Tagging Data In The Network Stack: *mbuf_tags*

Angelos D. Keromytis
Columbia University
angelos@cs.columbia.edu

Abstract

We describe the *mbuf_tag* API, a mechanism for tagging data as they flow through the network stack. Originally introduced in OpenBSD, *mbuf_tags* were initially intended for use by the IPsec stack. The API has matured enough to be used by several other kernel components, and formed the basis for the FreeBSD *mbuf_tags*. We present the API, discuss its various uses in the OpenBSD network stack, and describe some plans for future work. Our goal is to demonstrate the flexibility of this relatively simple mechanism and expose it to other kernel developers.

1 Introduction

The OpenBSD [1] *mbuf_tags* framework allows the kernel to attach arbitrary information to data packets as they flow through the network stack. This information is generally of two types: *(i)* a record of processing that has already been applied to the packet, *e.g.,* the fact that a packet was encrypted under a particular IPsec [5] security association, or *(ii)* a "reminder" to perform some operation to the packet in the future, *e.g.,* apply an encryption algorithm in software prior to transmitting the packet, if the outgoing interface does not provide integrated cryptographic facilities.

In the former case, the information is intended for consumption by the kernel itself (*e.g.,* detecting whether a processing loop has occurred, to avoid resource exhaustion) or by a user-level process (*e.g.,* exposing to a network daemon some IPsec-related information, indicating that a packet was protected by a particular security association). In the latter case ("reminders"), the information is intended for use by the lower levels of the network stack, *e.g.,* device drivers that offer specific functionality, such as outgoing packet checksumming.

Although originally developed for use by the IPsec stack, *mbuf_tags* have been in use by several other network components, such as various pseudo-devices, the packet filtering (PF) engine, and some device drivers. The use of *mbuf_tags* by such diverse elements demonstrates their effectiveness and usefulness as tools for the kernel developer. This is underlined by their adoption in the FreeBSD kernel for use in the recently revised IPsec stack.

The purpose of this paper is to expose the *mbuf_tags* mechanism to the general kernel developer community, both to encourage wider use and to solicit improvements to its functionality. We believe that *mbuf_tags* offer a flexible and simple mechanism that enables several types of processing that were previously difficult or impossible to perform in the BSD network stack.

The remainder of this paper is organized as follows. Section 2 discusses the design rationale and presents the API itself in some detail. Section 3 discusses the various uses of the *mbuf_tags* in the OpenBSD network stack, and Section 4 discusses some of our future work plans. Section 5 concludes the paper.

2 The OpenBSD *mbuf_tags*

In this section, we describe the design rationale and the API for *mbuf_tags*.

2.1 Design Rationale

The *mbuf_tags* were originally developed for use in conjunction with the OpenBSD IPsec stack [6]. Their primary purpose, described in more detail in Section 3, was to record how "securely" each packet was received, *i.e.,* under what IPsec security association(s) was a packet received. Practically immediately, however, a second use was established: to detect loops in outgoing IPsec packets. For this, we needed to record the same information as in the incoming packet case. However, we (correctly, as it turns out) foresaw the need for adding tags with different types of information in the future. Thus, we chose an approach similar in some respects to the BSD *struct sockaddr* for recording network address information.

```
struct m_tag {
        SLIST_ENTRY(m_tag) m_tag_link;
        u_int16_t          m_tag_id;
        u_int16_t          m_tag_len;
};

struct m_tag *m_tag_get(int type, int len, int flags);
struct m_tag *m_tag_find(struct mbuf *m, int type, struct m_tag *tag);
struct m_tag *m_tag_first(struct mbuf *m);
struct m_tag *m_tag_next(struct mbuf *m, struct m_tag *tag);
struct m_tag *m_tag_copy(struct m_tag *tag);

void          m_tag_free(struct m_tag *tag);
void          m_tag_prepend(struct mbuf *m, struct m_tag *tag);
void          m_tag_unlink(struct mbuf *m, struct m_tag *tag);
void          m_tag_delete(struct mbuf *m, struct m_tag *tag);
void          m_tag_delete_chain(struct mbuf *m, struct m_tag *tag);
void          m_tag_init(struct mbuf *m);

int           m_tag_copy_chain(struct mbuf *from, struct mbuf *to);
```

Figure 1: **The** *mbuf_tags* **API.**

mbuf_tags consist of two parts: a fixed-size header, which contains the length and type of the tag as well as a pointer to other tags attached to the same packet, followed by type-dependent data. These tags can be combined together in a chain, and attached to the first *mbuf* of a packet. An *mbuf* is a data structured used by the BSD networking stack to contain packets and other information. A chain of *mbufs* can be used to store large packets, with the first *mbuf* containing additional information about the packet as a whole.

