# CS915/435 Advanced Computer Security
# - Elementary Cryptography

Message Authentication Code

# Roadmap

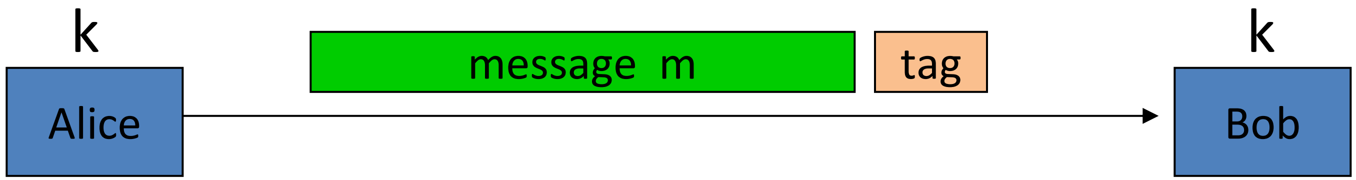
- Symmetric cryptography
  - Classical cryptographic
  - Stream cipher
  - Block cipher I, II
  - Hash
  - **MAC**
- Asymmetric cryptography
  - Key agreement
  - Public key encryption
  - Digital signature

# Message Integrity

- Goal: **Integrity**, not confidentiality

- Examples:
  - Protecting binaries of an Operating System
  - Protecting banner ads on web pages

# Message integrity: MACs




**Generate tag:**

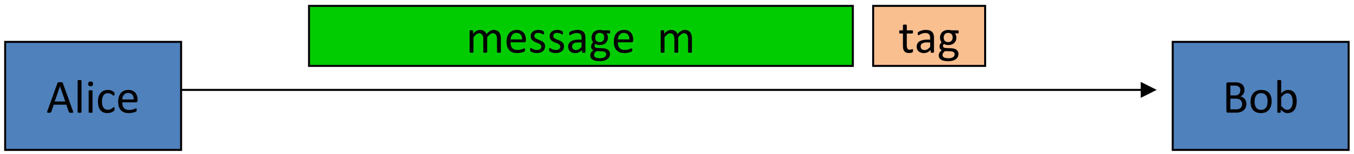
**tag $\leftarrow$ S(k, m)**

**Verify tag:**

**V(k, m, tag) $\stackrel{?}{=}$ `yes'**

- Def: MAC I = (S, V) defined over (K,M,T)
  - S(k,m) outputs t in T
  - V(k,m,t) outputs “Yes” or “No”

# Integrity requires a secret key




**Generate tag:**

**tag ← CRC(m)**

**Verify tag:**

**V(m, tag) $\stackrel{?}{=}$ `yes'**

- Cyclic Redundancy Code (CRC)
  - Designed to detect **random** errors, not malicious errors.
- CRC cannot be used for integrity check
  - Attacker can easily modify message m and re-compute CRC

# Secure MACs

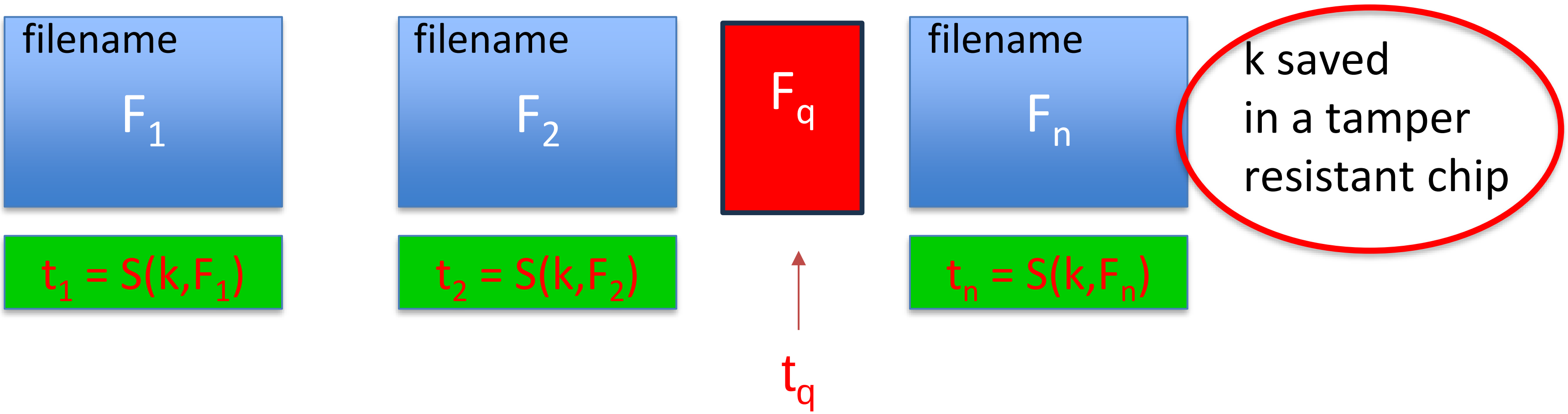
- Attacker's power: **chosen message attack**
  - For $m_1,m_2,\ldots,m_q$ attacker is given $t_i \leftarrow S(k,m_i)$

