.

In addition to this code analysis, tests were performed to present further evidence that several of these packets formats could easily be generated via typical use of these 6LoWPAN stacks.

These tests involved slightly modifying basic example networking apps on each stack, such that the existing 6LoWPAN interface could be used to send certain packets. These modified examples were flashed onto embedded hardware platforms supported by each. The transmitted packets were captured using a wireless packet sniffer, and the sniffed packets analyzed using Wireshark. This exercise verified that these non-interoperable packets could in fact be sent. Further description of this hardware generation of select packets can be found in [DESIGN].

3 Implementation Concerns

The 6LoWPAN specification was created with a clear goal---to allow for IPv6 connectivity over a link-layer with an order of magnitude smaller frame sizes than Ethernet. Unfortunately, fragmented IPv6 on its own requires header overhead much greater than typical wireless protocols designed for low power devices. As a result, the specification places an extreme focus on minimizing protocol overhead and, thus, radio utilization.

The primary problem with 6LoWPAN is that this focus was taken too far. This focus has resulted in complex implementations that require significant processor resources. In order for devices to interoperate, they must be able to parse any valid received 6LoWPAN packet that might be sent by others.

In practice, many 6LoWPAN implementations do not implement the entire specification and, therefore, are not interoperable. This is not a result of poor software design, but rather intentional choices to implement different subsets of the specification that favor limited RAM and code size, security concerns, and minimizing engineering effort.

In fact, in some cases even these incomplete 6LoWPAN implementations systems are too resource intensive for some devices. As a result, several implementations allow the developer to remove portions of the 6LoWPAN stack during compilation. Even when implementations use overlapping portions of the specification, additional interoperability conflicts arise from different choices of memory bounds for decompression.

3.1 Processor Resources

Evidence that developers of these 6LoWPAN stacks were concerned about 6LoWPAN's consumption of processor resources is baked into the design of each. One of the primary indicators that each implementation was concerned with code size is the prevalence of options to compile limited subsets of the 6LoWPAN stack. For example, Contiki defines the SICSLOWPAN_CONF_COMPRESSION compilation flag, which can be set to force all Contiki packets (sent and received!) to be processed as uncompressed IPv6. Riot presents extensive compilation options for 6LoWPAN, allowing for the exclusion of all IPHC compression, the exclusion of context based compression alone, the exclusion of fragmentation, the exclusion of ND, and the exclusion of next header compression. The Mbed stack allows users to exclude elements of the IPv6 stack such as security features, routing specific features, link-layer features, and more. Further, Mbed defines macros which can be used to save RAM at the expense of flash, or vice-versa. TinyOS by

default removes all code in a stack that is not being used by an application, and this can easily be observed by compiling different 6LoWPAN application binaries.

Table 2 shows the code size overhead of each of the five implementations broken into independent overheads for compression, fragmentation, mesh and broadcast headers, as well as totals for 6LoWPAN and the entire networking stack including physical layer drivers, IPv6, UDP, ICMP, etc.

Stack	Code Size Measurements (Bytes)				
	IP-All	6Lo-All	Compression	Frag	Mesh/Bcast Hdr
Contiki	37538	11262	5952	3319	N/A
OT	42262	26375	4146-20000	1310	4500
Riot	30942	7500	>4712	1514	N/A
Arm Mbed	46030	22061	17900	3104	1331
TinyOS	37312	16174	----	----	600

Table 2: 6LoWPAN Stack Code Size

The methodology use to collect these values can be found in [DESIGN]. These results likely overestimate the overhead of fragmentation and underestimate the overhead of certain kinds of compression since some of the complexity of compression is born on the fragmentation logic. Moreover, for OpenThread and Arm Mbed, which required manual examination of binaries, the results almost certainly underestimate the overhead of all 6LoWPAN components since we only counted procedures which unambiguously implemented particular functionality, though some of the complexity is implemented in other portions of the stack. In summary:

- 6LoWPAN stack developers were concerned with processor resource requirements of the protocol.

- Fragmentation, the only portion of 6LoWPAN that's strictly necessary for sending IPv6 packets, consumes significantly less ROM than compression.

- Implementations with more complete adherence to compression specification consume more code for compression

- Mesh and broadcast headers are relatively expensive given that few real-world applications use them

5 Contributing Factors

Several fundamental factors contributed to 6LoWPAN's interoperability problems.

When writing low power networking specifications, an important "slider'' exists - the tradeoff between code size and protocol efficiency. This tradeoff is similar to the historically significant tradeoff between RAM and code size. Techniques such as advanced MAC and physical layers, and tracking the state of a network can reduce packet sizes and, thus, radio energy consumption. However, these techniques typically require larger and more complex implementations.

