Network Security

Secure Channel

Cornelius Diekmann

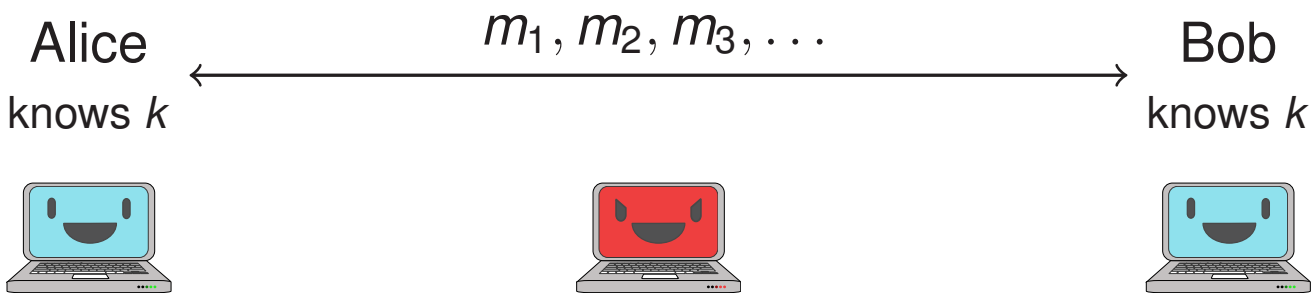
Lehrstuhl für Netzarchitekturen und Netzdienste
Institut für Informatik
Technische Universität München

Version: November 9, 2015

Agenda

1. Secure Channel
2. MAC-then-Enc vs. Enc-then-MAC
3. Secure Channel Implementation
4. Secure Channel (ESP) in the OpenBSD Kernel
5. Authenticated Encryption With Associated Data
6. AEAD & Secure Channel (ESP) in the Linux Kernel

Secure Channel



What do we want?

- Confidentiality, Integrity, Authenticity
- Messages received in correct order
- No duplicates and we know which messages are missing

MAC-then-Enc or Enc-then-MAC?

$$\mathrm{Enc}_{k\text{-}enc}(m, \mathrm{MAC}_{k\text{-}int}(m))$$

vs.

$$\mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(m)$$

vs.

$$\mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(\mathrm{Enc}_{k\text{-}enc}(m))$$

MAC-then-Enc or Enc-then-MAC?

$$\mathrm{Enc}_{k\text{-}enc}(m, \mathrm{MAC}_{k\text{-}int}(m))$$

vs.

$$\mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(m)$$

vs.

$$\mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(\mathrm{Enc}_{k\text{-}enc}(m))$$

vs. $\mathrm{Enc}_{k\text{-}enc}(\mathrm{MAC}_{k\text{-}int}(m))$

MAC-then-Enc or Enc-then-MAC?

$$\mathrm{Enc}_{k\text{-}enc}(m, \mathrm{MAC}_{k\text{-}int}(m))$$

vs.

$$\mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(m)$$

vs.

$$\mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(\mathrm{Enc}_{k\text{-}enc}(m))$$

vs. ~~$\mathrm{Enc}_{k\text{-}enc}(\mathrm{MAC}_{k\text{-}int}(m))$~~

- Cannot recover m

MAC-then-Enc or Enc-then-MAC?

$$\mathrm{Enc}_{k\text{-}enc}(m, \mathrm{MAC}_{k\text{-}int}(m))$$

vs.

$$\mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(m)$$

vs.

$$\mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(\mathrm{Enc}_{k\text{-}enc}(m))$$

$\mathrm{Enc}_{k\text{-}enc}(m, \mathrm{MAC}_{k\text{-}int}(m))$

- "*The encryption protects the MAC*"

$\mathrm{Enc}_{k\text{-}enc}(m, \mathrm{MAC}_{k\text{-}int}(m))$

- ~~"*The encryption protects the MAC*"~~
  - Encryption does not provide any message authenticity/integrity!

$\mathrm{Enc}_{k\text{-}enc}(m, \mathrm{MAC}_{k\text{-}int}(m))$

- ~~"*The encryption protects the MAC*"~~
  - Encryption does not provide any message authenticity/integrity!
- Example: A weak MAC cannot be "protected" by encrypting it
- CRC is not a MAC, OTP is perfect encryption
- $\mathrm{OTP}_k(m, \mathrm{CRC}(m))$ does not provide any integrity

$\mathrm{Enc}_{k\text{-}enc}(m, \mathrm{MAC}_{k\text{-}int}(m))$

- ~~"*The encryption protects the MAC*"~~
  - Encryption does not provide any message authenticity/integrity!
- Example: A weak MAC cannot be "protected" by encrypting it
- CRC is not a MAC, OTP is perfect encryption
- $\mathrm{OTP}_k(m, \mathrm{CRC}(m))$ does not provide any integrity
- Attacker can $\oplus x$ to encrypted message and $\oplus \mathrm{CRC}(x)$ to the encrypted CRC to fix it

$\mathrm{Enc}_{k\text{-}enc}(m, \mathrm{MAC}_{k\text{-}int}(m))$

- Horton principle:"*Authenticate what you mean, not what you say*"
- "*Authenticate the plaintext!*"

$\mathrm{Enc}_{k\text{-}enc}(m, \mathrm{MAC}_{k\text{-}int}(m))$

- Horton principle:“*Authenticate what you mean, not what you say*”
- ~~“*Authenticate the plaintext!*”~~
  - Do you really mean `char *`?

