

Network Security (NetSec)

IN2101 – WS 16/17

Prof. Dr.-Ing. Georg Carle

Cornelius Diekmann

Version: February 24, 2017

Chair of Network Architectures and Services
Department of Informatics
Technical University of Munich

Secure Channel

MAC-then-Enc vs. Enc-then-MAC

Secure Channel Implementation

Secure Channel (ESP) in the OpenBSD Kernel

Authenticated Encryption With Associated Data

AEAD & Secure Channel (ESP) in the Linux Kernel

Attacks against a Secure Channel (Stream Cipher)

Attacks against a Secure Channel (Padding oracle)

Secure Channel

MAC-then-Enc vs. Enc-then-MAC

Secure Channel Implementation

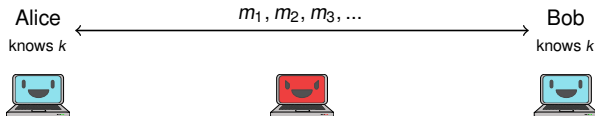
Secure Channel (ESP) in the OpenBSD Kernel

Authenticated Encryption With Associated Data

AEAD & Secure Channel (ESP) in the Linux Kernel

Attacks against a Secure Channel (Stream Cipher)

Attacks against a Secure Channel (Padding oracle)



What do we want?

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

Secure Channel

MAC-then-Enc vs. Enc-then-MAC

Secure Channel Implementation

Secure Channel (ESP) in the OpenBSD Kernel

Authenticated Encryption With Associated Data

AEAD & Secure Channel (ESP) in the Linux Kernel

Attacks against a Secure Channel (Stream Cipher)

Attacks against a Secure Channel (Padding oracle)

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

vs.

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

vs.

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

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

vs.

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

vs.

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

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

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

vs.

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

vs.

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

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

- Cannot recover m

- $\text{Enc}_{k\text{-enc}}(m, \text{MAC}_{k\text{-int}}(m))$:
 - *“The encryption protects the MAC”*

- $\text{Enc}_{k\text{-enc}}(m, \text{MAC}_{k\text{-int}}(m))$:
 - *“The encryption protects the MAC”*
 - Encryption does not provide any message authenticity/integrity!

- $\text{Enc}_{k\text{-enc}}(m, \text{MAC}_{k\text{-int}}(m))$:
 - ~~“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
 - $\text{OTP}_k(m, \text{CRC}(m))$ does not provide any integrity

- $\text{Enc}_{k\text{-enc}}(m, \text{MAC}_{k\text{-int}}(m))$:
 - *“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
 - $\text{OTP}_k(m, \text{CRC}(m))$ does not provide any integrity
 - Attacker can $\oplus x$ to encrypted message and $\oplus \text{CRC}(x)$ to the encrypted CRC to fix it

- $\text{Enc}_{k\text{-enc}}(m, \text{MAC}_{k\text{-int}}(m))$:
 - Horton principle: “*Authenticate what you mean, not what you say*”
 - “*Authenticate the plaintext!*”

- $\text{Enc}_{k\text{-enc}}(m, \text{MAC}_{k\text{-int}}(m))$:
 - Horton principle: “*Authenticate what you mean, not what you say*”
 - “~~Authenticate the plaintext!~~”
 - Do you really mean char *?

- $\text{Enc}_{k\text{-enc}}(m, \text{MAC}_{k\text{-int}}(m))$:
 - Horton principle: “*Authenticate what you mean, not what you say*”
 - “~~Authenticate the plaintext!~~”
 - Do you really mean char *?
- Horton principle applies to application layer

- $Enc_{k-enc}(m, MAC_{k-int}(m))$:
 - 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

- $Enc_{k-enc}(m, MAC_{k-int}(m))$:
 - 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

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

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

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

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

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

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

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

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

MAC-then-Enc vs. Enc-then-MAC

Examples

- $\text{Enc}_{k\text{-enc}}(m, \text{MAC}_{k\text{-int}}(m))$
 - MAC then encrypt
 - SSL ← many SSL attacks are a result of this scheme
 - Horton Principle
- $\text{Enc}_{k\text{-enc}}(m), \text{MAC}_{k\text{-int}}(m)$
 - MAC & encrypt
 - SSH
 - Horton Principle
 - Considered the weakest
- $\text{Enc}_{k\text{-enc}}(m), \text{MAC}_{k\text{-int}}(\text{Enc}_{k\text{-enc}}(m))$
 - Encrypt then MAC
 - IPSec (ESP), Signal (TextSecure ProtocolV2), probably TLS 1.3 [RCF7366]
 - Considered the most secure

