Project	**IEEE 802.16 Broadband Wireless Access Working Group <http://ieee802.org/16>**
Title	**Clarification to AES-CTR mode usage in Section 7.8.2.1 in IEEE P802.16e/D5a, Dec 2004**
Date Submitted	**2005-01-11**
Source(s)	L. Dondeti, H. He, M. Fong, H. Zhang (Nortel); Voice: +1 978.288.6406 ldondeti@nortel.com
Re:	Followup to sponsor ballot comment
Abstract	The current version of the standard specifies AES-CTR mode usage with a 32-bit nonce. There are a number of ways to generate a nonce and there is a potential for incorrect use as well as interoperability issues due to the lack of detail in the current specification. This submission ~~clarifies~~ provides the necessary clarification and revisions to the current text.
Purpose	This submission will describe the problem with the current specification, and a resolution. It contains necessary revisions to the draft version D5a.
Notice	This document has been prepared to assist IEEE 802.16. It is offered as a basis for discussion and is not binding on the contributing individual(s) or organization(s). The material in this document is subject to change in form and content after further study. The contributor(s) reserve(s) the right to add, amend or withdraw material contained herein.
Release	The contributor grants a free, irrevocable license to the IEEE to incorporate material contained in this contribution, and any modifications thereof, in the creation of an IEEE Standards publication; to copyright in the IEEE's name any IEEE Standards publication even though it may include portions of this contribution; and at the IEEE's sole discretion to permit others to reproduce in whole or in part the resulting IEEE Standards publication. The contributor also acknowledges and accepts that this contribution may be made public by IEEE 802.16.
Patent Policy and Procedures	The contributor is familiar with the IEEE 802.16 Patent Policy and Procedures <http://ieee802.org/16/ipr/patents/policy.html>, including the statement "IEEE standards may include the known use of patent(s), including patent applications, provided the IEEE receives assurance from the patent holder or applicant with respect to patents essential for compliance with both mandatory and optional portions of the standard." Early disclosure to the Working Group of patent information that might be relevant to the standard is essential to reduce the possibility for delays in the development process and increase the likelihood that the draft publication will be approved for publication. Please notify the Chair <mailto:chair@wirelessman.org> as early as possible, in written or electronic form, if patented technology (or technology under patent application) might be incorporated into a draft standard being developed within the IEEE 802.16 Working Group. The Chair will disclose this notification via the IEEE 802.16 web site <http://ieee802.org/16/ipr/patents/notices>.

Complete specification of AES-CTR mode use in Section 7.8.2.1

Lakshminath R. Dondeti

1. Introduction

Section 7.8.2.1 of IEEE P802.16e/D5a specifies “Data encryption with AES in CTR mode.” According to the specification, the mechanism in 7.8.2.1 may be implemented in conjunction with the pseudo code in Appendix E, and specifically the function *encrypt_pdu()* in pages 481-2. While there is some room for interpretation, a strict implementation following the specification would result in an incorrect use of CTR mode, and allows an attacker to derive clear text from cipher text making the encryption operation ineffective. Fortunately, there is a fairly easy fix to this problem.

2. CTR mode semantics

CTR mode is specified in NIST special publication 800-38A and RFC 3686. First, a counter block is encrypted with the encryption key, and the encrypted bits are XOR’ed with a block of data. Assuming AES-128 is the cipher used, a 128-bit counter is encrypted with the AES encryption key and the cipher text is XOR’ed with a 128-bit block of the PDU. CTR mode is only secure as long as all the counters are distinct from each other within the auspices of a given encryption key. In other words, as long as the encryption key stays the same, the counter cannot be repeated, and it is important to note that this rule applies at the *block* level, not at the MPDU level.

If a counter value were to be reused with the same key, an adversary can derive information about the PDU blocks encrypted with that counter block and key combination. For instance if P1 and P2 are two plain text blocks encrypted with the same key and counter block, then the XOR value of the associated cipher text blocks C1 and C2 is equivalent to the XOR value of P1 and P2. Furthermore, if any parts of either P1 or P2 are known values (or easily guessable values, such as protocol headers), then the corresponding parts in the other plain text block can be derived easily.

Thus, a data encapsulation protocol using CTR mode must ensure that the counter value does not repeat. The simplest technique to avoid repeating counter values is to initialize the counter to a random 128-bit (or whatever the block size is) value, and monotonically increase it per *block*. The secret key must be changed *before* the counter value becomes the initial value.

In unreliable communication (e.g., IPsec for IP), the counter value must be sent as part of the packet. To reduce per-packet overhead, the IV size must be kept to as small as possible; at the same time, a smaller IV would require more frequent rekeying. Another consideration is that a typical packet contains multiple blocks, which implies that there should be room for a block counter within the packet. Taking all these factors into consideration, RFC 3686 defines a 64-bit per packet IV and a 32-bit block counter (taking into account IPv6 jumbogram sizes). The most significant 32-bits contain a per-session nonce to protect against precomputation attacks.