$\mathrm{Enc}_{k\text{-}enc}(m, \mathrm{MAC}_{k\text{-}int}(m))$

- Horton principle:"*Authenticate what you mean, not what you say*"
- ~~"*Authenticate the plaintext!*"~~
  - Do you really mean `char *`?

- Horton principle applies to application layer

$\mathrm{Enc}_{k\text{-}enc}(m, \mathrm{MAC}_{k\text{-}int}(m))$

- Horton principle:"*Authenticate what you mean, not what you say*"
- ~~"*Authenticate the plaintext!*"~~
  - Do you really mean `char *`?

- Horton principle applies to application layer
- E.g., Signing a contract

`http://www.personalausweisportal.de/DE/Buergerinnen-und-Buerger/Online-Ausweisen/Das-brauche-ich/Kartenlesegeraete/Kartenlesegeraete_node.html`

$\mathrm{Enc}_{k\text{-}enc}(m, \mathrm{MAC}_{k\text{-}int}(m))$

- Horton principle:"*Authenticate what you mean, not what you say*"
- ~~"*Authenticate the plaintext!*"~~
  - Do you really mean `char *`?

- Horton principle applies to application layer
- E.g., Signing a contract
- The secure channel transports chunks of bytes

http://www.personalausweisportal.de/DE/Buergerinnen-und-Buerger/Online-Ausweisen/Das-brauche-ich/Kartenlesegeraete/Kartenlesegeraete_node.html

$\mathrm{Enc}_{k\text{-}enc}(m, \mathrm{MAC}_{k\text{-}int}(m))$

- Horton principle:"*Authenticate what you mean, not what you say*"
- ~~"*Authenticate the plaintext!*"~~
  - Do you really mean `char *`?

- Horton principle applies to application layer
- E.g., Signing a contract
- The secure channel transports chunks of bytes out of context
- $m_1 =$ `"<!--"`,
  $m_2 =$ `"I owe you $1000"`,
  $m_3 =$ `"-->"`

http://www.personalausweisportal.de/DE/Buergerinnen-und-Buerger/Online-Ausweisen/Das-brauche-ich/Kartenlesegeraete/Kartenlesegeraete_node.html

$\mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(m)$

- Not better than $\mathrm{Enc}_{k\text{-}enc}(m, \mathrm{MAC}_{k\text{-}int}(m))$

$\mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(\mathrm{Enc}_{k\text{-}enc}(m))$

- $c \leftarrow \mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(c)$

$\mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(\mathrm{Enc}_{k\text{-}enc}(m))$

- $c \leftarrow \mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(c)$
- Considered secure

$\mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(\mathrm{Enc}_{k\text{-}enc}(m))$

- $c \leftarrow \mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(c)$
- Considered secure
- Discard bogus messages before decryption

$\mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(\mathrm{Enc}_{k\text{-}enc}(m))$

- $c \leftarrow \mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(c)$
- Considered secure
- Discard bogus messages before decryption
  - Don't waste CPU power

$\mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(\mathrm{Enc}_{k\text{-}enc}(m))$

- $c \leftarrow \mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(c)$
- Considered secure
- Discard bogus messages before decryption
  - Don't waste CPU power
  - Don't generate error messages that might help an attacker

$\mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(\mathrm{Enc}_{k\text{-}enc}(m))$

- $c \leftarrow \mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(c)$
- Considered secure
- Discard bogus messages before decryption
  - Don't waste CPU power
  - Don't generate error messages that might help an attacker
  - Don't touch non-authentic data!

Examples

- $\mathrm{Enc}_{k\text{-}enc}(m, \mathrm{MAC}_{k\text{-}int}(m))$
  - MAC then encrypt
  - SSL $\leftarrow$ many SSL attacks are a result of this scheme
  - Horton Principle
- $\mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(m)$
  - MAC & encrypt
  - SSH
  - Horton Principle
  - Considered the weakest
- $\mathrm{Enc}_{k\text{-}enc}(m), \mathrm{MAC}_{k\text{-}int}(\mathrm{Enc}_{k\text{-}enc}(m))$
  - Encrypt then MAC
  - IPSec (ESP), Signal (TextSecure ProtovolV2), probably TLS 1.3 [RCF7366]
  - Considered the most secure

Our Secure Channel Implementation

Secure Channel Implementation

- We need
  - Message numbering
  - Authentication
  - Encryption
- Our Toy Implementation
  - Message numbering: n next slide
  - Authentication: HMAC-SHA-256