- Attacker's goal: **existential forgery**
  - Produce some new valid message/tag pair $(m,t)$

$$(m,t) \notin \{ (m_1,t_1) , \ldots , (m_q,t_q) \}$$

# Example: protecting system files

Suppose at install time the system computes:

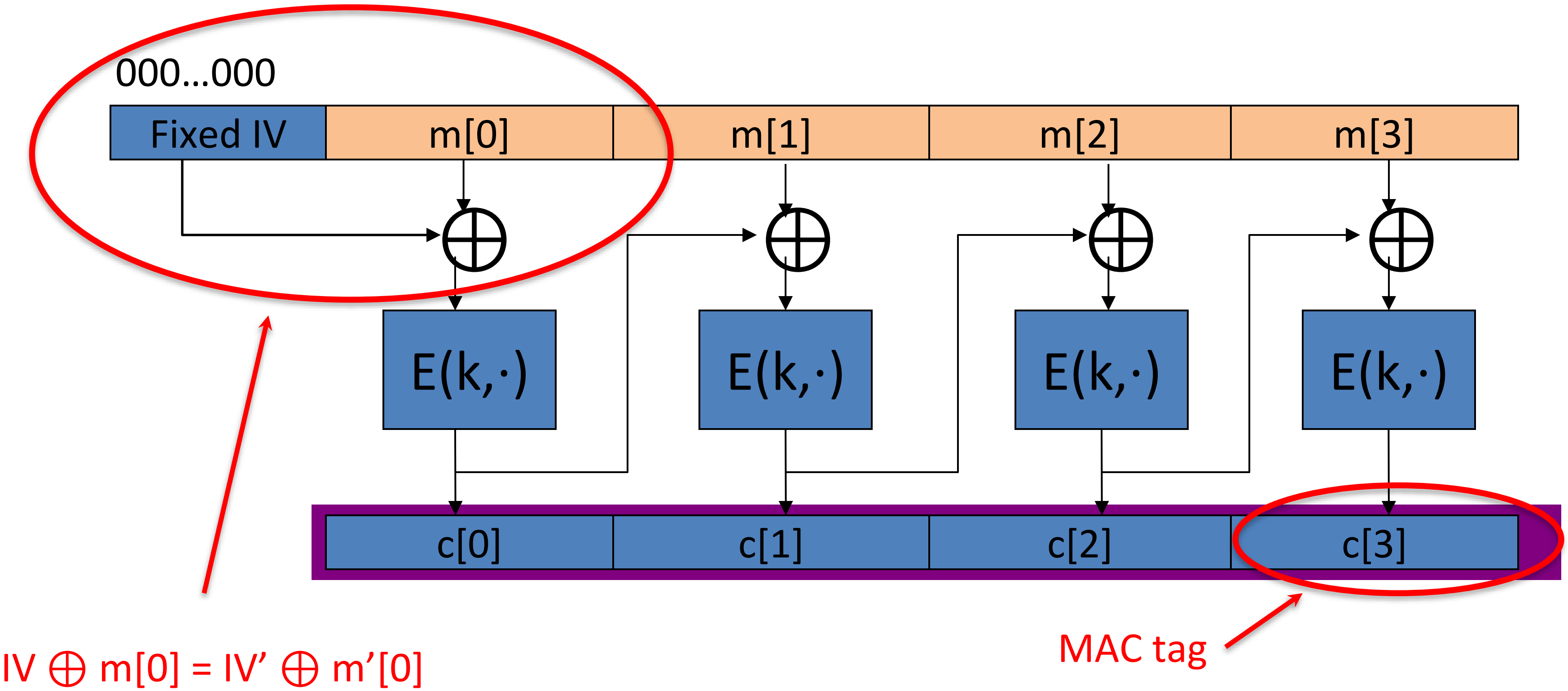



- Later a virus infects system and modifies system files
  - If MAC is secure, the virus cannot forge a valid tag for itself
- User reboots into clean OS
  - All modified files will be detected by the chip