3. CTR mode use in the current specification

The current specification (Section 7.8.2.1 in pages 206-7) uses a 32-bit nonce repeated 4 times to generate a 128-bit per block counter. From the Appendix (encrypt_pdu() in pages 481-2), it is clear that the nonce is a 32-bit per-MPDU r*andom* value. The same function suggests that after starting with a random nonce, the per-block keys (within the PDU) are obtained simply by incrementing the nonce (note: the entire 128-bit value is incremented).

There are two problems with this approach: first, due to birthday attack, there is a good chance of repeating the counter, if the nonce is chosen randomly (a 50% chance of collision in 78K PDUs transmitted), second, due to the method used for per-block counter generation (in the Appendix in encrypt_pdu() function), there is a guaranteed chance of collision (when 2^{32}-1 is the nonce value, adding 1 makes the IV=0000; an adversary could then just wait for the PDU with 0 as the 32-bit nonce for cryptanalysis).

4. Proposed resolution

The proposed resolution to this problem is rather simple, and requires a few changes to Section 7.8.2.1 and the pseudo code for the encrypt_pdu() and some related functions. First, we describe the outline of the solution. To keep the packet expansion to a minimum as in the current specification (we will discuss the ramifications of the small nonce at the end of this document), we continue the use a 32-bit counter (note: counter instead of a nonce), *initialized* to a random value. The 32-bit counter is concatenated four times (e.g., cur-ctr-val | cur-ctr-val | cur-ctr-val | cur-ctr-val) to generate a 128-bit block counter required by the AES cipher for CTR mode. The counter value is increased by 1 per block, but only the 32 least significant bits (i.e., no overflow thereafter) are used for the counting. This allows each PDU to have as many as 2^{32} 128-bit blocks (which is plenty for 802.16 PDUs)

5. Proposed changes to the specification

5.1 Proposed changes to Section 7.8.2.1

Revise 7.8.2.1 (page 206-7 in 802.16e/D5a) as follows:

7.8.2.1 Data encryption with AES in CTR mode

If the data encryption algorithm identifier in the cryptographic suite of an MBS GSA equals 0x80, data on connections associated with that SA shall use the CTR mode of the US Advanced Encryption Standard (AES) algorithm [NIST Special Publication 800-38A, FIPS 197, RFC 3686] to encrypt the MAC PDU payloads. In MBS, t~~T~~he AES block size ~~for AES~~ and cipher counter block are 128 bits.~~in this case is 128 bits; therefore we need a 128-bit counter block for the cipher. To account for unreliable or out-of-order delivery, the counter (or IV) is included in the PDU. To reduce per-PDU overhead, 32-bit IV is sent as part of the PDU and a full IV generation mechanism is specified below.~~

7.8.2.1.1 PDU payload format

The PDU payload shall be prepended with a 32-bit IV~~nonce~~. Each base station in the MBS zone shall use the same IV~~nonce~~. The IV~~nonce~~ shall be transmitted least-significant byte first. The IV~~nonce~~ shall not be encrypted.

The IV~~nonce~~ shall be repeated four times to construct 128-bit counter block required by the AES-128 cipher~~s nonce~~. (e.g.,~~Ex~~. IV~~NONCE~~|IV~~NONCE~~|IV~~NONCE~~|IV~~NONCE~~). This mechanism can reduce per-PDU overhead of transmitting counter.

~~7.8.2.1.2 Outbound PDU processing~~

The plaintext PDU shall be encrypted using the active MBS_Traffic_key (MTK) derived from MAK and MGTEK, according to CTR mode specification. The initial IV (*initialIV*) is randomly generated. Then the IV is incremented by 1 for each subsequent PDU.~~The sender maintains a 128-bit block level counter, initialized to a 32-bit random value (*initialIV*) and expanded to 128-bits as shown in the previous section. For each PDU to be transmitted, that sender divides the PDU into 128-bit blocks and follows the CTR mode specification to encrypt it.~~ A different 128-bit counter value~~ – obtained by incrementing the 32-bit IV by 1 and regenerating the 128-bit counter –~~ is used to encrypt each 128-bit block within a~~the~~ PDU. This can be achieved by only incrementing ~~Only~~ the lowest~~ order~~ 32-bits by 1;~~,~~ the lowest 32-bit value ~~are incremented – in other words, the counter~~ may rotate to 0 and counted up. ~~The IV for the next PDU will be *initialIV+1*.~~

The processing yields a payload that is 32 bits longer than the plaintext payload. The MTK must be rekeyed before the IV value becomes equal to *initialIV*. In other words, at the most 2^{32} PDUs can be encrypted with a single MTK.