These tags can serve multiple roles, as we shall see in Section 3. They can indicate processing that has already occurred to the packet (*e.g.*, IPsec SA under which a packet was received), or it may represent a "reminder" for processing that must be applied to the packet in the future (*e.g.*, cryptographic processing that must be done to the packet by a combined network+cryptographic accelerator card).

A similar approach was taken by NetBSD, in the form of *aux mbufs*. These are *mbufs* that are attached to an *mbuf* header in a way similar to *mbuf_tags* chains. Because they use unmodified *mbufs*, the former enable the use of all the *mbuf*-manipulating routines and, perhaps more importantly, allow space to be allocated in chunks without resorting to the kernel memory allocator with every allocation, as is the case with the use of *malloc(9)* in the *mbuf_tag* approach. Thus, the processing cost of adding new tags to a packet is a step function with *aux mbufs*, whereas it increases linearly with the number of tags attached to a packet in our scheme. However, the *mbuf_tag* approach does not place more pressure on the *mbuf* allocator, which can run out of space on a busy router or firewall, since *mbufs* are allocated from a reserved area of memory, which is typically fixed to a certain percentage of kernel memory at kernel-configuration time. Furthermore, we intend to use memory pool, as we discuss in Section 4, to reduce to overhead of the kernel memory allocator. Finally, *mbuf_tags* are better-integrated with the *mbuf* subsystem, allowing for seamless replication and de-allocation when the respective *mbufs* are duplicated or released.

Linux uses a 48-byte array in the *skbuff* structure, which the "owner" of a packet (the protocol or socket that queued the packet) can use to store private information. Apart from its limited size, the ownership semantics of this array make it unsuitable for use in certain scenarios (*e.g.*, when the producer and the consumer of a tag are separated by code that performs its own processing that requires use of the array).

FreeBSD recently adopted *mbuf_tags*, adding a *cookie* field in the tag header. This allows for private, module-specific definition of new tags without requiring coordinated allocation among different modules/developers. We intend to include this change in the OpenBSD tag implementation.

2.2 Tags API

The *mbuf_tag* API is shown in Figure 1. The code implementing the API is contained in the file

sys/kern/uipc_mbuf2.c of the OpenBSD distribution.

m_tag_get() allocates a new tag of type type with *len* bytes of space following the tag header itself. The flag argument is passed directly to the kernel *malloc(9)*. If successful, *m_tag_get()* returns a memory buffer of (*len* + *sizeof (struct m_tag))* bytes. The first *sizeof (struct m_tag)* bytes will contain a tag header, the definition of which is also given in Figure 1. The first field contains a pointer to other tags on the same mbuf. The length field contains the size, in bytes, of the array following the tag header itself. There are several types defined, and we describe their use in Section 3. *m_tag_free()* de-allocates a tag.

m_tag_find() finds an instance of a tag of the given type. The caller can specify that the search start from an arbitrary point in the tag list (as indicated by the third argument). This allows the caller to examine all tags of a given type that are attached to a packet, by repeatedly calling *m_tag_find()*, as shown in Figure 2.

```
/*
 * This code can be written
 * better, but this fits in
 * the two-column format :-)
 */
tag = m_tag_find(m, type, NULL);
while (tag != NULL) {
   ... code examining the tag ...
   tag = m_tag_find(m, type, tag);
}
```

Figure 2: **Using** *m_tag_find()***.**

For clarity, the *m_tag_first()* and *m_tag_next()* pair of calls can be used to the same effect.

m_tag_prepend() links a new tag to the head of the list. Tags are typically attached in a manner that reflects the order in which the operations they represent were applied to the packet. For example, a packet that is processed (*e.g.*, decrypted) by IPsec twice will have two attached tags, the first of which (as returned by *m_tag_find()*) will represent the second decryption. *m_tag_unlink()* detaches a tag from the packet, without deallocating the memory. *m_tag_delete()* combines *m_tag_unlink()* and *m_tag_free()*. *m_tag_delete_chain()* unlinks and frees all tags attached to a packet, starting from a caller-specified tag. If the last argument is left *NULL*, all attached flags will be deleted.

m_tag_copy() creates an identical copy of a tag. *m_tag_copy_chain()* creates a copy of the tag list attached to a packet and attaches it to another packet.