# How to construct a MAC?

- In general, two approaches
  1. Based on a block cipher (e.g., CBC-MAC)
  2. Based on a hash function (e.g., HMAC)

# Construction 1: CBC-MAC

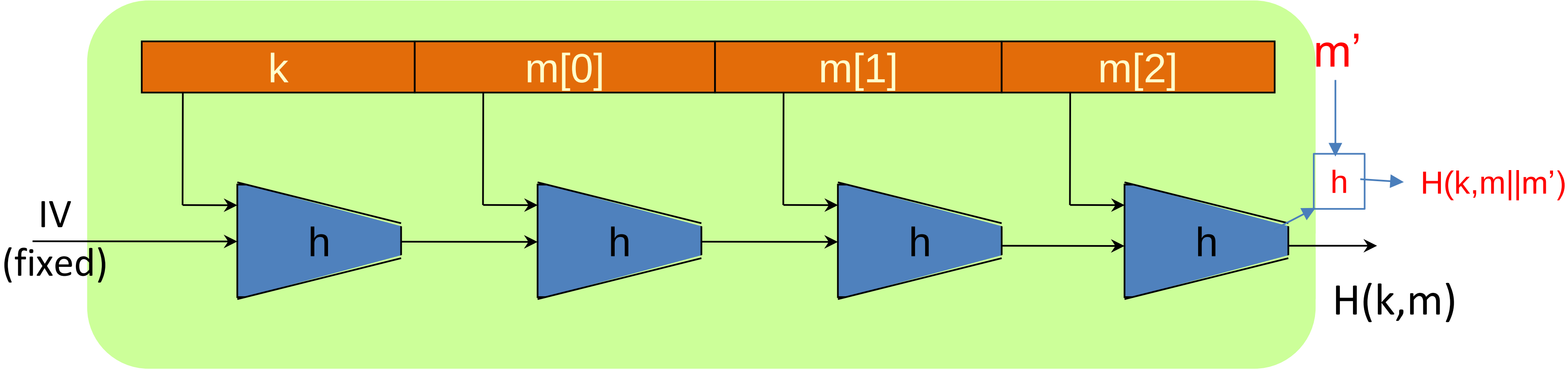



- Question
  - Why can't you use a random IV?

# Construction 2: hash-based MAC

- Build a MAC based on a hash function
  - Say using SHA-256
- An example of a construction
  - Assume a key k and message m
  - Construct a MAC by using: **H(k||m)**
- Is this secure?