Secure Channel

MAC-then-Enc vs. Enc-then-MAC

Secure Channel Implementation

Secure Channel (ESP) in the OpenBSD Kernel

Authenticated Encryption With Associated Data

AEAD & Secure Channel (ESP) in the Linux Kernel

Attacks against a Secure Channel (Stream Cipher)

Attacks against a Secure Channel (Padding oracle)

- Our Secure Channel Implementation:
 - We need
 - Message numbering
 - Authentication
 - Encryption
 - Our Toy Implementation
 - Message numbering: n (next slide)
 - Authentication: HMAC-SHA-256
 $MAC_{k-int}(n \parallel IV \parallel c)$
 - Encryption: AES-128-CTR
 $c \leftarrow ENC_{k-enc}(IV, m)$
 - 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

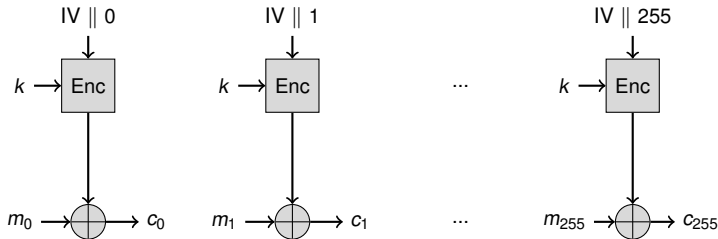
- Initialize (at Alice):

```
# Output: 128bit key
def KDF(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 = KDF(k || "Enc Alice to Bob")
    K_recv_enc = KDF(k || "Enc Bob to Alice")
    K_send_int = KDF(k || "MAC Alice to Bob")
    K_recv_int = KDF(k || "MAC Bob to Alice")
    n_send = 1
    n_recv = 0
    used_nonces = {}
```

- Generate one key for each purpose
- Where `· || ·` means string/byte concatenation

- AES-128-CTR Mode needs IV:
 - $ctr_i = IV \parallel i$
 - 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 $i \in \{0 \dots 254\}$: $ctr_{i+1} = ctr_i + 1$

- 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 → 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
```

Secure Channel

MAC-then-Enc vs. Enc-then-MAC

Secure Channel Implementation

Secure Channel (ESP) in the OpenBSD Kernel

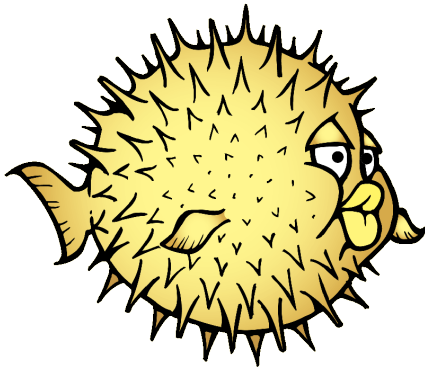
Authenticated Encryption With Associated Data

AEAD & Secure Channel (ESP) in the Linux Kernel

Attacks against a Secure Channel (Stream Cipher)

Attacks against a Secure Channel (Padding oracle)

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;
    }
}
```

int checkreplaywindow(struct tdb *tdb, u_int32_t seq, u_int32_t *seqh, int commit) i.e. do not update replay window

```
/* 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 && esp_x ? 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);  
}
```