Finally, *m_tag_init()* is called by kernel components that manually initialize *mbufs*. There are only a handful of such locations, practically all of them in device drivers that perform their own buffer management (*e.g.*, maintaining a cache of *mbufs*).

3 Using *mbuf_tags*

We now describe the various uses of the tags in the OpenBSD network stack. The tags currently in use can be classified into four categories: IPsec, loop-detection, PF (packet filter), and miscellaneous. We describe each of these categories in turn.

3.1 IPsec-related Tags

This category includes tags that are used internally by the IPsec stack [6] to detect processing loops and propagate information to high-level protocols.

- IPSEC_IN_DONE records the fact that the packet was received under a particular IPsec Security Association (SA). If the packet has been encrypted under several SAs, there will be one such tag for each SA, with the most recently processed SA located closer to the head of the list. The tag contains enough information to locate the SA data structure in the relevant kernel database. This is used for two purposes:

  1. Determine whether a packet has been processed by an SA that satisfied the IPsec policy requirements, *e.g.*, "all TCP packets from host A must arrive encrypted". There are various locations in the network stack where such checks are performed.
  2. Propagate the information to the socket layer, whereby it is made available to applications via the *getsockopt()* call. Thus, applications can determine whether a connection is protected, the relevant parameters, the peer's identity (*e.g.*, public-key certificate), *etc.*

  This is one of the few tags that is used both inside and outside the kernel component where it is created (IPsec stack). The fact that tag processing (in particular freeing) is integrated with *mbuf* processing (freeing) helped in limiting the amount of supporting code that needed to be added throughout the stack.

- IPSEC_OUT_DONE records IPsec SAs that have been applied to an outgoing packet. This is primarily used to catch processing loops in the network stack, which could cause repeated processing (encryption) of a packet under the same set of SAs. This is necessary because the IPsec standards [5] require support for nested SA processing. Consider the following legitimate policy: "all packets to subnet 10.1.2.0/24 must be encrypted to the security gateway 10.1.2.1". Notice that the security gateway's address lies within the destination subnet's address space. A packet that matched this rule once would thus repeatedly match it every time it was re-evaluated by the IPsec policy database, causing a loop. Using the IPSEC_OUT_DONE tag, we can detect this cycle (or any cycle, of arbitrary length) and transmit the packet without further IPsec processing.

- IPSEC_IN_CRYPTO_DONE is issued by device drivers to indicate that an incoming IPsec packet has been successfully processed by a network card that has integrated support for IPsec, indicating the SA(s) processed. Incoming packets undergo regular IPsec processing; just prior to decryption/verification, the kernel checks for the presence of this tag for the specific SA. If this is present, decryption is skipped and processing continues as if it were successful. This allowed us to integrate IPsec-offloading support with less than 10 lines of kernel code.

- IPSEC_OUT_CRYPTO_NEEDED is used for the outgoing case of using a network card with integrated cryptographic processing. If the kernel is aware that the outgoing network interface offers such capabilities it simply attaches this tag to the packet, again indicating which SA it should be processed under. The device driver is then responsible for loading the SA parameters to the network card (if necessary), and for indicating to the hardware that IPsec processing under that SA is needed.

- IPSEC_IN_COULD_DO_CRYPTO is issued by device drivers that detect incoming IPsec packets for which they do not have the SA. The IPsec stack can use this tag as a signal that cryptographic processing can be off-loaded to the network interface. Although the device driver could silently load the relevant SA for end-systems, the situation is more complicated for gateways and firewalls that allow IPsec traffic to traverse them: in that case, the driver may not know which SAs are "local" and which refer to hosts behind the firewall. Since such knowledge is implicitly available to the network stack (different code will be executed), we made that code responsible for SA loading.

  Furthermore, the IPsec stack is (or can be) more aware about usage patterns across multiple SAs and can make better-informed decisions as to how best to use the limited resources available in the network card (such cards can typically support a limited number of SAs in their internal RAM).

- IPSEC_PENDING_TDB is used by the network stack to indicate that IPsec processing should occur to the packet before it is transmitted to the network. One tag for each SA that needs to be applied to the packet is attached, in the order in which they must be applied. This tag is necessary because of the requirement for SA-bundle processing (*i.e.*, policy may require that a packet be processed by a series, or "bundle", or SAs — not just one SA) and the fact that in OpenBSD cryptographic processing uses continuations [8].

