Attack #1

- Malicious player waits until all others have revealed their choice.
- Then, chooses the side that makes him a winner.

**Solution:**
- Keep all random choices secret (confidential) until all players have played.
- Specifically, two options exist:
  - Option 1: encryption based
  - Option 2: digital signature based
Thwarting Attack #1

Solution based on encryption:
- Players encrypt messages containing their choices using a randomly selected key.
- Keys are revealed after all encrypted messages have been exchanged so that choices may be decrypted.

Solution based on digital signatures:
- Players compute a digital signature of their choices and transmit the signature.
- After all digital signatures are submitted, choices are transmitted in clear text.
- Digital signatures are verified to ensure that choice has not been altered.
- Problem: message must be randomized otherwise choice can be deduced from signature! Include random nonce.

Attack #2

- Malicious player impersonates another player, i.e., sends messages on behalf of another player.
- Similarly, malicious player (server?) alters message from another player.
  - Not clear how this benefits attacker, but certainly destroy the game (denial of service)

Solution:
- Ensure all messages are available to all players -> Transparency.
- Players can verify that their own message has not been tampered with, or
- That no messages have been generated by others on their behalf.
- All players can verify computations and outcome of game.
Attack #3

- Malicious player pre-computes a number of random key/choice pairs so that all produce the same encrypted message.
- In the digital signature based approach, malicious player computes pairs of random nonces and choices yielding identical signatures.
- Then, reveals the key (or nonce) that decrypts his message so that he is a winner.
  - Very hard but not impossible.

Thwarting Attack #3

- Solution (for encryption based approach):
  - Digitally sign content of encrypted message
  - Include random encryption key in encrypted message
  - Include a random nonce to prevent pre-computing
- Specifically:
  - Server generates a one-time random nonce
    - Could be a very high resolution time-stamp
  - Players compute hash (digital signature) over:
    - Their choice
    - Their random key
    - Nonce
  - Encrypted message contains:
    - Their choice
    - Random key
    - Nonce
    - Hash
Thwarting Attack #3

- Solution (for digital signature based approach):
  - Include a random number selected by player
  - Include a random nonce chosen by server
- Specifically:
  - Server generates a one-time random nonce
    - Could be a very high resolution time-stamp
  - Players compute and send digital signature over:
    - Their choice
    - Their random number
    - Server-provided nonce
  - When revealing their choice, players send:
    - Their choice, and
    - Random number
  - This allows all players to verify that message has not been altered or pre-computed.

Crypto code

- The digital signature based approach appears quite a bit simpler.
- You will get from me:
  - C-code for:
    - MD5 hash function
    - With instructions for use.
  - Example provided in class:
    - CryptTest.c
    - md5.c, md5.h
Computing Digital Signatures

Define structure for the message to be signed and the signed message:

/* Two data structures that help us construct the message */
typedef struct {
    uint32 balance; /* 4 bytes, uint32 is defined above */
    char memo[84]; /* 84 bytes */
    uint8 nonce[8]; /* 8 bytes, uint8 is defined above */
} DataStruct;

typedef struct {
    DataStruct data; /* 96 bytes of data in format defined above */
    uint8 MD5Hash[16]; /* 16 bytes of hash */
} PlaintextStruct;

Computing Digital Signatures

The actual computation, including a random number:

/* Fill the structure with data */
plaintext.data.balance = 145847;
strcpy(plaintext.data.memo, "This is your balance");
/* Fill the nonce with random data, 4 bytes at a time*/
srandom(time(NULL)); /* Initialize the random number generator */
for (j = 0; j < 2; j++)
    RandQuad = random();
    memcpy(plaintext.data.nonce + 4*j, &RandQuad, 4);

/* Compute the hash over the data */
md5_starts( &ctx ); /* Initialize MD5 context, always done first */
/* Add the data to be hashed, may be called multiple times. Note, that we
only hash over the data - not the hash - by specifying the length as
sizeof(DataStruct) */
md5_update( &ctx, (uint8 *) &plaintext, sizeof(DataStruct));
/* Store the hash in MD5Hash element of plaintext */
md5_finish( &ctx, plaintext.MD5Hash);
Verifying Digital Signatures

- Compute the digital signature of the message and compare to supposed signature:

```c
/* Compute the hash over the received data - not including the hash. 
   The result will then be compared to the transmitted hash */
md5_starts( ctx );
md5_update( ctx, (uint8 *) &received, sizeof(DataStruct));
/* Store the hash in ReceivedHash */
md5_finish( ctx, ReceivedHash );

/* Formal comparison of hashes */
if(memcmp(ReceivedHash, plaintext.MD5Hash, 16))
    { printf("Failed: Hashes are different!\n"); }
else
    { printf("Passed: Hashes are equal!\n"); }
```