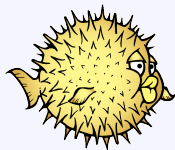
- Dispatch to crypto driver (similar to Linux)
- A callback will be called once the crypto was done

```
/*
 * 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

Secure Channel

MAC-then-Enc vs. Enc-then-MAC

Secure Channel Implementation

Secure Channel (ESP) in the OpenBSD Kernel

Authenticated Encryption With Associated Data

AEAD & Secure Channel (ESP) in the Linux Kernel

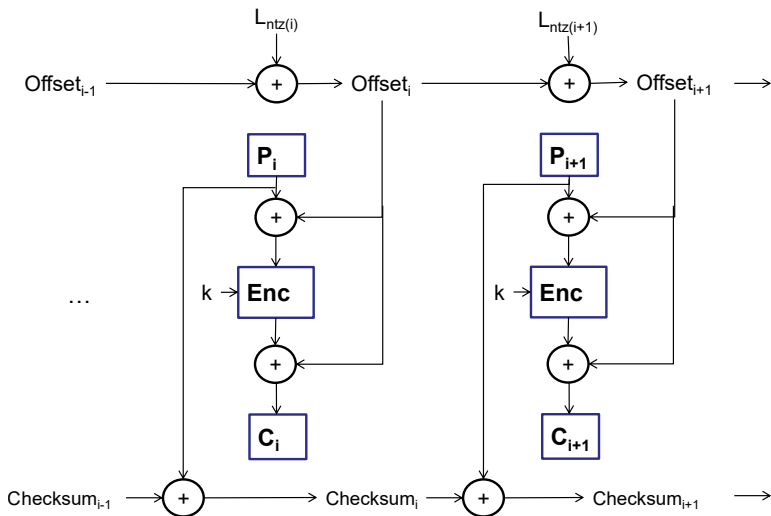
Attacks against a Secure Channel (Stream Cipher)

Attacks against a Secure Channel (Padding oracle)

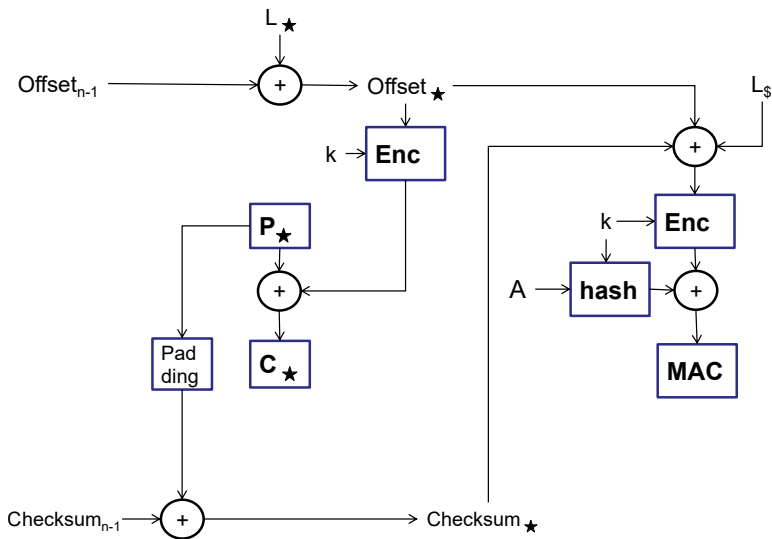
- Authenticated Encryption With Associated Data (AEAD):
 - Authenticated encryption: Encrypt then MAC
 - Associated Data: Additional non-encrypted data but authenticated
 - Example AD: IV, information necessary for message routing, ...
 - Special AEAD Algorithms: only need one pass over the data
 - Encrypt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

- Offset Codebook Mode
 - Authenticated Encryption Mode
 - Proposed 2001 [OCB1]
 - Standardized May 2014 [RFC 7253]
 - Encryption
 - Inspired by ECB with block-dependent offsets (avoids ECB problems!)
 - Associated Data A
 - A is not encrypted but authenticated
 - For example: Unencrypted header data
 - MAC
 - Checksum = XOR over plaintext, length- and key-dependent variables
 - MAC = (Encryption of checksum with shared key k) XOR (hash(k, A))
 - Requires only one key K for encryption and authentication
 - Requires a fresh nonce every time

- Let double be multiplication by the variable in the OCB Galois Field
- Variables depending on the key: $L_*, L_{\$}, L_0, L_1, L_2, \dots$
 - $L_* = \text{Enc}_K(0)$
 - $L_{\$} = \text{double}(L_*)$
 - $L_0 = \text{double}(L_{\$})$
 - $L_i = \text{double}(L_{i-1})$
- Let ntz be number of trailing zeros (zero bits at the end)
- Usage of the L 's
 - $L_{\$}$ \rightarrow MAC
 - L_* \rightarrow last block
 - $L_{\text{ntz}(i)}$ \rightarrow intermediate blocks
- Note: $L_{\text{ntz}(i)}$ is used
 - Only few L_i are needed (for a fixed K)
 - They can be pre-computed and stored in a Lookup table



- $Offset_0$ depends on the key and the nonce
- “It is crucial that, as one encrypts, one does not repeat a nonce.” [RFC 7253, §5.1]
- Nonce *may* not be random, e.g. a counter works fine
- A new nonce for every authenticated encryption API call is needed!
- Details about the initialization: <http://www.cs.ucdavis.edu/~rogaway/ocb/ocb-faq.htm>

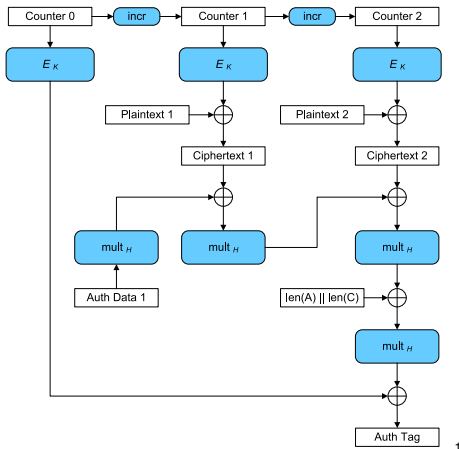


- Question: XOR plaintext and then encrypt, that sounds like the weak MAC example from Chapter 2.2. Why is OCB more secure than the easy-to-break example?

- Question: XOR plaintext and then encrypt, that sounds like the weak MAC example from Chapter 2.2. Why is OCB more secure than the easy-to-break example?
- “OCB enjoys provable security: the mode of operation is secure assuming that the underlying blockcipher is secure. As with most modes of operation, security degrades as the number of blocks processed gets large” [RFC 7253]

- Galois/Counter Mode (GCM)
 - Developed by John Viega and David A. McGrew