3.2 Loop-Detection Tags

These tags are issued and consumed entirely by the bridge [7], *gif* interface, and *gre* interface subsystems respectively. Their main purpose is to detect processing cycles that would cause endless encapsulation or layer-2 packet forwarding. In all cases, the packet is dropped (in contrast to the IPsec loop-detection recovery, discussed in the previous section).

Systems without *mbuf_tags* have addressed the problem of loop detection/avoidance through ad-hoc and unsafe methods. In most cases, processing is assumed to be single-threaded; loops are detected through the use of locks or global variables. Not only is this approach is infeasible in multi-threaded and SMP kernels, it will also not work in certain configurations. For example, consider the case of two or more virtual bridges sharing two Ethernet interfaces: a packet scheduled for transmission in one interface of one bridge will be also scheduled for transmission on the other interface of the same bridge, also "jumping" to the second virtual bridge, whereupon it will be scheduled for transmission on the original Ethernet interface, whereupon the cycle will restart. The previous method cannot detect this failure, because bridge processing occurs at a software interrupt, making it impossible to keep packet state independently of the packet. Other similar cases arise then two or more of GRE, IP-in-IP, and bridging are combined in particular ways.

Furthermore, use of locks or global variables for detecting re-entry would make it difficult to implement

some legitimate configurations, *e.g.*, hierarchical bridges (whereby traffic propagated through one set of interfaces is also sent through a second set, but not vice versa). Admittedly, these uses are rather arcane and error-prone — they are mentioned only as an example of the flexibility of using tags for loop detection.

- BRIDGE is used by the bridge subsystem [7] to detect loops. The tag contains a pointer to the bridge interface that already forwarded the frame, allowing multi-bridge packet processing.

- GIF is used by *gif*, a network pseudo-interface that implements IP-in-IP encapsulation [9], to detect loops. Such loops are possible when the outer IP header, which is attached during *gif* processing, has a destination address that will cause the routing table to transmit the packet through the same *gif* interface again. The tag contains a pointer to the *gif* interface that processed the packet.

- Finally, GRE is used by the GRE encapsulation [3] code to detect cycles. The details are the same as with the GIF case; here, IP packets are encapsulated within GRE (and then IP) frames, and the tag contains a pointer to the *gre* interface that processed the packet.

3.3 PF-related Tags

These tags are used exclusively by PF, the OpenBSD packet filtering engine [4]. Unless indicated otherwise, these tags do not carry any additional data.

- PF_GENERATED is used to mark packets that are *generated* by PF itself, *e.g.*, ICMP messages indicating a dropped packet, or firewall-generated TCP RST packets. Such packets should not be subjected to the PF filtering rules, thus PF unconditionally accepts packets that carry this tag.

- PF_ROUTED is used to mark packets that are *routed* by the packet filtering engine, *e.g.*, using the *rdr* rule. Such packets are not tested by PF more than once, to prevent loops caused by subsequent matching routing rules.

- PF_FRAGCACHE is used to mark fragmented packets cached by PF. PF may cache such fragments as directed by its configuration, for traffic normalization purposes, *e.g.*, to avoid overlapping-fragment attacks. Packets with this tag have been cached by the fragment cache already and will short-circuit it if processed again. If they were to re-enter the fragment cache, they would be indistinguishable from a duplicate packet, and would be dropped.

- PF_QID is used by PF to indicate to the network traffic-shaping discipline, ALTQ, which queue the packet should go to. The tag contains the identifier of the queue.

- PF_TAG is used by PF to tag packets with user-defined information, and filter on those later on. Effectively, the tag is an internal marker that can be used to identify these packets. For example, such tags can be used to propagate information between input and output filtering rules on different interfaces, or to determine if packets have been processed by address-translation rules. These tags are *sticky*, meaning that the packet will be tagged even if the rule that attaches the tag is not the last matching rule. Further matching PF rules can replace that tag with a new one, but will not remove a previously-applied tag. A packet is only ever assigned one tag at a time.

3.4 Miscellaneous Tags

- IN_PACKET_CHECKSUM is used by network cards that can compute complete packet checksums to pass that information to higher-level protocols. That tag contains the 2-byte checksum of the complete packet. A protocol such as TCP needs to "subtract" the non-relevant parts of the packet from the checksum. This type of support was added for some of the older Intel cards, that did not compute protocol-specific checksums as newer hardware does.