    $\mathrm{MAC}_{k\text{-}int}(n \parallel IV \parallel c)$
  - Encryption: AES-128-CTR

    $c \leftarrow \mathrm{ENC}_{k\text{-}enc}(IV, m)$
  - keys for each purpose

Message Numbering

- $n \in \mathbb{N}$
- Increased monotonically for each valid message
- n must be unique for every message
- Remember last message n_{last} and only accept $n > n_{last}$

Message Numbering

- $n \in \mathbb{N}$
- Increased monotonically for each valid message
- n must be unique for every message
- Remember last message n_{last} and only accept $n > n_{last}$
- Detect replays
- Correct order
- Detect lost messages

Message Numbering

- $n \in \mathbb{N}$
- Increased monotonically for each valid message
- n must be unique for every message
- Remember last message n_{last} and only accept $n > n_{last}$
- Detect replays
- Correct order
- Detect lost messages
- Number overflow $\rightarrow$ rekeying

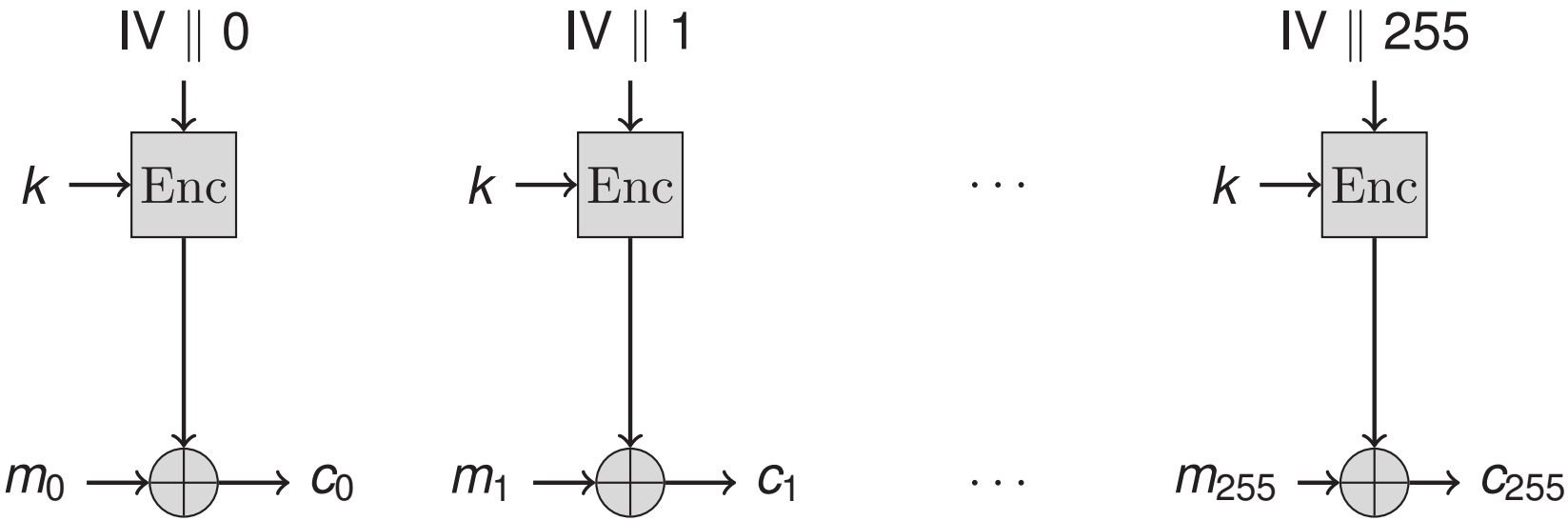
Initialize (at Alice)

```
# Output: 128bit key
def KDRV(k):
    # TODO: There are better key derivation functions
    # Assumes: random oracle property of SHA1
    return SHA1(k)

# Initialize global variables (keys and message number)
def init_globals(k):
    global K_send_enc, K_recv_enc, K_send_int, K_recv_int, n_send,
        n_recv, used_nonces
    K_send_enc = KDRV(k || "Enc Alice to Bob")
    K_recv_enc = KDRV(k || "Enc Bob to Alice")
    K_send_int = KDRV(k || "MAC Alice to Bob")
    K_recv_int = KDRV(k || "MAC Bob to Alice")
    n_send = 0
    n_recv = 0
    used_nonces = {}
```

- Generate one key for each purpose
- Where $\cdot \,||\, \cdot$ means string/byte concatenation

AES-128-CTR Mode needs IV

- $ctr_i = \text{IV} \parallel i$
- ctr_i is of length 128 bit: We chose 120 bit IV and 8 bit i



- Max message size per IV: $2^8 \cdot 128 = 32768$ bit $= 4096$ Bytes
- For $i \in \{0 \ldots 254\}$: $ctr_{i+1} = ctr_i + 1$