 - Standardized by NIST in 2007, IETF standards for cipher suites with AES-GCM for TLS (SSL) and IPsec exist.
 - Follows the Encrypt-then-MAC concept
 - Combines concept of Counter Mode for encryption with Galois Field Multiplication to compute MAC on the ciphertext
 - $GF(2^{128})$ based on polynomial $x^{128} + x^7 + x^2 + x + 1$

- Galois/Counter Mode (GCM)
 - Developed by John Viega and David A. McGrew
 - Standardized by NIST in 2007, IETF standards for cipher suites with AES-GCM for TLS (SSL) and IPsec exist.
 - Follows the Encrypt-then-MAC concept
 - Combines concept of Counter Mode for encryption with Galois Field Multiplication to compute MAC on the ciphertext
 - $GF(2^{128})$ based on polynomial $x^{128} + x^7 + x^2 + x + 1$
- Definitions
 - H is $Enc(k,0)$
 - Auth Data is data not to be encrypted. GCM generates check value by XOR and GF multiplication with H for each block.
 - For the MAC, this process continues on the ciphertext and a length field in the end.



- Counter 0 = IV, Auth Tag = MAC

¹ Image Source = https://en.wikipedia.org/wiki/Galois/Counter_Mode

- In a Galois Field we consider the bitstring to represent a polynomial.
 - E.g. $1011 = x^3 + x + 1$
- As a consequence Galois Field Multiplication is based on polynomial multiplication modulus the polynomial of the field.

- In a Galois Field we consider the bitstring to represent a polynomial.
 - E.g. $1011 = x^3 + x + 1$
- As a consequence Galois Field Multiplication is based on polynomial multiplication modulus the polynomial of the field.
- Example: In $GF(2^{128})$ based on polynomial $g(x) = x^{128} + x^7 + x^2 + x + 1$
 - $P(x) = x^{127} + x^7$
 - $Q(x) = x^5 + 1$
 - $P(x) * Q(x) = x^{132} + x^{127} + x^{12} + x^7$
 - To compute the modulus, we have to compute a polynomial division $P(x) * Q(x) / g(x)$.
 - We can see that $x^4 * g(x)$ removes the x^{132} , so $P(x) * Q(x) - x^4 * g(x) = x^{127} + x^{12} + x^{11} + x^7 + x^6 + x^5 + x^4$
 - Since this polynomial fits into the 128 bit, this is the remainder of the division, thus the result, in bits: 1000...01100011110000.

Secure Channel

MAC-then-Enc vs. Enc-then-MAC

Secure Channel Implementation

Secure Channel (ESP) in the OpenBSD Kernel

Authenticated Encryption With Associated Data

AEAD & Secure Channel (ESP) in the Linux Kernel

Attacks against a Secure Channel (Stream Cipher)

Attacks against a Secure Channel (Padding oracle)

AEAD & ESP in the Linux Kernel

Linux 4.3 (stable, vanilla)



AEAD & Secure Channel (ESP) in the Linux Kernel



- 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

AEAD & Secure Channel (ESP) in the Linux Kernel



- 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

AEAD & Secure Channel (ESP) in the Linux Kernel



- 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

Secure Channel

MAC-then-Enc vs. Enc-then-MAC

Secure Channel Implementation

Secure Channel (ESP) in the OpenBSD Kernel

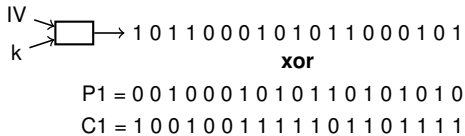
Authenticated Encryption With Associated Data

AEAD & Secure Channel (ESP) in the Linux Kernel

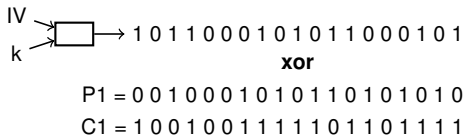
Attacks against a Secure Channel (Stream Cipher)

Attacks against a Secure Channel (Padding oracle)

