Authentication For Fragments

Craig Partridge
BBN Technologies
craig@bbn.com

Abstract

Recently there's been renewed interest in authenticating fragments, especially for mobile wireless networks and delay tolerant networks.

In this paper we review the issues in authenticating fragments and show how painfully hard that process is. We then suggest a few new ideas for reducing the authentication data that must be transmitted, which may make authenticating fragments a little less painful.

1 Introduction

Fragmentation is the process of breaking up a self-contained chunk of data into smaller pieces for the purposes of transmission. Historically, network protocol designers have tried to avoid using fragmentation because fragmentation increases data's vulnerability to loss and can substantially reduce effective throughput. Recently, however, there has been renewed interest in fragmentation as a way to address a challenge in delay tolerant networks [7] (DTNs) and certain mobile ad-hoc networks.

The challenge is the following. Suppose the network has important links that are only available (a short) part of the time. That is, when the link comes up, it will only stay up for a brief period before being taken down. This can easily be the case in a deep space DTN, as deep space communications links are usually available only for fixed time periods. Another example is a mobile ad-hoc network, where some nodes move between clusters and carry data with them [6]. We would like to maximize the use of the link – especially as the link may be the only means of communication with some portions of the network.

Now consider a router or gateway node that has data to send over the link. Some of this data is complete: a full datagram or a DTN bundle. But some of the data may be partial: a datagram still being received or a partial DTN bundle. Given the limited lifetime of the link, it may be desirable to push some of the partial data across the link *now*, to maximize link utilization and send the remainder the next time the link is up, or via less-effective, alternative paths. Fragmentation is the clear solution to this problem: one sends the incomplete data as fragments – to be reassembled with the still-to-be-received data later.

A variant on this situation is where the traffic at the gateway node is of different priorities and we may wish to pre-empt the transmission of low-priority traffic upon the arrival of new high-priority traffic for the link.

But there's a hazard here. The gateway has a strong interest in knowing that data sent over a link is supposed to go over the link (that we're not wasting the valuable bandwidth on the lower-layer equivalent of spam or a denial of service attack). That suggests the gateway needs to authenticate the data. Unfortunately, insofar as any data is authenticated today, the unit of authentication is typically the full data chunk, the datagram or bundle, *not* the fragment. So when we send a fragment, we risk discovering later that the fragment should not have been sent.

This paper discusses methods to authenticate fragments to address this problem.

2 Prior Work

The literature in fragmentation and authentication of fragments is modest.

2.1 Issues in Fragmentation

The literature on the perils of fragmentation is small, but extremely good. Kent and Mogul [1] demonstrated that use of fragmentation could substantially reduce the reliability of delivery of IP datagrams and even found a case where a router's buffering mechanism caused every instance of a fragmented IP datagram to be lost. Mathis *et al.* [4] have recently gone a step further and shown that the IP identifier (used to relate fragments back to their original datagram) is small enough that an adversary can attack the reassembly process.

Floyd and Romanow expanded our understanding of the effects of fragmentation while studying the fragmentation of IP datagrams into ATM cells [2]. In particular, they showed that once a fragment (cell) of a datagram is lost, such that the datagram cannot be reassembled, all subsequent fragments should also be discarded because otherwise those subsequent fragments continue to compete for bandwidth and buffering and may cause fragments of other datagrams to be lost (thus making it impossible to reassemble those datagrams).[1]

2.2 Authenticating Fragments

There's no published literature on authenticated fragments. There's been a little discussion on the *dtn-interest* mailing list. In the DTN community, there's a scheme for authenticating fragments known as the *toilet paper* approach [3].

The idea is that, much as toilet tissue is subdivided into regular pieces, the sender chooses to authenticate the datagram in regular pieces. The fragmenting gateway can then fragment the datagram into those pieces with confidence.

A challenge in this scheme, of course, is choosing the size of the pieces such that they are consistent with (or at least less than or equal to) a fragment size that the gateway can support.

Another challenge is where to put the authentication information for each fragment. There are two choices and they differ in the cost of protecting the authentication information from being altered or replaced in transit.