Nonces as IV for AES-CTR

```
used_nonces = {}

# Output: A fresh 120bit nonce
def nonce():
    global used_nonces
    n = random_bits(120)
    if n not in used_nonces:
        used_nonces.add(n)
        return n
    else:
         # TODO: may not terminate if no unused nonces are left
        return nonce()
```

- We want a fresh IV $\rightarrow$ remember used nonces
- We are super paranoid:
  - Random nonces
  - A counter would suffice

Sending a Message

```
def send(m):
    global n_send, K_send_enc, K_send_int
    if n_send >= MAX_INT:
        return ERROR("MSG Number overflow, needs rekeying")

    if len(m) > 4096:
        return ERROR("MSG too large, needs fragmentation")

    IV = nonce()

    c = ENC-AES-128-CTR(K_send_enc, IV, m)

    t = HMAC-SHA-256(K_send_int, n_send || IV || c)

    socket_send(n_send || IV || c || t)

    n_send = n_send + 1
```

Verifying a MAC

```
def verify(k, msg, t):
    return HMAC-SHA-256(k, msg) == t
```

Verifying a MAC correctly

```
def verify(k, msg, t):
    return timingsafe_bcmp(HMAC-SHA-256(k, msg), t, 32)
```

OpenBSD/sys/lib/libkern/timingsafe_bcmp.c

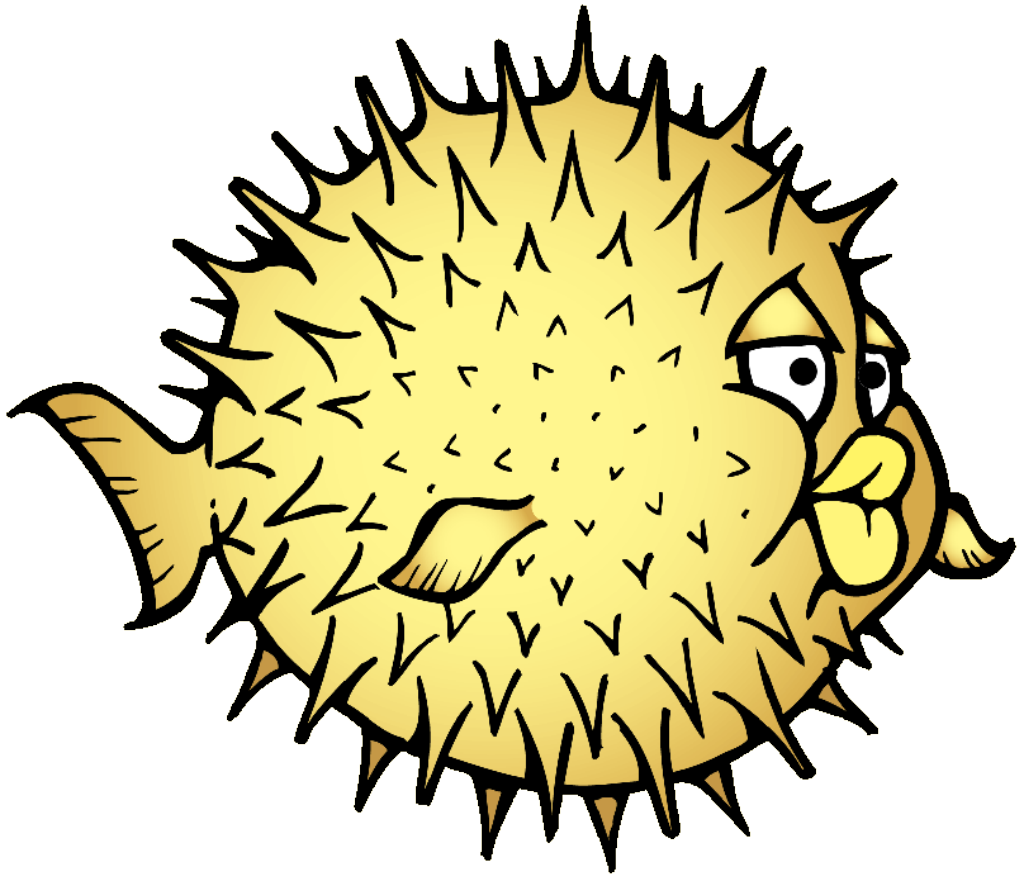
```
int timingsafe_bcmp(const void *b1, const void *b2, size_t n)
{
    const unsigned char *p1 = b1, *p2 = b2;
    int ret = 0;

    for (; n > 0; n--)
        ret |= *p1++ ^ *p2++;
    return (ret != 0);
}
```

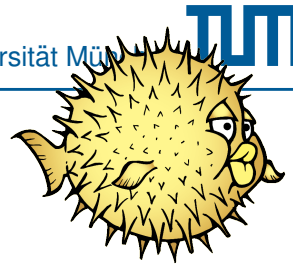
The timingsafe_bcmp() and timingsafe_memcmp() functions lexicographically compare the first len bytes (each interpreted as an unsigned char) pointed to by b1 and b2. Additionally, their running times are independent of the byte sequences compared, making them safe to use for comparing secret values such as cryptographic MACs. In contrast, bcmp(3) and memcmp(3) may short-circuit after finding the first differing byte.