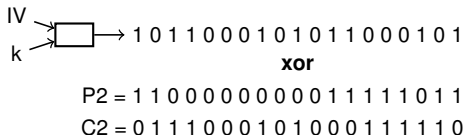
- Re-Use of Initialization Vector (IV):



- Re-Use of Initialization Vector (IV):



- Then some time later the same IV is used again:



- Re-Use of Initialization Vector (IV) continued:

$$\begin{aligned} C1 &= 1\ 0\ 0\ 1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 1\ 1 \\ C2 &= 0\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 0 \\ C1 + C2 &= 1\ 1\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 0\ 0\ 1 \\ &== \quad \quad \quad (\rightarrow P1+P2=C1+C2) \\ P1 + P2 &= 1\ 1\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 0\ 0\ 1 \\ P1 &= 0\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0 \\ P2 &= 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 0\ 1\ 1 \end{aligned}$$

- As we see from the example, the attacker can compute $C1+C2$ because he observes $C1$ and $C2$, but that means he knows also $P1+P2$.
- Known Plaintext (e.g. $P1$) \rightarrow attacker can compute other plaintext
- Statistical properties of plaintext can be used if plaintext is not random-looking. That means if entropy of $P1+P2$ is low.

Secure Channel

MAC-then-Enc vs. Enc-then-MAC

Secure Channel Implementation

Secure Channel (ESP) in the OpenBSD Kernel

Authenticated Encryption With Associated Data

AEAD & Secure Channel (ESP) in the Linux Kernel

Attacks against a Secure Channel (Stream Cipher)

Attacks against a Secure Channel (Padding oracle)

- Passwords
 - N: size of alphabet (number of different characters)
 - L: length of password in characters
- Complexity of guessing a randomly-generated password / secret
 - The assumption is, we generate a password and then we test it.
→ $\mathcal{O}(N^L)$
- Complexity of guessing a randomly-generated password character by character
 - The assumption is that we can check each character individually for correctness.
 - For each character it is $N/2$ (avg) and N (worst case)
 - So, overall $L * N/2$ (avg)
- In the subsequent slides we will show an attack that reduces the decryption of a blockcipher in CBC mode to byte-wise decryption (under special assumptions).

P	MAC
Ciphertext	

- Operation
 - P and MAC are encrypted and hidden in the ciphertext.
 - Receiver
 - Decrypts P
 - Decrypts MAC
 - Computes and checks MAC → MAC error or success

P	MAC
Ciphertext	

- Operation
 - P and MAC are encrypted and hidden in the ciphertext.
 - Receiver
 - Decrypts P
 - Decrypts MAC
 - Computes and checks MAC → MAC error or success
- Consequence
 - MAC does not protect the ciphertext.
 - Integrity check can only be done once everything is decrypted.
 - As a consequence, receiver will detect malicious messages at the end of the secure channel processing and not earlier.
 - But is that more than a performance issue? Well, yes.

- If we use a block cipher, we have to ensure that the message encoding fits to the blocksize of the cipher.

P	Pad	MAC
Ciphertext		

- Encode-then-MAC-then-Encrypt:
 - Format P so that with the MAC added the encryption sees the right size.
 - Needs that we know the size of the MAC and blocksize of cipher when generating P | Padding.

- If we use a block cipher, we have to ensure that the message encoding fits to the blocksize of the cipher.