Looking at that tradeoff in more detail, each piece of data has some associated information (such as the result of a one-way hash) that can be used to authenticate it. The authentication information, however, must be protected lest someone change the data in flight and also replace the authentication information such that the new data appears genuine. This protection requires more information be sent with the data. One method for providing this protection is to digitally sign the authentication information using public key techniques.

The first choice is to put all the authentication information in one place (usually the front of data chunk) and then protect all the authentication information as a unit. This approach has the benefit of amortizing the protection information (which is of fixed size) across all the authentication information, but as section 4.2 points out, is problematic when fragments travel different paths.

The alternative is to embed each piece of data's authentication information with the piece. But that means we must also embed protection information with each piece. As protection information is often larger than authentication information, this is a non-trivial space requirement.

[1] One can use burst erasure coding to convert this situation into one where if k ($k>1$) out of m fragments are lost, the datagram is lost [8,9].

3 Some Formality

To avoid confusion in the rest of the paper, this section defines some terms and states the basic problem in those terms.

We start with a unit of data, called a *datagram*, which has an *origin* that has generated authentication information for the datagram, both in its entirety and into pieces $d_1 \ldots d_m$. The m pieces need not be the same size. A datagram can be a file, a DTN bundle, an IP datagram, or an ATM cell. The point is that the datagram is self-standing unit of data that is being sub-divided.

As the datagram works its way through a network towards its destination, it reaches a *gateway*. The gateway wishes to break the datagram into n pieces, called *fragments* ($f_1 \ldots f_n$). While the length of each fragment may be the same, the length may also vary (widely) by fragment. The gateway need not have received the entire datagram to begin fragmenting it. There be more than one gateway in a path, and so fragments may be refragmented further. To simplify discussion, we assume that the gateway can fragment (or refragment) at any point in the data and at any time, including during transmission.[2]

We assume the presence of an *adversary* that is unable to generate datagrams that can be authenticated and yet show someone else as their origin[3], but by virtue of its position in the network, is capable of changing or deleting data in datagrams or fragments in transit, may influence the path that a fragment takes through the network, and is also capable of creating plausible looking fragments that will, however, fail authentication. For rigor, we will assume this is an active or intelligent adversary, even though these harmful events can happen naturally.

The challenge is to devise a method of authentication such that we can examine a particular fragment, f_i, and determine if it is indeed a valid fragment of a particular datagram.

4 What, Exactly, Are We Protecting?

It is important to keep in mind that the goal is to protect valuable link bandwidth and that authenticating fragments is a means to that end, not the end itself. In that spirit, it is useful to examine how the link bandwidth needs protecting.

[2] This assumption is often not true – fragments may have to be a minimum size.

[3] The adversary can, of course, generate datagrams that authenticate and show the adversary as the origin.

4.1 No Loss, No Multipath

We start with a simple situation. We assume that there is no loss of fragments and that all fragments follow the same path. The adversary is capable of corrupting or spoofing fragments. We have a gateway *G* with a link *L* over which it seeks to protect the bandwidth and some number of links that feed data to *G* that could be transmitted over *L*.

Now consider *L* and *G*'s possible states.

- ❑ *L* is idle and *G* has no data.
- ❑ *L* is in use and *G* has no other data to send.
- ❑ *L* is idle and *G* has data to send (which may or may not be authenticated).
- ❑ *L* is in use and *G* has data to send (which may or may not be authenticated).

Only the last two states are of interest – we look more deeply at both.

If *L* is idle and *G* has data, then the interesting situation is what to do if none of the data is authenticated yet. (If some of the data is authenticated, obviously *G* sends that data.) If no data is authenticated, the answer, in most situations, is to pick some piece of data and start sending it until either (a) another piece of data is authenticated — in which case *G* stops sending the current fragment and starts sending the authenticated data; or (b) the data being transmitted can be authenticated — in which case *G* either stops sending (if authentication fails) or continues to send (now with more confidence). The logic is that we wish to maximize the use of bandwidth, and sending unauthenticated data that may prove to be valid is better than leaving the link idle.[4]

The case of *L* being in use and *G* having data is simply case (b) of the previous paragraph.