Receiving a Message

```
def receive(msg):
    global n_recv, K_recv_int, K_recv_enc
    if n_recv + 1 >= MAX_INT:
        return ERROR("MSG Number overflow, need rekeying")

    n, IV, c, t = parse(msg)

    if not verify(K_recv_int, n || IV || c, t):
        return ERROR("MAC verification failed")

    if n < n_recv:
        return ERROR("Received old message")

    if n != n_recv + 1:
         print "lost %d messages" % (n - (n_recv + 1))

    n_recv = n

    m = DEC-AES-128-CTR(K_recv_enc, IV, c)

    return m
```

IPSec ESP in the OpenBSD Kernel



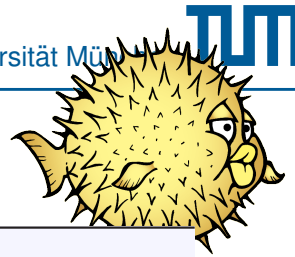


ESP Input Processing

sys/netinet/ip_esp.c OpenBSD 5.8

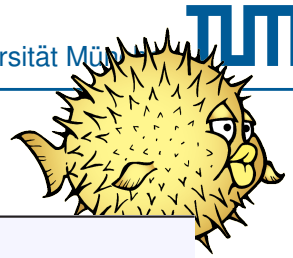
```
/*
 * ESP input processing, called (eventually) through the protocol switch.
 */
int
esp_input(struct mbuf *m, struct tdb *tdb, int skip, int protoff)
{
    struct auth_hash *esph = (struct auth_hash *) tdb->tdb_authalgxform;
    struct enc_xform *espx = (struct enc_xform *) tdb->tdb_encalgxform;
    struct cryptodesc *crde = NULL, *crda = NULL;
    struct cryptop *crp;
    struct tdb_crypto *tc;
    int plen, alen, hlen;
    u_int32_t btsx, esn;

    /* Determine the ESP header length */
    hlen = 2 * sizeof(u_int32_t) + tdb->tdb_ivlen; /* "new" ESP */
    alen = esph ? esph->authsize : 0;
    plen = m->m_pkthdr.len - (skip + hlen + alen);
    if (plen <= 0) {
        DPRINTF(("esp_input: invalid payload length\n"));
        espstat.esps_badilen++;
        m_freem(m);
        return EINVAL;
    }
```

- Both encryption and authentication are optional in ESP

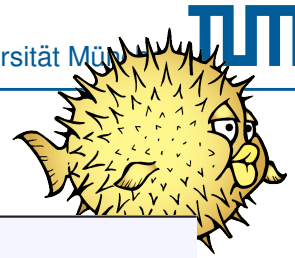


```
if (espx) {
    /*
     * Verify payload length is multiple of encryption algorithm
     * block size.
     */
    if (plen & (espx->blocksize - 1)) {
        DPRINTF(("esp_input(): payload of %d octets "
            "not a multiple of %d octets, SA %s/%08x\n",
            plen, espx->blocksize, ipsp_address(&tdb->tdb_dst,
            buf, sizeof(buf)), ntohl(tdb->tdb_spi)));
        espstat.esps_badilen++;
        m_freem(m);
        return EINVAL;
    }
}
```

- if encryption is to be applied

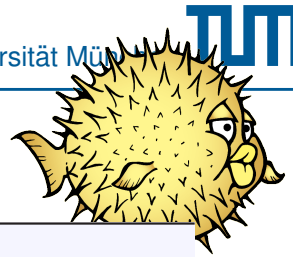


```
/* Replay window checking, if appropriate -- no value commitment. */
if (tdb->tdb_wnd > 0) {
    m_copydata(m, skip + sizeof(u_int32_t), sizeof(u_int32_t), (unsigned char *) &btsx);
    btsx = ntohl(btsx);

    switch (checkreplaywindow(tdb, btsx, &esn, 0)) {
    case 0: /* All's well */
        break;
    case 1:
        m_freem(m);
        DPRINTF(("esp_input(): replay counter wrapped for SA %s/%08x\n",
            ipsp_address(&tdb->tdb_dst, buf, sizeof(buf)), ntohl(tdb->tdb_spi)));
        espstat.esps_wrap++;
        return EACCES;
    case 2:
        m_freem(m);
        DPRINTF(("esp_input(): old packet received in SA %s/%08x\n",
            ipsp_address(&tdb->tdb_dst, buf, sizeof(buf)), ntohl(tdb->tdb_spi)));
        espstat.esps_replay++;
        return EACCES;
    case 3:
        m_freem(m);
        DPRINTF(("esp_input(): duplicate packet received in SA %s/%08x\n",
            ipsp_address(&tdb->tdb_dst, buf, sizeof(buf)), ntohl(tdb->tdb_spi)));
        espstat.esps_replay++;
        return EACCES;
    default:
        m_freem(m);
        DPRINTF(("esp_input(): bogus value from checkreplaywindow() in SA %s/%08x\n",
            ipsp_address(&tdb->tdb_dst, buf, sizeof(buf)), ntohl(tdb->tdb_spi)));
        espstat.esps_replay++;
        return EACCES;
    }
}
```