# Insecure MAC: H(k, m)



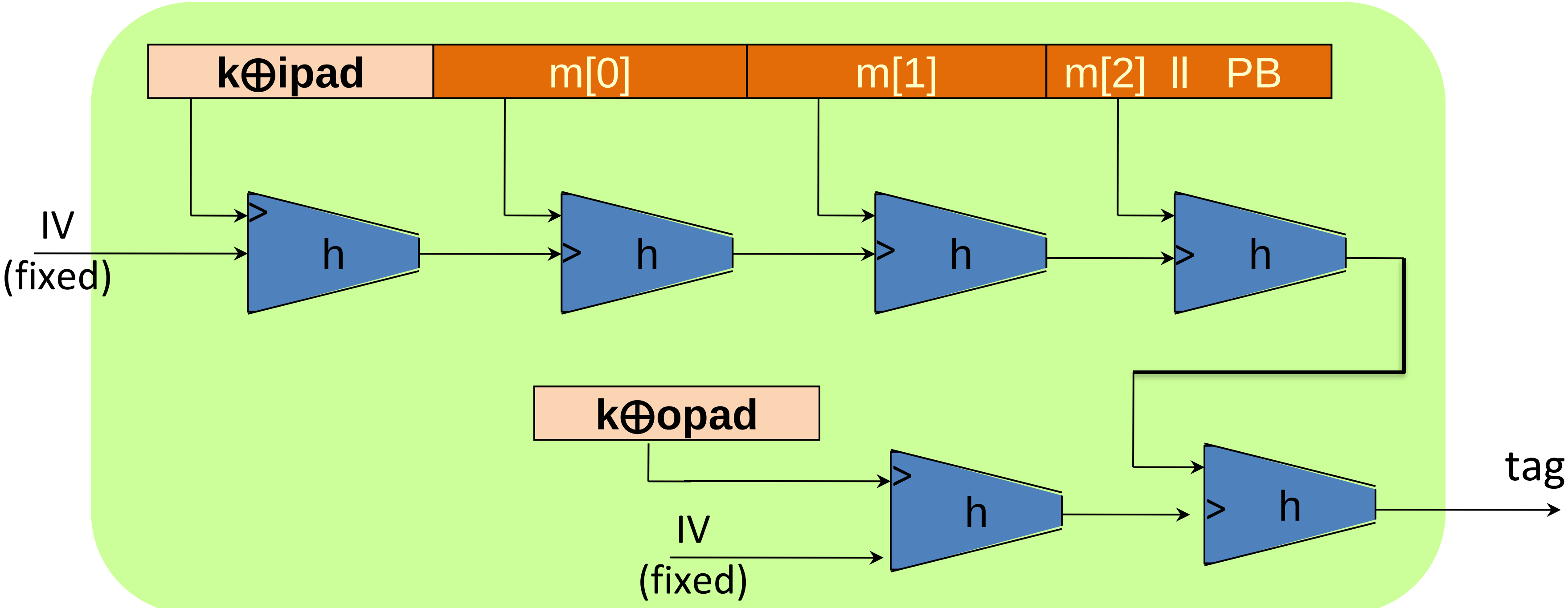

- Suppose MAC tag = H(k, m[0]||m[1]||m[2])
  - Mallory can easily append another block m[3] and compute H(k, m[0]||m[1]||m[2]||m[3])
- An alternative construction MAC tag = H(m, k)
  - Still insecure

# HMAC

- Basic intuition
  - Need to have a secret key to protect the front
  - Need to have a secret key to protect the end
- An example of a secure construction
  - **H(k1, H(k2, m))** where k1 and k2 are two different keys
- HMAC
  - Use only one secret key (hence efficient)
  - Define ipad and opad as constants

$$\text{HMAC}(k,m) = H(\, k \oplus \text{opad} \,\|\, \mathbf{H(\, k \oplus ipad \,\|\, m\,)}\,)$$

# HMAC in picture




HMAC(k,m) = H( k⊕opad ll **H( k⊕ipad ll m )**)

# Verification Timing attacks on HMAC

Example: Keyczar crypto library (Python)

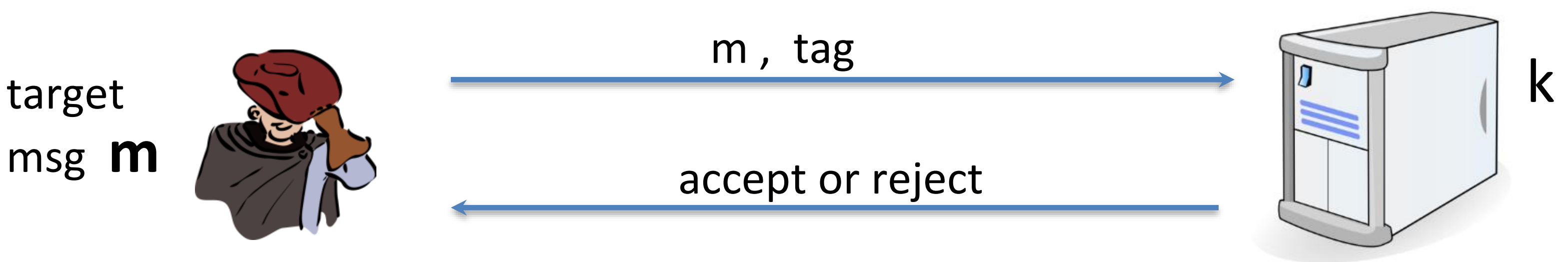
```
def Verify(key, msg, sig_bytes):
    return HMAC(key, msg) == sig_bytes
```