Note that if data has multiple priorities, the decisions are slightly tougher. In particular, if *L* is in use and is sending low-priority authenticated data and *G* receives unauthenticated data that claims to be higher priority, the question is what to do. The assumption here is that *G* only acts once the higher priority data has been authenticated, in which situation, case (a) applies.

4.2 Adding Multipath (and Loss)

We now make life harder by assuming that fragments may follow multiple paths, so that if *G* is the second (or third or …) gateway in the path, it cannot expect all fragments of a datagram to go through it. The relevance is that Floyd and Romanow's work comes into play. In a single path environment, *G* could assume that a missing fragment meant that subsequent fragments (even if they were authenticated) were no longer useful. In a multipath environment, that's not the case. Each fragment stands alone. And unless there is some method of cross-path bookkeeping to detect dead fragments, each fragment must be assumed to be part of a datagram that is awaiting reassembly. So we may continue to send fragments along path B, even though the loss of a fragment on path A means the fragments on path B are no longer useful.

Unfortunately, there isn't a good solution to this problem. We could, of course, force all fragments to follow the same path, but that's inconsistent with the idea, expressed earlier, of using fragmentation to opportunistically utilize links that are available only part of the time.

Another important issue is that *G* cannot be sure of seeing d_1. That observation constrains how authentication data is placed in the datagram. In particular, it is not possible to envision, say, placing all authentication information in d_1 and then simply checking the fragments as they arrive. The authentication information must be available for any individual d_i, which may sharply increase authentication overhead (in particular, because of the need to separately protect each piece of authentication information from being tampered with in flight).

There are a variety of solutions to this problem, including placing authentication information in each d_i, or transmitting d_1 (or the authentication section of d_1) down every path that fragments follow, or transmitting d_1 down a path before sending any subsequent fragments over that path. A key question tends to be how large the authentication information is – an issue we examine in more detail in section 5.3.

4.3 Boundary Problems

So far, we've not worried about the fact that the sizes of the d_i could be different from the size of the fragments. So fragment, f_x, could contain trailing data from d_y and starting data from d_{y+1}. In short, authentication boundaries and fragmentation boundaries may not match.

That turns out be a serious issue for several reasons.

First, consider f_x. In the general case, with multiple paths and loss, a gateway *G* has no way to authenticate the fragment. If *G* also receives f_{x+1} and the data in that fragment completes d_{y+1}, then *G* can partially

[4] There are exceptions to this rule. If the link is a shared media, some other sender may have authenticated data and *G* should stay quiet.

authenticate f_x, but only if G can find the boundary between d_y and d_{y+1} in the data payloads.[5]

But even if we assume that G sees all fragments, there are opportunities for an adversary to make mischief. Suppose the adversary sees f_x and inserts a bad fragment right after it, g_{x+1}, which claims to be f_{x+1}. At best, G must be prepared to recover from this situation, by saving the incremental authentication information from f_x until the true f_{x+1} arrives. At worst, G may send some data from the bad fragment before realizing the error (and there may be multiple instances of the bad fragment before the real f_{x+1} arrives).

The strong suggestion is that authentication boundaries should be consistent with fragmentation boundaries (in particular, no fragment should contain an incomplete portion of an authenticated piece of data). Note this has implications for the discussion in section 4.1 as it may no longer be a good idea to stop sending an unauthenticated fragment when another fragment gets authenticated — rather G must keeping sending until it reaches an authentication boundary (the end of a d_i).

4.4 Authentication Challenges

Before looking at a new idea or two, we need to summarize the challenges of applying existing authentication schemes to the fragmentation problem.

Because there are multiple paths that a datagram (much less its fragments) could take, it seems unlikely that the origin could establish a security association with every gateway the datagram might travel through. So, the most logical alternative is that every datagram or fragment is self-authenticating – which strongly suggests using hashes protected by public-key signatures.

Now public-key signatures are expensive to check, so there's a strong desire to do the check once. That's quite feasible – one can put all the authenticating information for $d_1 \ldots d_m$ in one place and use a public-key signature to sign the information.

