At first, we will briefly review the Armor64 library, which is utilized in lab 1. Next, we discuss the necessity of integrity when transmitting data over a channel potentially containing malicious entities. The overall idea behind the concept of integrity involves passing a representative piece of the actual message, along with the message, to ensure its integrity. Having to perform this task with regular hash functions leaves a gap in the desired security implementation, which brings us to Secure Hash Functions (SHF’s) and Message Authentication Codes.

1 Armor64

1.1 Purpose

Armor64 is used to encode binary data back to somewhat readable ASCII characters. This is used not only for writing out results of bit-level data manipulation, but also provides an easy way to debug the output of sections of your program.

1.2 How It Works

Data is encoded by transforming 6-bit sections of the input data into 8-bit ASCII values. The process used does not induce any additional security (since the data is only re-encoded differently) and therefore, should not be treated like a cryptographic function.

![Armor64 encoding](image)

Figure 1: Armor64 encoding transforms 6-bit blocks into 8-bit ASCII values

2 Data Integrity

2.1 Cyclic Redundancy Checks

When discussing data validation, the concept of Cyclic Redundancy Checking (CRC) often comes to mind. It is important to note that CRC validation (such as parity bit checking) is intended to
prevent against systematic or hardware problems, **not** against malicious interference. Nonetheless, let’s describe a simple instance of CRC; the *parity check*.

Parity checking on ASCII data is done by setting aside the last bit in an 8-bit construction. Parity checking is done by summing the bit values of the other positions. If the value of the summation is even, the parity bit is set to 1. If, on the other hand, the summation is odd, the parity bit is sent at 0. When the receiver decodes the 7 bits of ASCII data, it performs the same check, and compares the result with the last bit.

### 2.2 CBC mode

CBC (Cipher Block Chaining) provides enhanced security by linking each round of encryption processing with all previous rounds. This helps induce randomness in the input for the next round of the encryption algorithm.

### 2.3 What is Data Integrity?

The purpose of data integrity is to prevent the attacker from **mauling** the data without our knowledge. Note that we don’t care if the data is read, but only if it is modified. Therefore, the type of channel used for the purpose of explaining this concept is somewhat different than those which were previously used. Therefore, these are the two resultant cases for any particular message sent with data integrity:

- Bob knows the message he received hasn’t been tampered with, and accept it
- Bob knows the message was tampered with, and discards it

### 2.4 Channel Information

Both the channel used for data transmission and malicious goal are different than those which were previously used. The channel utilized here is not assumed to be private. In fact, it is irrelevant of the purpose of data integrity. Another important difference in this setup is that Mallory can alter data. Because of this, we need to handle situations where data may be incorrect, malformed, or missing altogether.

![Figure 2: Channel used for communications](image-url)

### 2.5 Use Case Scenario

During the year, you store data on your Linux Lab account. A week before spring break, an email is sent out notifying you that there will be maintenance performed on the Linux Lab computers.
During this maintenance, you don’t know who will have access to your data, or how much access they have (read/write or just read). Since you store homework assignments in your account, you need to ensure these are not altered.

**Solution** You decided to hash each file, then keep a record of the digests. When you return from spring break, you use the same hash function to compare the stored hashes with the freshly taken ones. Any mismatch signifies a problem with the original data, and therefore, the data should be discarded, since it has been modified.

3 Collisions

3.1 Definition

A hash function is defined as:

\[ H : \{0, 1\}^* \rightarrow \{0, 1\}^x \quad \text{where} \quad |H(x)| < |x| \quad \forall x \]

A collision within a hash function’s mapping is shown as follows:

\[ x_1, x_2 \text{ s.t. } H(x_1) = H(x_2) \]

3.2 Why collisions occur

Because the cardinality of the input set is (exponentially) larger than the output set, there will be many collisions when using the hash function. This is shown by application of the pigeon-hole principle. However, in practice, these collisions rarely occur. This is a result of the hash function’s intended design. Hash functions that retain this (and several other properties) are referred to as **Secure Hash Functions (SHF)**.

4 Secure Hash Functions

**Definition 1.** An attempt to define a secure hash function:

\[ H : \{0, 1\}^* \rightarrow \{0, 1\}^x \quad \forall x \in \{0, 1\}^x \quad |H(x)| < |x| \]

if its hard to come by

\[ x_1 \neq x_2 \text{ s.t. } H(x_1) = H(x_2) \]

A more formal definition is:

\[ \forall \text{ efficientattackers } A, \exists \text{ negligible probability} \quad \forall x \in \{0, 1\}^x \quad \Pr[H(x_1) = H(x_2) | (x_1, x_2) \leftarrow A] < \nu \]

where:

- The result is deterministic
- This occurs because collisions exist
- There are efficient attackers against this

We will now discuss the families of Secure Hash Functions.
4.1 CRHF

CRHF (Collision-Resistant Hash Functions) work by providing the attacker with a string of bits. The attacker then must send back to the challenger $x_1$ and $x_2$ such that $x_1 \neq x_2$ but $H_v(x_1) = H_v(x_2)$.