The problem: '==' implemented as a byte-by-byte comparison

- Comparator returns false when first inequality found

http://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library/

# How does it work?



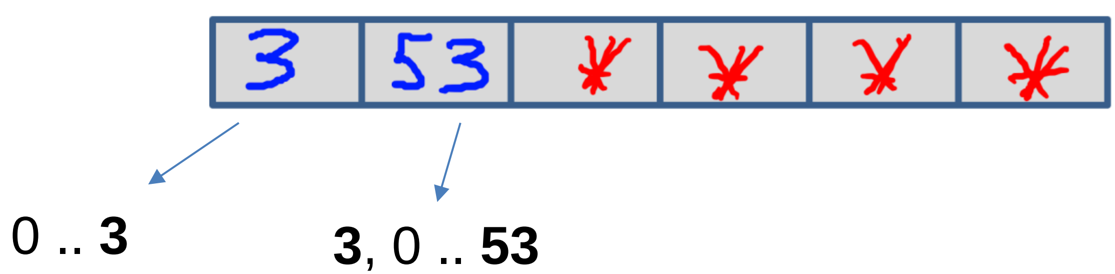

Timing attack: to compute tag for target message m do:

Step 1: Query server with random tag

Step 2: Loop over all possible first bytes and query server.

Stop when verification takes a little longer than step 1

Step 3: repeat for all tag bytes until valid tag found



# Defense #1

Make string comparator always take same time (Python) :

```
return false if  sig_bytes  has wrong length
result = 0
for x, y in zip( HMAC(key,msg) , sig_bytes):
    result  |= ord(x) ^ ord(y)
return result == 0
```

Can be difficult to ensure due to optimizing compiler.

# Defense #2

Make string comparator always take same time (Python) :

```
def Verify(key, msg, sig_bytes):
    mac = HMAC(key, msg)
    return HMAC(key, mac) == HMAC(key, sig_bytes)
```

Attacker doesn't know values being compared

Lesson

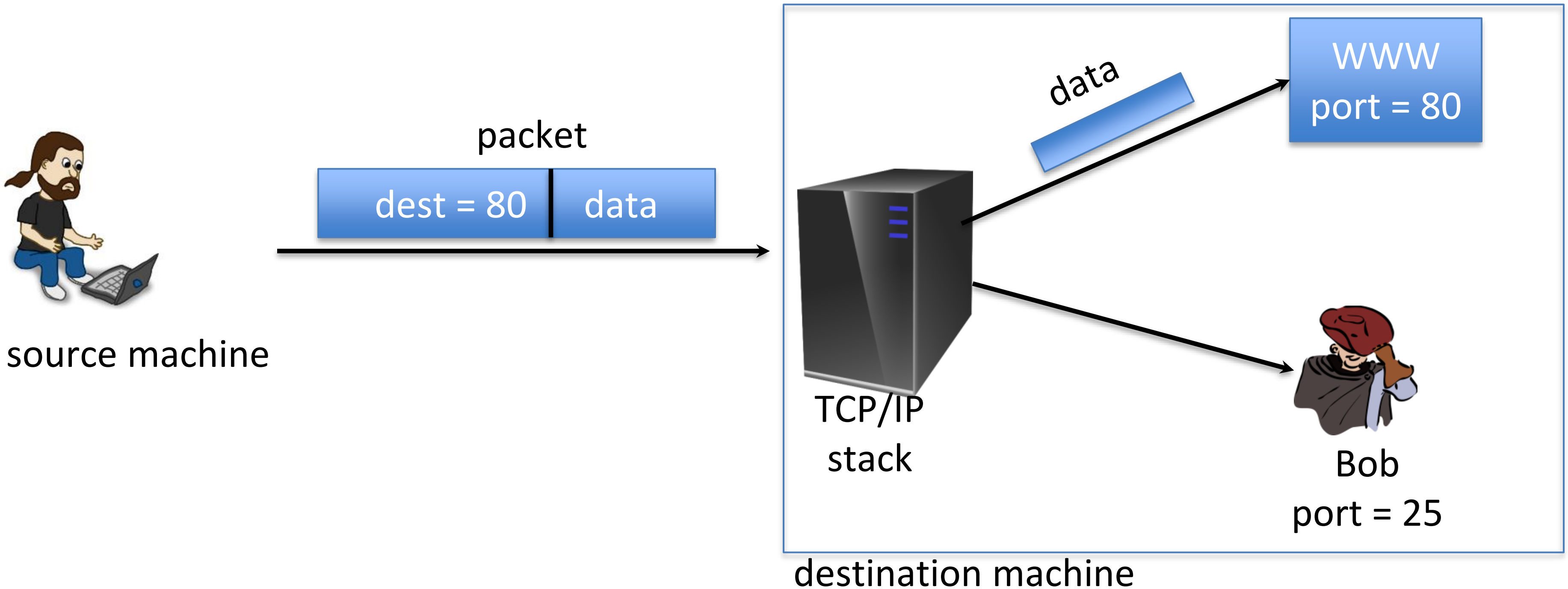
- **An attacker often bypasses cryptography and exploits weaknesses in the implementation**