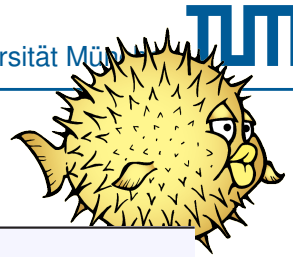
```
    /* Update the counters */
    tdb->tdb_cur_bytes += m->m_pkthdr.len - skip - hlen - alen;
    espstat.esps_ibytes += m->m_pkthdr.len - skip - hlen - alen;

    /* Hard expiration */
    if ((tdb->tdb_flags & TDBF_BYTES) &&
        (tdb->tdb_cur_bytes >= tdb->tdb_exp_bytes))    {
        pfkeyv2_expire(tdb, SADB_EXT_LIFETIME_HARD);
        tdb_delete(tdb);
        m_freem(m);
        return ENXIO;
    }

    /* Notify on soft expiration */
    if ((tdb->tdb_flags & TDBF_SOFT_BYTES) &&
        (tdb->tdb_cur_bytes >= tdb->tdb_soft_bytes)) {
        pfkeyv2_expire(tdb, SADB_EXT_LIFETIME_SOFT);
        tdb->tdb_flags &= ~TDBF_SOFT_BYTES;       /* Turn off checking */
    }

    /* Get crypto descriptors */
    crp = crypto_getreq(esph && espx ? 2 : 1);
    if (crp == NULL) {
        m_freem(m);
        DPRINTF(("esp_input(): failed to acquire crypto descriptors\n"));
        espstat.esps_crypto++;
        return ENOBUFS;
    }

...
```

- Keys may expire after certain number of bytes
- Note: packet might still be bogus, replay window not updated



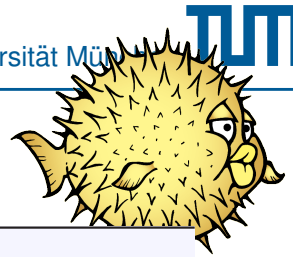
```
    if (esph) {
        crda = crp->crp_desc;
        crde = crda->crd_next;

        /* Authentication descriptor */
        crda->crd_skip = skip;
        crda->crd_inject = m->m_pkthdr.len - alen;

        crda->crd_alg = esph->type;
        crda->crd_key = tdb->tdb_amxkey;
        crda->crd_klen = tdb->tdb_amxkeylen * 8;

        if ((tdb->tdb_wnd > 0) && (tdb->tdb_flags & TDBF_ESN)) {
            esn = htonl(esn);
            bcopy(&esn, crda->crd_esn, 4);
            crda->crd_flags |= CRD_F_ESN;
        }

        if (espx && espx->type == CRYPTO_AES_GCM_16)
            crda->crd_len = hlen - tdb->tdb_ivlen;
        else
            crda->crd_len = m->m_pkthdr.len - (skip + alen);

        /* Copy the authenticator */
        m_copydata(m, m->m_pkthdr.len - alen, alen, (caddr_t)(tc + 1));
    } else
        crde = crp->crp_desc;

    /* Crypto operation descriptor */
...
```

- if authentication is to be applied



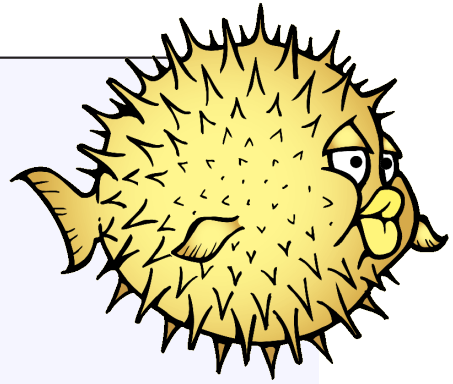
```
/* Decryption descriptor */
if (espx) {
    crde->crd_skip = skip + hlen;
    crde->crd_inject = skip + hlen - tdb->tdb_ivlen;
    crde->crd_alg = espx->type;
    crde->crd_key = tdb->tdb_emxkey;
    crde->crd_klen = tdb->tdb_emxkeylen * 8;
    /* XXX Rounds ? */

    if (crde->crd_alg == CRYPTO_AES_GMAC)
        crde->crd_len = 0;
    else
        crde->crd_len = m->m_pkthdr.len - (skip + hlen + alen);
}
```