Even moving beyond the constraints of code size, added complexity harms interoperability in the general case, and complex implementations are undesirable in the space of low power embedded devices. Finally, 6LoWPAN failed to consider the reality that some implementations of the protocol may be incomplete, and accordingly failed to include any affirmative indications of interoperability failures, with interoperability failures instead only being visible as silent packet drops. All of these factors contributed to 6LoWPAN's interoperability problems, and inspire the guidelines that follow.

6 Design Guidelines

This section describes four protocol design guidelines which, if followed, lead to low-power protocols that are more likely to have interoperable implementations. In the next section, these are further explained by showing how each can be applied to 6LoWPAN.

6.1 Guideline 1: Capability Spectrum

A low power protocol should be implementable on devices which are at the low end of code and RAM resources. Rather than require every device pay the potential energy costs of fewer optimizations, a protocol should support a spectrum of device capabilities. This spectrum defines a clear ordering via which especially resource constrained devices can reduce code size or RAM use by eliding features. Such a spectrum makes a protocol usable by extremely low resource devices without forcing more resourceful devices to communicate inefficiently.

This capability spectrum should be a linear scale. For a device to support capability level N, it must also support all lower capability levels. More complex configuration approaches (e.g., a set of independent options) would allow for a particular application or implementation to be more efficient, picking the features that give

the most benefit at the least complexity cost. However, this sort of optimization then makes interoperability more difficult, as two devices must negotiate which features to use.

6.1.1 Guideline 1 Application to 6LoWPAN

Application of this guideline would require replacing the large collection of "MUST'' requirements - those "features'' in Table 1 - into 6 levels of functionality. These levels prioritize features that provide the best packet size savings given the resulting implementation complexity. For example, the greatest savings results from compressing 128-bit IPv6 addresses.

0. Uncompressed IPv6

    0a. Uncompressed IPv6

    0b. 6LoWPAN Fragmentation and the Fragment Header

    0c.1280 Byte Packets

1. IPv6 Compression Basics + Stateless Address Compression

    1a. Support for the Dispatch_IPHC Header Prefix

    1b. Correctly handle elision of IPv6 length and version

    1c. Stateless compression of unicast addresses

    1d. Stateless compression of multicast addresses

    1e. Compression even when 16 bit addresses are used at the link layer

    1f. IPv6 address autoconfiguration

2. Stateful IPv6 Address Compression

    2a. Stateful compression of unicast addresses

    2b. Stateful compression of multicast addresses

3. IPv6 Traffic Class and Flow Label Compression

    3a. Traffic Class compression   3b. Flow Label Compression   3c. Hop Limit Compression

4. IPv6 and UDP NH Compression + UDP Port Compression

4a. Handle Tunneled IPv6 correctly

4b. Handle the compression of the UDP Next Header

4c. Correctly handle elision of the UDP length field

4d. Correctly handle the compression of UDP ports

4e. Correctly handle messages for which headers go on longer than the first fragment, and the headers in the first fragment are compressed.

5. Entire Specification

5a. Support the broadcast header and the mesh header as described in RFC 4944

5b Support compression of all IPv6 Extension headers

The classes in this scale do not precisely reflect the current feature support of the implementations described above. For example, Contiki supports UDP port compression (level 5) but does not support 802.15.4 short addresses (level 2) or tunneled IPv6 (level 5): following this formulation, Contiki only provides level 1 support. If Contiki supported 16-bit addresses, it would provide level 4 support.

The specific spectrum presented here is based off of measurements of code size, the saved bits that each additional level of compression allows for, and observations of existing 6LoWPAN implementations.

6.2 Guideline 2: Capability Discovery

The second guideline immediately follows from the first: if two implementations may have different capability levels, there should be an explicit mechanism by which two devices can efficiently discover what level to use when they communicate

If two devices wish to communicate, they default to the lower of their supported capability levels. For example, suppose a TinyOS device supports level 2 and a Contiki device supports level 4; Contiki must operate at level 2 when communicating with the TinyOS device. This requires keeping only a few bits of state for any device to communicate with. Also, note that this state is per-hop; for a layer 3 protocol like IP, it is stored for link-layer neighbors (not IP endpoints) and so does not require knowledge of the whole network.

6.2.1 Guideline 2 Application to 6LoWPAN

6lowpan could implement capability discovery using two mechanisms: neighbor discovery (ND) and ICMP. Neighbor discovery allows devices to probe and determine capability levels, while ICMP allows devices to determine when incompatible features are used, or when ND is not available.

Neighbor discovery: 6LoWPAN ND should add an option that allows a device to communicate its capability class during association with a network. The inclusion of a few extra bits in ND messages would allow all devices that learn neighbor addresses via ND to also know how to send packets which that neighbor can receive. This option minimizes the energy cost of communicating capabilities. It is worth noting that [RFC7400] already employs a similar method for communicating whether devices implement General Header Compression: adding such an option is clearly viable.