4.2 UOWHF

UOWHF (Universal One-Way Hash Functions) further lock down part of the protocol by requiring the attacker to commit to a particular $x_1$ first. After $x_1$ is chosen, the challenger sends the hash function that will be used to the attacker. The final step is for the attacker to reply with an $x_2$ that causes a collision.

4.3 EUHF

EUHF (Epsilon Universal Hash Functions) requests both $x_1$ and $x_2$ from the attacker before a hash function is chosen and sent to the attacker. This prevents the attacker from knowing ahead-of-time which hashing function will be used.

5 Message Authentication Schemes

5.1 Message Authentication Codes

![Figure 3: Alice and Bob with MACs](image)

**Definition 2.** A *message authentication code* (or *MAC*) is a tuple of probabilistic polynomial-time algorithms $(\text{Gen}), \text{Mac}, \text{Vrfy}$ such that:

1. **Gen:** A *key-generation algorithm* that takes as input the security parameter $1^n$ and outputs a key $k$ with $|k| \geq n$. $k \leftarrow \{0,1\}^n$ at random

2. **Mac:** The *tag-generation algorithm* that takes as input a key $k$ and a message $m \in \{0,1\}^*$, and outputs a tag $t$. Since this algorithm may be randomized, we write this as $t \leftarrow \text{Mac}_k(m)$. While this may be randomized it is often deterministic.
3. **Vrfy**: The *verification algorithm* that takes as input a key $k$, a message $m$, and a tag $t$. It outputs a bit $b$, with $b = 1$ meaning *valid* and $b = 0$ meaning *invalid*. We assume without loss of generality that $\text{Vrfy}$ is deterministic, and so write this as $b := \text{Vrfy}_k(m, t)$.

### 5.2 Possibility of a replay attack

*Replay attack* has the possibility of being accomplished if $k$ is used multiple times. $(m_1, t_1) \rightarrow (m_1, t_1)$ We may assume for all $m$ and all $k$. Then $\text{TagVerify}(k, m, \text{TagGenerate}(k, m)) = \text{YES}$ $(m_1, t_1) \rightarrow (m_1, t_1)$ Bob: $m_1$ check $(m_2, t_2) \rightarrow (m_2, t_2)$ Bob: $m_2$ check $(m_3, t_3) \rightarrow (m_1, t_1)$ Bob: $m_1$ check this Additional $m_1$ can be used in a replay attack.

### 5.3 Existentially Unforgeability against Chosen-Message-Attack

**Definition 3.** A *message authentication code* $\Pi = (\text{Gen}, \text{Mac}, \text{Vrfy})$ is *existentially unforgeable under an adaptive chosen-message attack*, or just *secure*, if for all probabilistic polynomial-time adversaries $\mathcal{A}$, there exists a negligible function $\negl$ such that:

$$\Pr[\text{Mac - forge}_{\mathcal{A}, \Pi}(n) = 1] \leq \negl(n)$$

Is a PRF a good MAC? YES!

### 5.4 Connection between Hash Functions and MACs

*Diagram Depiction*

![Hash function and MAC connection diagram](image)

**Figure 4:** Hash function and MAC connection diagram

### 6 Keyed Hash Functions

With hash functions, it is difficult to find collisions (according to Pigeonhole principle, there will be collisions, no matter how rare, if the domain is larger than the range).
Definition 4. A Hash function Π = (Gen, H) is collision resistant if for all probabilistic polynomial-time adversaries A there exists a negligible function negl such that

Pr[Hash – coll_A,Π(n) = 1] ≤ negl(n).

6.1 HMAC

![HMAC Diagram](image)

Figure 5: HMAC diagram

Definition 5. Let (Gen, H) be a variable-length collision-resistant hash function. Let IV, opad, and ipad be fixed constants of length n. HMAC defines a MAC as follows:

1. Gen: on input 1^n, run Gen(1^n) to obtain a key s. Also choose k ← {0, 1}^n at random. Output the key (s, k).

2. Mac: on input a key (s, k) and a message m ∈ {0, 1}^* of length L, output the tag

   \[ t := H^s_IV((k \oplus \text{opad})||H^s_IV((k \oplus \text{ipad})||m)) \]

3. Vrfy: on input a key (s, k), a message m ∈ {0, 1}^*, and a tag t, output 1 if and only if \( t = \text{Mac}_{s,k}(m) \).

6.2 Types of Hash Functions

1. Universal: This way is the hardest for the attacker because \( k \in 0,1^\lambda \) is given after the Attacker has guessed \( x_0 \) or \( x_1 \).

2. One way Universal: This way is in between because \( k \in 0,1^\lambda \) is given after \( x_0 \) is guessed, but before \( x_1 \) is guessed.

3. Collision Resistant: This way is easiest for the attacker because \( k \in 0,1^\lambda \) is given before the Attacker guesses \( x_0 \) or \( x_1 \).
References