Unfortunately, this approach runs into the problem that different fragments may follow different paths. That yields three ugly choices: either copy the (signed) authentication data into each fragment, or arrange to send a copy of the authentication data down each path a fragment might traverse, or authenticate and publicly sign each fragment separately. Depending on the value of m, one or the other approach may make sense. However, the overhead (both in bits and repeated public key operations) in any situation is potentially large.

Further we have a problem of packing data efficiently into fragments, both to maximize link utilization and also to maximize the data covered by each each self-standing fragment's authentication information.

[5] Finding data boundaries in arbitrary payloads or data stream subject to corruption is a difficult problem. Almost all bit and byte stuffing protocols can permanently fail to recover sync. See chapter 2 of [5] for a discussion and chapter 3 for a byte-stuff algorithm that always recovers sync.

5 New Ideas for Authenticating Fragments

The remainder of the paper is a discussion of some new ideas for authenticating fragments.

5.1 Cumulative Authentication

The first set of ideas are some a simple extensions of the toilet paper approach.

In cumulative authentication, rather than authenticating each fragment individually, we authenticate cumulatively. Slightly more formally, given $d_1 \ldots d_m$, we define the authentication information such that a_i is the result of running the authentication function A over $d_1 \ldots d_i$ (vs. the toilet paper approach where a_i is the result of running A over d_i alone).

The appeal of cumulative authentication is that a gateway can only authenticate d_i if it has seen $d_1 \ldots d_{i-1}$. This approach solves the Floyd-Romanow problem of "dead fragments." However, it requires that all fragments follow the same path and (roughly) in order – a requirement that often is not reasonable.

5.2 Multi-Increment Authentication

Multi-increment authentication supposes that the origin divides the datagram up twice, once into chunks $d_1 \ldots d_m$ and also into $\delta_1 \ldots \delta_p$, where $p \neq m$ and the sizes of the d's and the δ's are different. The idea is to give the gateways greater flexibility in how they choose to fragment the data, by allowing them to fragment in different sizes. For instance, one could set the size of the δ's to be 1/4th the size of the d's and allow gateways to pack data more tightly into fragments.

The potential drawbacks of this scheme are the need to run an authentication algorithm twice over each datagram and the possibly higher costs of two sets of authentication information.

5.3 Authentication Using Function Definitions

One of the recurring themes with authentication of fragments is the size of the authentication information that validates each fragment. The idea in this section is to sharply reduce the authentication information by using a different approach to authenticating the data. In particular, rather than protecting the data with a hash value from an authentication function – we protect the data with a dynamically generated function that gives a known result.

Making this idea concrete, consider an authentication function A, whose definition is that, given $d_1 \ldots d_m$ of a particular datagram, and an initialization vector iv for that datagram, the hashes are:

$$A(iv,d_1)=1, A(iv,d_2)=2,\ldots,A(iv,d_m)=m$$

That is, we find an iv for each datagram, such that the hash value for each piece of the datagram is that piece's sequence number or ordering in the datagram.

The hope here is that we can create an iv which is relatively small and thus produces less overhead than, say, m digitally signed one-way hashes. In the next couple of subsections, we look at two possible approaches to creating A.

5.3.1 Minimal Perfect Hashing

One possible function definition is that of a *minimal perfect hashing* function. Recall that a perfect hashing function is one that, given a set of m distinct keys, generates a different hash value for each key in a range n ($n>m$). A minimal perfect hashing function generates different hash values in the case where $m=n$.

Any set of distinct keys is guaranteed to have a minimal perfect hash function, though finding the function is currently an exponential computation. The resulting functions can apparently be represented in $O(log(m^2))$ or less bits, which for practical purposes (millions of fragments of one datagram) is sharply smaller than m digitally signed hashes, each of which could be up to 2048 bits long (e.g. an RSA signature).

If the function definition is small enough, it becomes easier to transmit the function definition down any path that a fragment might follow. Note that a function definition is probably not, however, smaller than a single digitally signed hash, for the simple reason that we have to sign the function definition.