ICMP: All IPv6 devices are already required to support ICMP. A new ICMPv6 message type - 6LoWPAN Class Unsupported - should be added, which could be sent in response to messages received encoded using a 6LoWPAN class higher than the class of the receiving host. This would allow for communication of capabilities even in networks not constructed using IPv6 ND. This ICMPv6 message would allow hosts to indicate exactly what class the receiving host does support, preventing any need for repeated retransmissions using different compression or fragmentation formats.

6.3 Guideline 3: Provide Reasonable Bounds

Specifications should impose reasonable bounds on recursive or variable features so implementations can bound RAM use. These bounds have two benefits. First, it allows implementations to safely limit their RAM use without silent interoperability failures. E.g., today, if an mbed device sends a 6lowpan packet whose compression is greater than 38 bytes to a Contiki device, Contiki will silently drop the packet. Second, it ensures that capability discovery is sufficient to interoperate.

The original designers of a specification may not know exactly what these values should be. This is not a new problem: TCP congestion control, for example, had to specify initial congestion window values. The bounds should initially be very conservative. Over time, if increasing resources or knowledge suggests they should grow, then future devices will have the onus of using fewer resources to interoperate with earlier ones.

6.3.1 Guideline 3 Application to 6LoWPAN

Section 2 discussed two unreasonable bounds which affect 6LoWPAN

interoperability. The first is the 1280 byte bound on maximum header decompression (the amount a header will grow when decompressed). A bound allows implementations to conserve RAM. As a result, some implementations impose their own lower bounds, but these bounds do not agree so some stacks cannot decompress some packets sent by other stacks. The lack of a bound on arbitrary next header compression was demonstrated as adding significant complexity to implementations to service packets which should rarely be used.

To address this, maximum header decompression in 6LoWPAN packets should be bounded to 50 bytes. This bound allows for significant RAM savings in implementations that decompress first fragments into the same buffer in which the fragment was originally held prior to any copying into a 1280 byte buffer.

Second, the requirement for compression of interior headers for tunneled IPv6 should be removed. Currently, section 4.2 of RFC 6282 states "When the identified next header is an IPv6 Header...The following bytes MUST be encoded using LOWPAN_IPHC''. This is problematic because it places no bound on how many tunneled IPv6 headers may need to be compressed or decompressed, creating locations in code that require unbounded amounts of recursion. Implementations should adjust their path MTU constraints and responses to support inserting source routing headers, rather than tunnel IPv6.

This change would limit the complexity of arbitrary next header compression slightly. In addition, an ordering should be imposed on the order of IPv6 extension options if they are to be compressed. This would allow for implementations to avoid recursive functions to decompress these headers, and instead use simple if/else statements. If for some reason IPv6 extension headers must be placed in a different order for a particular packet, those options must be sent uncompressed.

6.4 Guideline 4: Don't Break Layering

Designers should ensure that interoperability is a central priority for specifications throughout the design process, and that interoperability is not simply assumed from the fact that devices will be communicating via a shared protocol. In particular, specifications should be careful that considerations introduced to save energy in certain scenarios should not make assumptions about the rest of the stack. Layering is a foundational network design principle. As the difficulty NATs introduced to Internet connectivity in the early 2000s demonstrated, breaking layering can introduce unforeseen and extremely difficult to fix interoperability problems.

The appeal of cross-layer optimization in embedded systems is even

stronger than in traditional computers. Designed for a specific application, a developer can understand and know exactly how the entire system works, from hardware to application code. However, while this whole-system knowledge makes sense for a particular device or iteration of an application, long-lived systems will evolve and change. This is especially true if the device will need to interoperate with new gateways or application devices. Furthermore, as embedded systems have grown more complex, their software has begun to resemble more traditional systems. Rather than write software from scratch every time, systems use and draw on existing operating systems as well as libraries. By breaking layering, cross-layer optimizations require that developers own and customize the entire software stack.

6.4.1 Guideline 4 Application to 6LoWPAN

UDP checksum compression, as defined in section 4.3.2 of RFC 6282, should be removed from the 6LoWPAN specification. The RFC says that a higher layer may request the checksum be elided if it has an integrity mechanism that covers the UDP header. At first glance, this seems sufficient: if the UDP header is covered by a message integrity code (MIC) or other checksum, then corrupted packets will be correctly dropped.

However, it misses an important error case: if the UDP ports are corrupted, then a packet missing a checksum may be delivered to the wrong application, and this incorrect application may not impose a replacement integrity measure or know one exists. It therefore cannot verify the MIC. Furthermore, protecting the header with a link-layer MIC is insufficient, as it only protects packets against sub-link corruption.