P	Pad	MAC
Ciphertext		

- Encode-then-MAC-then-Encrypt:

- Format P so that with the MAC added the encryption sees the right size.
- Needs that we know the size of the MAC and blocksize of cipher when generating P | Padding.

P	MAC	Pad
Ciphertext		

- MAC-then-Encode-then-Encrypt:

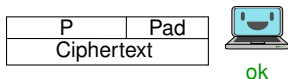
- Used in TLS/SSL
- Here, we add the MAC first and then pad the P | MAC to the correct size.
- How do we know what is padding and what not? Padding in TLS/SSL:
 - If size of padding is 1 byte, the padding is 1.
 - If size of padding is 2 bytes, the padding is 2 2.
 - If size of padding is 3 bytes, the padding is 3 3 3.
 - ...

- In ancient times, people asked oracles for guidance.

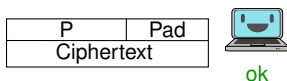
- In ancient times, people asked oracles for guidance.
- In computer science, oracles are functions that give as cheaply access to information that would otherwise be hard to compute.
 - E.g. $\mathcal{O}(1)$ cost to ask specific NP-complete question \rightarrow polynomial hierarchy

- In ancient times, people asked oracles for guidance.
- In computer science, oracles are functions that give as cheaply access to information that would otherwise be hard to compute.
 - E.g. $\mathcal{O}(1)$ cost to ask specific NP-complete question \rightarrow polynomial hierarchy
- In cryptography, an attacker can trigger some participant \mathcal{O} in a protocol or communication to leak information that might or might not be useful.
 - Participant \mathcal{O} may re-encrypt some message fragment
 - Participant \mathcal{O} responds with an error message explaining what went wrong
 - Response time of participant \mathcal{O} may indicate where error happened
 - Response time may leak information about key if processing time depends (enough) on which bits are set to 1.
 - More obvious for the computationally expensive public key algorithms, but implementations of symmetric ciphers have also been attacked.

- Side Channel Attacks
 - A general class of attacks where the attacker gains information from aspects of the physical implementation of a cryptosystem.
 - Can be based on: Timing, Power Consumption, Radiation, . . .



- Side Channel Attacks
 - A general class of attacks where the attacker gains information from aspects of the physical implementation of a cryptosystem.
 - Can be based on: Timing, Power Consumption, Radiation, . . .

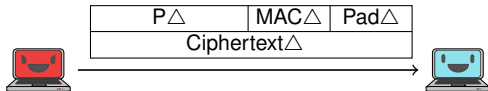


- Padding Oracle
 - The oracle tells the attacker if the padding in the message was correct.
 - This may be due to a message with the information.
 - It can also be due to side channel like the response time.

- Attacker sees unknown ciphertext $C =$

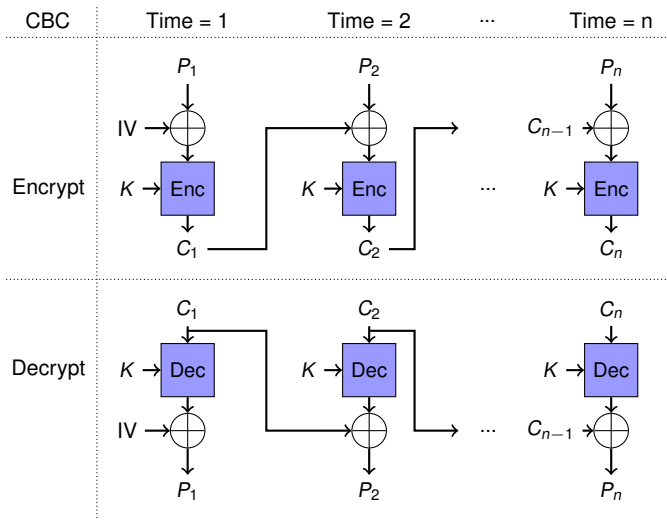
P	MAC	Pad
Ciphertext		

 that was sent from Alice to Bob
- To decrypt the ciphertext, the attacker modifies C and sends it to Bob.



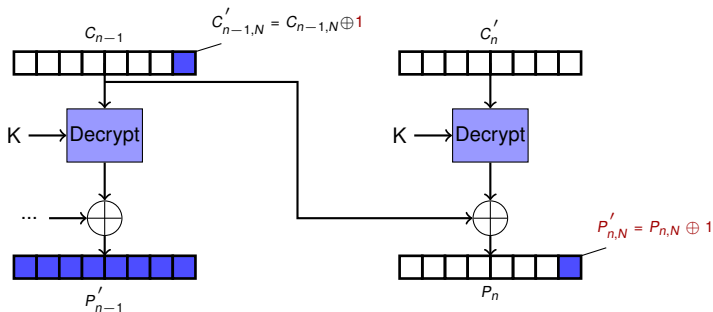
- It is unlikely that the MAC and padding are correct. So, Bob will send an error back to Alice (and the attacker).
- In earlier versions of TLS, Bob sent back different error messages for padding errors and for MAC errors.