- if encryption is to be applied



```
    return crypto_dispatch(crp);
}
```

- Dispatch to crypto driver (similar to Linux)
- A callback will be called once the crypto was done



```
/*
 * ESP input callback, called directly by the crypto driver.
 */
int
esp_input_cb(struct cryptop *crp)
{
...
    /* If authentication was performed, check now. */
    if (esph != NULL) {
...
        /* Verify authenticator */
        if (timingsafe_bcmp(ptr, aalg, esph->authsize)) {
            free(tc, M_XDATA, 0);
            DPRINTF(("esp_input_cb(): authentication failed for packet in SA %s/%08x\n",
                ipsp_address(&tdb->tdb_dst, buf, sizeof(buf)), ntohl(tdb->tdb_spi)));
            espstat.esps_badauth++;
            error = EACCES;
            goto baddone;
        }

        /* Remove trailing authenticator */
        m_adj(m, -(esph->authsize));
    }
    free(tc, M_XDATA, 0);

    /* Replay window checking, if appropriate */
...
    /* Verify pad length */
...
    /* Verify correct decryption by checking the last padding bytes */
...
}
```

- Check if everything was correct (in the right order)
- update replay window

AEAD

Authenticated Encryption With Associated Data (AEAD)

- Authenticated encryption: Encrypt then MAC
- Associated Data: Additional non-encrypted data but authenticated
- Example AD: IV, information necessary for message routing, . . .
- Special AEAD Algorithms: only need one pass over the data
  - Encrypt and MAC usually requires two passes
- Examples
  - Offset Codebook Mode (OCB)
  - Galois/Counter Mode (GCM)
- authenticated_encryption_and_padding_oracles slides now

AEAD & ESP in the Linux Kernel

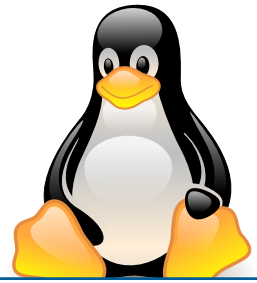
Linux 4.3 (stable, vanilla)

Authenticated Encryption With Associated Data (AEAD)

include/crypto/aead.h

```
/**
 * DOC: Authenticated Encryption With Associated Data (AEAD) Cipher API
 *
 * The AEAD cipher API is used with the ciphers of type CRYPTO_ALG_TYPE_AEAD
 * (listed as type "aead" in /proc/crypto)
 *
 * The most prominent examples for this type of encryption is GCM and CCM.
 * However, the kernel supports other types of AEAD ciphers which are defined
 * with the following cipher string:
 *
 *    authenc(keyed message digest, block cipher)
 *
 * For example: authenc(hmac(sha256), cbc(aes))
```

- AEAD API
- AEAD algorithm or combination of Enc and MAC



Authenticated Encryption With Associated Data (AEAD)

include/crypto/aead.h

```
/**
 * crypto_aead_encrypt() - encrypt plaintext
 * @req: reference to the aead_request handle that holds all information
 *     needed to perform the cipher operation
 *
 * Encrypt plaintext data using the aead_request handle. That data structure
 * and how it is filled with data is discussed with the aead_request_*
 * functions.
 *
 * IMPORTANT NOTE The encryption operation creates the authentication data /
 *          tag. That data is concatenated with the created ciphertext.
 *          The ciphertext memory size is therefore the given number of
 *          block cipher blocks + the size defined by the
 *          crypto_aead_setauthsize invocation. The caller must ensure
 *          that sufficient memory is available for the ciphertext and
 *          the authentication tag.
 *
 * Return: 0 if the cipher operation was successful; < 0 if an error occurred
 */
static inline int crypto_aead_encrypt(struct aead_request *req)
{
    return crypto_aead_alg(crypto_aead_reqtfm(req))->encrypt(req);
}
```

- encrypt and generate authentication tag

Authenticated Encryption With Associated Data (AEAD)

include/crypto/aead.h

```
/**
 * crypto_aead_decrypt() - decrypt ciphertext
 * @req: reference to the ablkcipher_request handle that holds all information
 *     needed to perform the cipher operation
 *
 * Decrypt ciphertext data using the aead_request handle. That data structure
 * and how it is filled with data is discussed with the aead_request_*
 * functions.
 *
 * IMPORTANT NOTE The caller must concatenate the ciphertext followed by the
 *          authentication data / tag. That authentication data / tag
 *          must have the size defined by the crypto_aead_setauthsize
 *          invocation.
 *
 *
 * Return: 0 if the cipher operation was successful; -EBADMSG: The AEAD
 *       cipher operation performs the authentication of the data during the
 *       decryption operation. Therefore, the function returns this error if
 *       the authentication of the ciphertext was unsuccessful (i.e. the
 *       integrity of the ciphertext or the associated data was violated);
 *       < 0 if an error occurred.
 */
static inline int crypto_aead_decrypt(struct aead_request *req)
...
```