~~7.8.2.1.3 Inbound PDU processing~~

~~In CTR mode, the inbound PDU processing is quite similar to the outbound processing. The receiver obtains the current IV from the PDU header, and initializes the 128-bit block level counter to the received IV. Next it forms the 128-bit counter value using the procedure described in Section 7.8.2.1.1. Successive counter values are obtained by incrementing the least significant 32-bits of the IV value by 1. The receiver then uses the process in AES-CTR mode specification to decrypt the PDU on a block by block basis.~~

```
/*****************************************************************/
/* It increment counter by one upon encryption of each block */
```

```
/*************************************************************/
void add_counter(char *ctr)
{
        int value, i;
        int overflow;
        overflow = 1;
        for ( i=3~~15~~; i>=0 ; i-- ) {
        if ( overflow == 0 ) break;
        value = ctr[i] & 0xff;
        value ++;
        if ( value >= 256 )
        overflow = 1;
        else overflow = 0;
        ctr[i] = value & 0xff;
        }
}
void ~~generate~~initialize_nonce(unsigned char *nonce)
{
        unsigned long value = htonl(random_32bit());
        memcpy(nonce,(char*)&value,4);
}
/*******************************************************/
/* int encrypt_pdu() */
/* Encrypts a plaintext pdu in accordance with */
/* the proposed 802.16e AES CTR specification. */
/* Nonce insertion takes place. */
/* Returns the resulting cipher text */
/*******************************************************/
int encrypt_pdu(unsigned char *key, unsigned char nonce, unsigned char *plain, int len, unsigned char *cipher)
{
        int i, n_blocks, n_remain, out_len = 0;
        unsigned char ctr[16]~~,nonce[4];~~;
        unsigned char aes_out[16], remain[16], temp[16];
        ~~generate_nonce(nonce);~~
        ~~#ifdef DEBUG~~
        ~~printf("Generate 32bit nonce : ");~~
        ~~print_hex(nonce,4);~~
        ~~#endif~~
        for (i=0; i<4; i++)
                cipher[i] = nonce[i];
        out_len += 4;
        n_blocks = len / 16;
        n_remain = len % 16;
        init_counter(nonce,ctr);
        #ifdef DEBUG
        printf("Initialize Counter: ");
        print_hex(ctr,16);
        printf("Key: ");
        print_hex(key,16);
        #endif
        for ( i=0; i< n_blocks; i++ ) {
                aes128k128d(key, ctr, aes_out);
                bitwise_xor(aes_out, &plain[i*16], &cipher[i*16+4]);
                add_counter(ctr);
                out_len += 16;
        }
        for ( i=0; i<16; i++ ) {
                remain[i] = 0;
        }
        for ( i=0; i<n_remain; i++ ) {
                remain[i] = plain[n_blocks*16+i];
        }
        aes128k128d(key,ctr,aes_out);
        bitwise_xor(aes_out,&remain[0], &temp[0]);
        for ( i=0; i<n_remain; i++ ) {
                cipher[n_blocks*16+4+i] = temp[i];
        }
        out_len += n_remain;
        return out_len;
}


int test_case(int length)
{
        unsigned char key[16], nonce[4];
        unsigned char plain[MAX_BUF];
        unsigned char cipher[MAX_BUF+4];
        unsigned char decrypt[MAX_BUF];
        /* 0. Get a 128bits key */
        generate_key(key);
        generate_nonce(nonce);
        /* strcpy(initialIV, nonce); */
        /* 1. Generate Plain Text with length */
        generate_plain(plain,length);
        #ifdef DEBUG
        printf("PLAIN TEXT ------------------------------------\n");
        print_hex(plain,length);
        #endif
        /* 2. Encrypt Plain Text to Cipher Text */
        encrypt_pdu(key,nonce,plain,length,cipher);
        /* if there are multiple PDUs, nonce will be incremented and checked as follows*/
        /* nonce++; */
        /* if strcmp(nonce, initialIV) != 0 */
        /*      call encrypt next pdu */
        /* else rekey(key) */
        #ifdef DEBUG
        printf("CIPHER TEXT -----------------------------------\n");
        print_hex(cipher,length+4);
        #endif
        /* 3. Decrypt Cipher Text to decrypt text */
        decrypt_pdu(key,nonce,cipher,length+4,decrypt);
        #ifdef DEBUG
        printf("DECRYPT TEXT ----------------------------------\n");
        print_hex(decrypt,length);
        #endif
        /* 4. Compare decrypt text and original plain text */
        if ( compare(decrypt,plain,length) == 0 ) {
                return 1; /* Test Success */
        } else {
                return 0; /* Test Failure */
        }
}
```