# Authenticated encryption

- In the real-world security, encryptions are often done in **authenticated** mode
  - Produce a MAC as part of the encryption process
  - Provide both confidentiality and integrity
- Examples of authenticated encryption
  - CBC mode encryption + CBC-MAC
  - Counter mode encryption + CBC-MAC (IEEE802.11i)

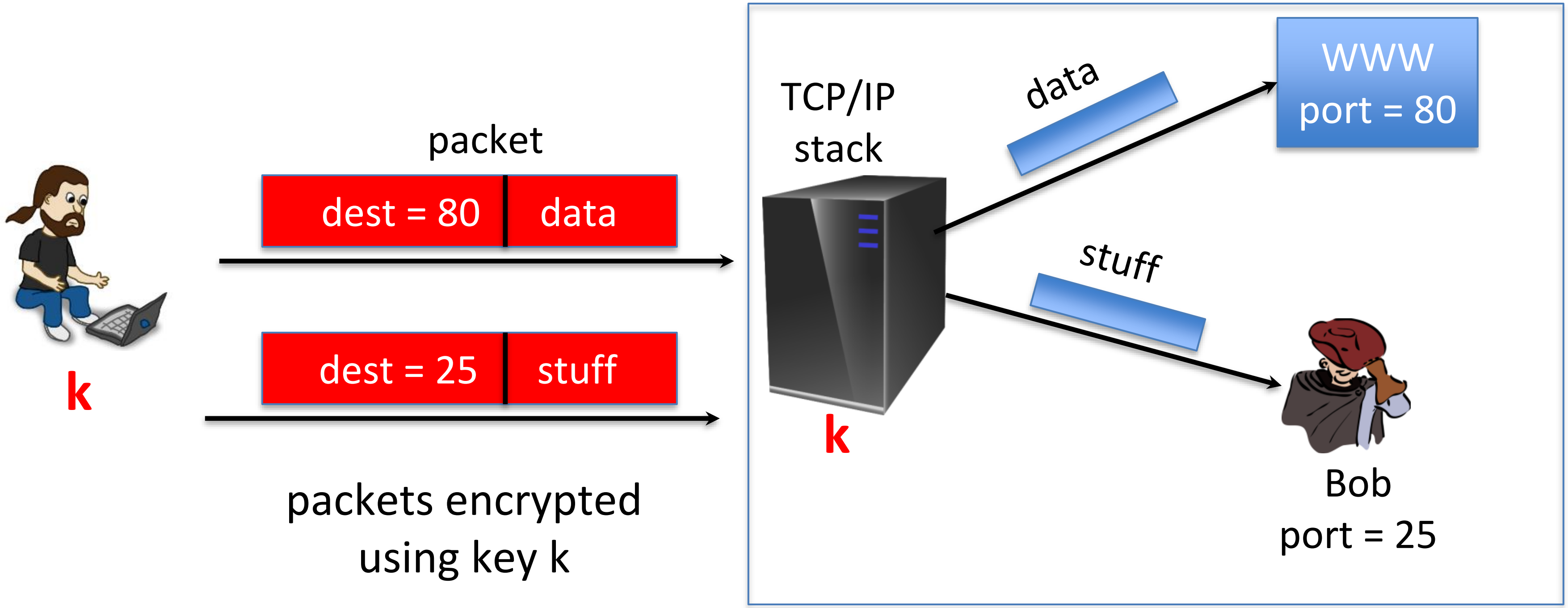
# Sample tampering attacks

TCP/IP: (highly abstracted)