- check authentication tag and decrypt

Authenticated Encryption With Associated Data (AEAD)

include/crypto/aead.h

```
/**
 * aead_request_set_crypt - set data buffers
 * @req: request handle
 * @src: source scatter / gather list
 * @dst: destination scatter / gather list
 * @cryptlen: number of bytes to process from @src
 * @iv: IV for the cipher operation which must comply with the IV size defined
 *      by crypto_aead_ivsize()
 *
 * Setting the source data and destination data scatter / gather lists which
 * hold the associated data concatenated with the plaintext or ciphertext. See
 * below for the authentication tag.
 *
 * For encryption, the source is treated as the plaintext and the
 * destination is the ciphertext. For a decryption operation, the use is
 * reversed - the source is the ciphertext and the destination is the plaintext.
 *
 * For both src/dst the layout is associated data, plain/cipher text,
 * authentication tag.
 *
 * The content of the AD in the destination buffer after processing
 * will either be untouched, or it will contain a copy of the AD
 * from the source buffer.  In order to ensure that it always has
 * a copy of the AD, the user must copy the AD over either before
 * or after processing.  Of course this is not relevant if the user
 * is doing in-place processing where src == dst.
 *
```

- constructing API request: encrypt this

```
/* IMPORTANT NOTE AEAD requires an authentication tag (MAC). For decryption,
 *          the caller must concatenate the ciphertext followed by the
 *          authentication tag and provide the entire data stream to the
 *          decryption operation (i.e. the data length used for the
 *          initialization of the scatterlist and the data length for the
 *          decryption operation is identical). For encryption, however,
 *          the authentication tag is created while encrypting the data.
 *          The destination buffer must hold sufficient space for the
 *          ciphertext and the authentication tag while the encryption
 *          invocation must only point to the plaintext data size. The
 *          following code snippet illustrates the memory usage
 *          buffer = kmalloc(ptbuflen + (enc ? authsize : 0));
 *          sg_init_one(&sg, buffer, ptbuflen + (enc ? authsize : 0));
 *          aead_request_set_crypt(req, &sg, &sg, ptbuflen, iv);
 */
static inline void aead_request_set_crypt(struct aead_request *req,
                      struct scatterlist *src,
                      struct scatterlist *dst,
                      unsigned int cryptlen, u8 *iv)
{
    req->src = src;
    req->dst = dst;
    req->cryptlen = cryptlen;
    req->iv = iv;
}
```

- Never forget about integrity/authenticity

```
/**
 * aead_request_set_ad - set associated data information
 * @req: request handle
 * @assoclen: number of bytes in associated data
 *
 * Setting the AD information.  This function sets the length of
 * the associated data.
 */
static inline void aead_request_set_ad(struct aead_request *req,
                       unsigned int assoclen)
{
    req->assoclen = assoclen;
}
```

- constructing API request: check integrity of this

ESP Implementation for IPv4

`net/ipv4/esp4.c`

- XFRM module

```
static const struct xfrm_type esp_type =
{
    .description    = "ESP4",
    .owner          = THIS_MODULE,
    .proto          = IPPROTO_ESP,
    .flags          = XFRM_TYPE_REPLAY_PROT,
    .init_state     = esp_init_state,
    .destructor     = esp_destroy,
    .get_mtu        = esp4_get_mtu,
    .input          = esp_input,
    .output         = esp_output
};

static struct xfrm4_protocol esp4_protocol = {
    .handler          =    xfrm4_rcv,
    .input_handler    =    xfrm_input,
    .cb_handler       =    esp4_rcv_cb,
    .err_handler      =    esp4_err,
    .priority         =    0,
};
```

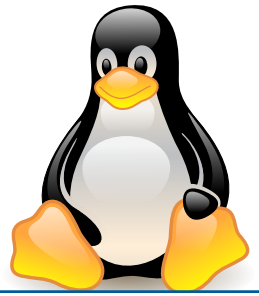
- Flags: XFRM will check the sequence number

An ESP Header Definition

include/uapi/linux/ip.h

```
struct ip_esp_hdr {
    __be32 spi;
    __be32 seq_no; /* Sequence number */
    __u8  enc_data[0]; /* Variable len but >=8. Mind the 64 bit alignment! */
};
```

- `spi`: Security Parameter Index, needed to associate packet
- `enc_data`: data of arbitrary length
- Just the header, the MAC/authenticator will be at the end of the packet (not defined in the struct)



ESP Input Processing

net/ipv4/esp4.c

```
static int esp_input(struct xfrm_state *x, struct sk_buff *skb)
{
    struct ip_esp_hdr *esph;
    struct crypto_aead *aead = x->data;
    struct aead_request *req;
    int ivlen = crypto_aead_ivsize(aead);
    int elen = skb->len - sizeof(*esph) - ivlen;
    int assoclen;
...
    assoclen = sizeof(*esph);

...
    aead_request_set_callback(req, 0, esp_input_done, skb);
...
    aead_request_set_crypt(req, sg, sg, elen + ivlen, iv);
    aead_request_set_ad(req, assoclen);

    err = crypto_aead_decrypt(req);
...
    return err;
}
```

- `aead_request_set_callback`: call this function when done
- add to AEAD request: decrypt the payload and verify the associated data (esp hdr)
- execute request