- Encryption and Decryption in CBC mode

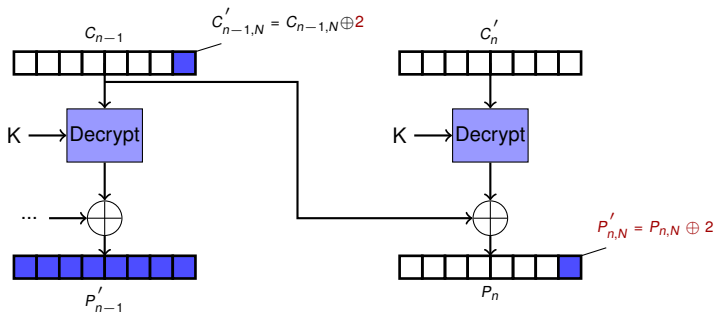


- Assumptions:
 - Attacker got hold of a ciphertext C (n blocks, N bytes per block)
 - C was protected with Encryption in CBC mode used in MAC-then-Encode-then-Encrypt mode.
 - For padding PKCS7 was used (padding of 1 byte: $\text{pad} = 1$, padding 2 bytes: $\text{pad} = 2, 2, \dots$)
 - An oracle replies to sent ciphertexts with error messages:
 - Padding error if padding doesn't match (checked before MAC).
 - MAC error if padding fits but MAC is wrong.
- Goal: Decrypt the complete ciphertext using the oracle.
- Approach:
 - Start decrypting the last byte of the last block $P_{n,N}$ by altering $C_{n-1,N}$ and sending the resulting ciphertext C' to the oracle.
 - When the oracle replies with a MAC error $P_{n,N}$ can be calculated (see following slides).

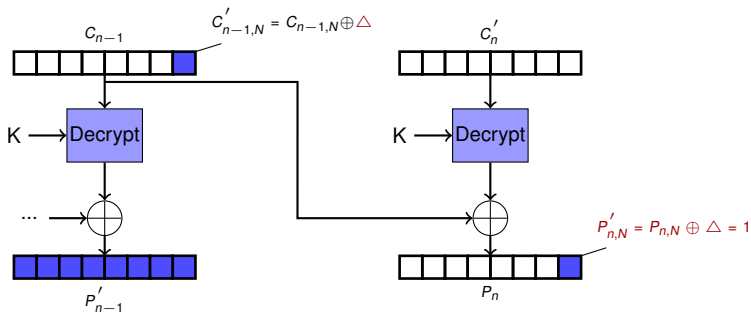
- Change the last byte of the original ciphertext block C_{n-1} by XORing it with a chosen Δ : $C'_{n-1,N} = C_{n-1,N} \oplus \Delta$. Then send C' to the oracle.
- **Padding error returned:**
 - Try again using a new Δ (max of 256 tries needed).



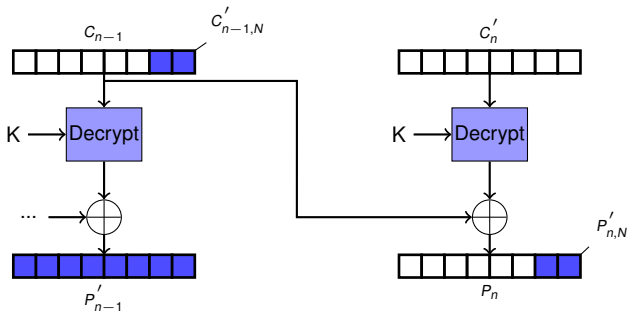
- Change the last byte of the original ciphertext block C_{n-1} by XORing it with a chosen Δ : $C'_{n-1,N} = C_{n-1,N} \oplus \Delta$. Then send C' to the oracle.
- **Padding error returned:**
 - Try again using a new Δ (max of 256 tries needed).



- Change the last byte of the original ciphertext block C_{n-1} by XORing it with a chosen Δ : $C'_{n-1,N} = C_{n-1,N} \oplus \Delta$. Then send C' to the oracle.
- Padding error returned:
 - Try again using a new Δ (max of 256 tries needed).
- **MAC error:**
 - padding was fine $\rightarrow P'_{n,N} = 1$ (since a padding size of 1 byte means padding = $P'_{n,N} = 1$)
 - correct padding means the last byte was 1 $\rightarrow P'_{n,N} = P_{n,N} \oplus \Delta = 1 \rightarrow P_{n,N} = 1 \oplus \Delta$

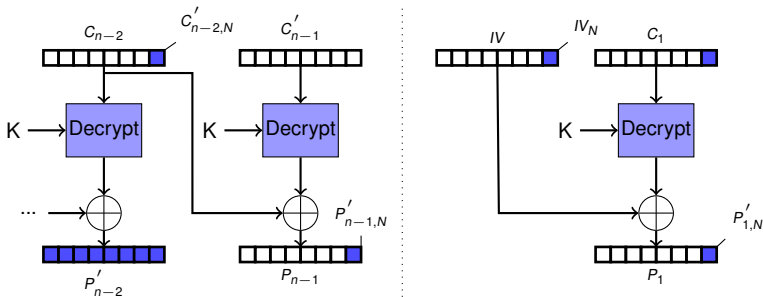


- Now we want to decrypt $P_{n,N-1}$. For that a padding of length 2 is needed.
- Since $P_{n,N}$ is known, we can calculate $C'_{n-1,N}$ so that $P'_{n,N} = 2$
 - $P_{n,N} \oplus C'_{n-1,N} = 2 \rightarrow C'_{n-1,N} = P_{n,N} \oplus 2$
- Now find $C'_{n-1,N-1}$ that satisfies $C'_{n-1,N-1} \oplus P_{n,N-1} = 2$



- As before, we need to try up to 256 values, all values except for the correct one generate a padding error. The correct one produces a MAC error. \rightarrow We know $P_{n,N-1}$

- To completely decrypt C_n we have to repeat the procedure until all bytes of the block are decrypted. In the figure with 8 bytes per block, the last padding we generate is 8 8 8 8 8 8 8 8.
- To decrypt C_{n-1} we can cut off C_n and repeat the same procedure with C_{n-1} as last block. For decrypting C_1 we can use the IV as ciphertext for the attack modifications.



- The attack was against CBC mode used in MAC-then-Encode-then-Encrypt mode.
 - Padding Oracle attack known long in cryptography.
 - Mode still used in SSL / TLS. Hacks have utilized that. However, defenses have been added.
- CBC with Encode-then-Encrypt-then-MAC does not have this vulnerability.