4 Future Directions

There are several improvements we intend to make to the current API. More specifically:

- Use of memory pools (see the *pool(9)* manual page) and tag-specific deallocation routines, to improve performance. The limitation of using memory pools is that it supports fixed-size allocations which can lead to either inefficient use of memory or to a large numbers of pools. Fortunately, it appears that all the tags that have been defined to date fall in one of three categories with respect to memory allocation: they require no additional memory (beyond the tag header itself) — as was the case with

most of the PF tags, see Section 3.3, one extra word (the loop-detection tags, Section 3.2), or an IPsec SA identification payload (Section 3.1). Thus, we could simply have three different memory pools, one for each size. This change is fairly simple and does not require any changes in the API itself, so we intend to integrate it fairly soon.

- Tag-triggers, which will invoke specific packet-processing at various points in the network stack, depending on existing tags. One example is calculating the TCP or IP checksum of a packet that is about to be encapsulated inside another protocol. In the extreme case, these tags will carry a pointer to a protocol-specific function and enough data to indicate the location where the desired operation should take place. This is particularly useful when used in conjunction with network cards that support some type of functionality offloading and IPsec: if only some packets are IPsec-processed, we need a way to defer expensive processing (such as checksum computations) as late as possible, under the assumption that the packet will not be encrypted and thus the expensive operation can be offloaded to the NIC. When that assumption is violated (*i.e.,* the packet does need to be encrypted), we need to detect and apply the deferred operation. Such deferred processing is already done for TCP and UDP checksum computation, but the approach is tailored for that application. We intend to create a more general framework for deferred processing using *mbuf_tags*.

- An API for application-defined tags. This will be used either directly by applications or through *setsockopt()* calls to attach information to packets that will cause deferred processing. We have an application of this, for accelerating TLS [2] and SSH in the presence of network cards with integrated cryptographic functionality. In that scenario, the tags are used by crypto-aware network interfaces to provide application-layer protocol encryption. We intend to investigate this approach further in future work. Naturally, the type of tags that can be attached by applications must be carefully controlled, since these tags can affect network processing in ways that may not have been intended or allowed by the system administrator (*e.g.,* bypassing PF rules). Fortunately, doing so should be fairly straightforward, since the relevant interface to the kernel (*setsockopt()*) is "narrow" enough that we can perform the necessary checks.

5 Conclusions

We have presented the OpenBSD *mbuf_tags*, a mechanism for tagging packets as they flow through the network stack. These tags are used by many different kernel components such as the IPsec stack, various pseudo-interfaces, the packet filtering engine (PF), *etc.* We discussed the design rationale, the API, and the uses of the tags in OpenBSD, as well as some future improvements we intend to make. The *mbuf_tags* have been in use in OpenBSD for several years, and were recently ported to FreeBSD. *mbuf_tags* represent a powerful and flexible mechanism for allowing kernel developers to perform certain types of processing on packets in different parts of the network stack.

References

[1] T. de Raadt, N. Hallqvist, A. Grabowski, A. D. Keromytis, and N. Provos. Cryptography in OpenBSD: An Overview. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, pages 93 – 101, June 1999.

[2] T. Dierks and C. Allen. The TLS protocol version 1.0. Request for Comments (Proposed Standard) 2246, January 1999.

[3] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina. Generic routing encapsulation (GRE). Request for Comments 2784, Internet Engineering Task Force, March 2000.

[4] Daniel Hartmeier. Design and Performance of the OpenBSD Stateful Packet Filter (pf). In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, pages 171–180, June 2002.

[5] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401, November 1998.

[6] A. D. Keromytis, J. Ioannidis, and J. M. Smith. Implementing IPsec. In *Proceedings of Global Internet (GlobeCom)*, pages 1948–1952, November 1997.

[7] A. D. Keromytis and J. L. Wright. Transparent Network Security Policy Enforcement. In *Proceedings of the USENIX Technical Conference*, pages 201–214, June 2000.

[8] A. D. Keromytis, J. L. Wright, and T. de Raadt. The design of the openbsd cryptographic framework. In *Proceedings of the USENIX Technical Conference*, pages 181–196, June 2003.

[9] C. Perkins. IP encapsulation within IP. Request for Comments 2003, Internet Engineering Task Force, October 1996.