Assuming $d_1 \ldots d_m$ are distinct[6] we could define A to be a minimal perfect hash function, and iv would be the state for the function. Note that while the function maps $d_1 \ldots d_m$ to values in $1..m$, there's no guarantee that it maps d_i to i. To achieve that step we can either (if ordering is not important), reshuffle the order of the d_i's to match their hash order, or add to iv a table of m offsets which convert the particular value of d_i to be i.

5.3.1.1 Minimal Perfect Hashes Can Be One-Way

One important question is whether a minimal perfect hash function can be a one-way hash function. This section shows an inefficient one-way minimal perfect hash.

Take a standard one-way hash function, H, where $H(d_i)=h_i$. A good example is MD5. Now define iv, our initialization vector to be m entries long, where $iv[i]=i-h_i$. The one-way minimal perfect hash function, OW, is then simply: $OW(d_i)=H(d_i)+iv[i]$.

5.3.1.2 Open Issue: Space Efficient One-Way Minimal Perfect Hash Functions

Given that one-way minimal perfect hash functions exist, the challenge is to find ones that have a small representation. (The algorithm of section 5.3.1.1 has an initialization vector as big as the authentication information that H would have produced anyway).

The author does not, alas, have such a function to propose. However, we note that minimal perfect hash functions are highly sensitive to small changes in their input data, and the functions themselves are rare. So it is plausible to hope that space efficient one-way minimal perfect hash functions exist.

5.3.2 Multiple Functions

Another idea is to create pairs or groups of functions that together work to authenticate a fragment.

So, for instance, one could define a pseudo-random number generator function, and create an authentication function that produces the values of the sequence as the successive hashes of the d_i's.

6 Conclusions

Fragmentation is a mechanism fraught with problems. As we try to find ways to authenticate fragments, much of the time we are simply reminded of how painful fragmentation is. We cannot solve the Floyd/Romanow problem in a network where individual fragments follow different paths. We need to match the size of the origin's authenticated data pieces with the size of the eventual fragments.

In this painful design space, we've offered a few ideas that may be useful. Multi-increment authentication may make packing data pieces into fragments easier. Authentication using function definitions may reduce the size of authentication information (if appropriate functions can be found – and we suggest that's a plausible expectation).

7 References

1. C.A. Kent and J.C. Mogul, "Fragmentation Considered Harmful," *Proc. of SIGCOMM '87*, pp. 390-401, Stowe, Vermont, August, 1987.

[6] We can, if necessary, add a salt or sequence number, to make each one unique.

2. A. Romanow and S. Floyd, "Dynamics of TCP Traffic over ATM Networks," *IEEE Jour. Selected Areas Communication*, v. 13, n. 4, May 1995, pp. 633-641.

3. http://mailman.dtnrg.org/pipermail/dtn-interest/2005-April/001892.html

4. M. Mathis, J. Heffner, and B. Chandler, *Fragmentation Considered Very Harmful*, http://www.psc.edu/~jheffner/drafts/draft-mathis-frag-harmful-XX.html.

5. S.D. Cheshire, *Consistent Overhead Byte Stuffing*, Stanford Doctoral Dissertation, 1998.

6. J.P.G. Sterbenz, R. Krishnan, R.R. Hain, A.W. Jackson, D. Levin, R. Ramanathan, and J. Zao, "Survivable Mobile Wireless Networks: Issues, Challenges and Research Directions," *ACM Workshop on Wireless Security (WiSe)*, September 2002.

7. Delay Tolerant Networks Working Group. http://www.dtnrg.org/wiki

8. A. McAuley, "Reliable Broadband Communication using a Burst Erasure Correcting Code," *Proc. ACM SIGCOMM '90*, pp. 297-306.

9. N. Shacham and P. McKenney, "Packet recovery in high-speed networks using coding and buffer management," *Proc. IEEE INFOCOM '90*, pp. 124-131.

Acknowledgements

Rajesh Krishnan and Steve Kent provided a number of very valuable comments on an early draft of this paper. The author takes all responsibility for any deficiencies.