The end-to-end principle [E2E], foundational to all modern network design, says that only endpoints can verify correct communication. The only place that can safely verify the UDP header is the UDP stack. It is worth noting that the seminal example that led to definition of the end-to-end principle was a memory corruption: packets held in memory to be sent were corrupted before being sent. The recommended workarounds in RFC 6282 are vulnerable to such an event. A packet sent by an application that elides the UDP checksum could be corrupted in memory before the link-layer MIC is computed. Such a packet would be successfully received by the destination and dispatched to the wrong application, which would not check the application-level MIC.

The payoff of UDP checksum compression is not even significant - 2 bytes of checksum is a small portion of a 127 byte frame. The problematic nature of UDP checksum compression is further

demonstrated by the fact that only one of the five stacks analyzed in this document implements the feature.

7 Security Considerations

This informational document does have some implications for security if followed.

First, capability advertisements of the type recommended in this document are liable to leak some information regarding the type of device sending those advertisements. In any situation for which this information is priveleged, such advertisements must be suppressed.

Second, implementations should be careful not to take for granted that the suggestions in this document will be implemented by all other transmitting devices. Accordingly, though this document recommends reasonable bounds, receivers still must be careful to prevent buffer overflows in the event these bounds are not followed.

Finally, it is worth noting that breaking layering has clear security implications, and that the recommendation in this document to avoid this practice should be expected to improve security by allowing the security protocols in place at individual layers to work as intended.

8 IANA Considerations

This is an informational document, and accordingly does not formally request any IANA changes. However, it is worth noting that the example application of the guidelines to 6LoWPAN would require some changes by IANA, if actually implemented.

Namely, IANA would be requested to update some of the "6LoWPAN Capability Bits" under the "Internet Control Message Protocol Version 6 (ICMPv6) Parameters" registry such that some of the unassigned bits could be repurposed for capability advertisements as described in this document.

Additionally, IANA would be requested to update the "IPv6 Neighbor Discovery Option Formats" registry to include a new ND option format for capability advertisements [RFC4861].

9 References

9.1 Normative References

[RFC4919] Kushalnagar, N., Montenegro, G., and C. Schumacher, "IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals", RFC

4919, DOI 10.17487/RFC4919, August 2007, <https://www.rfc-editor.org/info/rfc4919>.

[RFC4944] Montenegro, G., Kushalnagar, N., Hui, J., and D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks", RFC 4944, September 2007, <http://www.rfc-editor.org/info/rfc4944>.

[RFC6282] Hui, J. and P. Thubert, "Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks", RFC 6282, September 2011, <http://www.rfc-editor.org/info/rfc6282>.

[RFC6775] Shelby, Z., Chakrabarti, S., Nordmark, E., and C. Bormann, "Neighbor Discovery Optimization for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs)", RFC 6775, November 2012, <http://www.rfc-editor.org/info/rfc6775>.

9.2 Informative References

[DESIGN] Ayers, H. et al., "Design Considerations for Low Power Internet Protocols", Arxiv, June 2018.

[TINYOS] TinyOS Alliance, "TinyOS", 2018, <https://github.com/tinyos/tinyos-main>.

[ARM] ARM Mbed, "ARM Mbed OS", 2018, <https://github.com/ARMmbed/mbed-os>.

[RIOT] FU Berlin, "Riot OS", 2018, <https://github.com/RIOT-OS/RIOT>.

[CONTIKI] Dunkels, A., "Contiki OS", 2018, <https://github.com/contiki-os/contiki>.

[OPENTHREAD] Nest, "OpenThread", 2018, <https://github.com/openthread/openthread>.

[TOCK] Levy, A., Campbell, B., Pannuto, P., Dutta, P., Levis, P., "The Case for Writing a Kernel in Rust", APSys, 2017, <https://doi.org/10.1145/3124680.3124717>.

[6LO-CHART] Lemon, T., "IPv6 over Low power WPAN WG Charter", IETF, 2005, <https://datatracker.ietf.org/doc/charter-ietf-6lowpan/>.

[E2E] Saltzer, J. H., Reed, D. P., Clark, D. D., "End-to-end Arguments in System Design", ACM Trans. Comput. Syst., November 1984.

[RFC7400] Bormann, C., "6LoWPAN-GHC: Generic Header Compression for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs)", RFC 7400, DOI 10.17487/RFC7400, November 2014, <https://www.rfc-editor.org/info/rfc7400>.

Authors' Addresses

Hudson Ayers
Stanford University

EMail: hayers@stanford.edu

Paul Crews
Stanford University

EMail: ptcrews@stanford.edu

Hubert Hua Kian Teo
Stanford University

EMail: hteo@stanford.edu

Conor McAvity
Stanford University

EMail: cmcavity@stanford.edu

Amit Levy
Stanford University

EMail: levya@cs.stanford.edu

Philip Levis
Stanford University

EMail: pal@stanford.edu