 - Because MAC check would fail first, process would be aborted, and padding problems would then not be leaked.

- Bell95** M. Bellare and P. Rogaway, Provably Secure Session Key Distribution - The Three Party Case, Proc. 27th STOC, 1995, pp 57–64
- Boyd03** Colin Boyd, Anish Mathuria, “Protocols for Authentication and Key Establishment”, Springer, 2003
- Bry88a** R. Bryant. Designing an Authentication System: A Dialogue in Four Scenes. Project Athena, Massachusetts Institute of Technology, Cambridge, USA, 1988.
- Diff92** W. Diffie, P. C. van Oorschot, and M. J. Wiener. Authentication and authenticated key exchanges. Designs, Codes, and Cryptography, 1992
- Dol81a** D. Dolev, A.C. Yao. On the security of public key protocols. Proceedings of IEEE 22nd Annual Symposium on Foundations of Computer Science, pp. 350-357, 1981.
- Fer00** Niels Ferguson, Bruce Schneier, “A Cryptographic Evaluation of IPsec”.
<http://www.counterpane.com/ipsec.pdf> 2000
- Fer03** Niels Ferguson, Bruce Schneier, „Practical Cryptography“, John Wiley & Sons, 2003
- Gar03** Jason Garman, “Kerberos. The Definitive Guide”, O’Reilly Media, 1st Edition, 2003
- Kau02a** C. Kaufman, R. Perlman, M. Speciner. Network Security. Prentice Hall, 2nd edition, 2002.
- Koh94a** J. Kohl, C. Neuman, T. T’so, The Evolution of the Kerberos Authentication System. In Distributed Open Systems, pages 78-94. IEEE Computer Society Press, 1994.

- Mao04a** W. Mao. Modern Cryptography: Theory & Practice. Hewlett-Packard Books, 2004.
- Nee78** R. Needham, M. Schroeder. Using Encryption for Authentication in Large Networks of Computers. Communications of the ACM, Vol. 21, No. 12, 1978.
- Woo92a** T.Y.C Woo, S.S. Lam. Authentication for distributed systems. Computer, 25(1):39-52, 1992.
- Lowe95** G. Lowe, „An Attack on the Needham-Schroeder Public-Key Authentication Protocol“, Information Processing Letters, volume 56, number 3, pages 131- 133, 1995.
- OCB1** Rogaway, P., Bellare, M., Black, J., and T. Krovetz, "OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption", ACM Conference on Computer and Communications Security 2001 - CCS
- OCB** T.Krovetz, P. Rogaway, „The OCB AuthenticatedEncryption Algorithm“
<http://tools.ietf.org/html/draft-irtf-cfrg-ocb-03>
- RFC 4106** The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP)
- RFC 5288** AES Galois Counter Mode (GCM) Cipher Suites for TLS.
- RFC 7253** The OCB Authenticated-Encryption Algorithm

- RFC2560** M. Myers, et al., “X.509 Internet Public Key Infrastructure Online Certificate Status Protocol – OCSP”, June 1999
- RFC3961** K. Raeburn, “Encryption and Checksum Specifications for Kerberos 5”, February 2005
- RFC3962** K. Raeburn, “Advanced Encryption Standard (AES) Encryption for Kerberos 5”, February 2005
- RFC4757** K. Jaganathan, et al., “The RC4-HMAC Kerberos Encryption Types Used by Microsoft Windows ”, December 2006
- RFC4120** C. Neuman, et al., “The Kerberos Network Authentication Service (V5)”, July 2005
- RFC4537** L. Zhu, et al, “Kerberos Cryptosystem Negotiation Extension”, June 2006
- RFC5055** T. Freeman, et al, “Server-Based Certificate Validation Protocol (SCVP)”, December 2007