# Sample tampering attacks

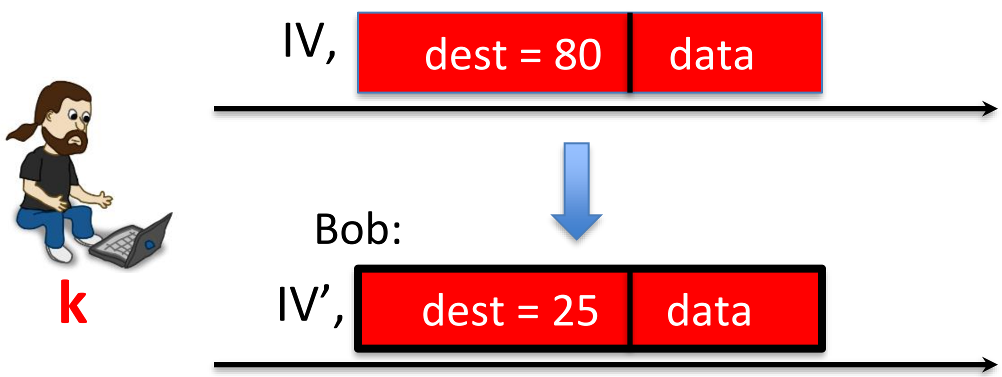
IPsec: (highly abstracted)

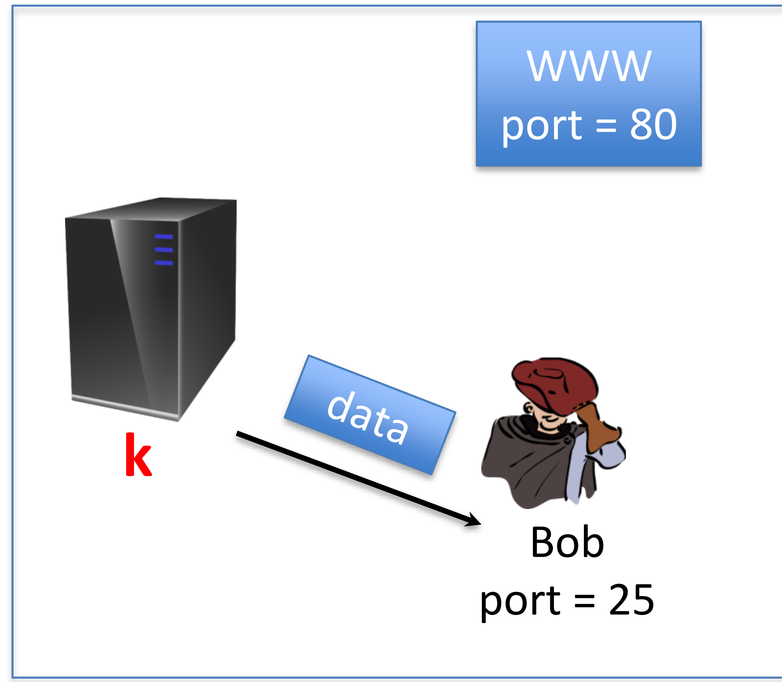


# Reading someone else's data

Note: attacker obtains decryption of any ciphertext

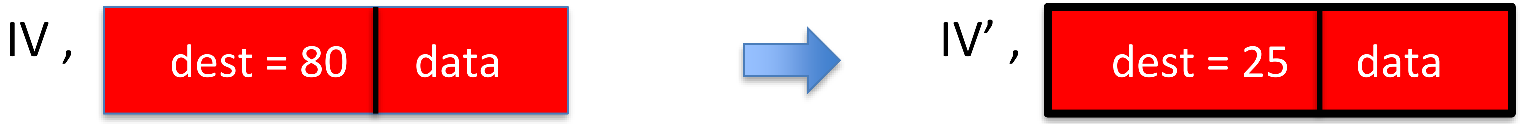
beginning with "dest=25"




Easy to do for CBC with random IV

(only IV is changed)




Encryption is done with CBC with a random IV.

What should IV' be? $m[0] = D(k, c[0]) \oplus IV$ = "dest=80..."

a) $IV' = IV \oplus (\ldots 25 \ldots)$
b) $IV' = IV \oplus (\ldots 80 \ldots)$
c) $IV' = IV \oplus (\ldots 80 \ldots) \oplus (\ldots 25 \ldots)$
d) It can't be done