Project 1 :: Defeating SkyNet
(Security Essentials)

Dean Pisani 311210775
Kristy Hughes 310186293

1 Authentication

Our authentication system used Diffie-Hellman. We used a 3073-bit safe prime because it is a larger prime than the skeleton code, hence harder to factorise. We added a variable called prime generator so that the developer can change it from the default of 2. We set it at 5 so that the public key is larger and harder to crack, but it is generally set to 2, 3 or 5. The private key is a random number between 0 and the prime. It is better to have a large set of numbers that this could be so that it is difficult to guess, however it is pointless making it bigger than the prime because all arithmetic is done modular the prime. Our public key is the standard Diffie-Hellman public key based on the private key and the prime.

Parameters:

- Raw Prime: raw_prime is a 3073-bit safe prime from RFC 3526
- Prime Generator: $g = 3$
- Private Key: Random number between 0 and the prime
- Public Key: $g^{\text{private\_key}} \mod \text{raw\_prime}$

```python
# raw_prime is a 3073-bit safe prime from RFC 3526
prime = read_hex(raw_prime)
# Choose a prime generator, usually 2,3 or 5
g = 3

# Creates a Diffie-Hellman key
def create_dh_key():
    # Choose a random integer between
    # 0 and the prime number as the private key
    private = random.randint(0,prime)
    # The public key is is the
    # prime generator to the power of
    # the random private key mod a prime
    public  = pow(g, private, prime)
    return (public, private)
```

2 Confidentiality

To preserve confidentiality, we encrypted each message using AES. We chose AES as our cypher because although TDES is cryptographically secure, it is not as fast or as secure as AES. AES is now industry standard thus we thought it best to use it. Additionally AES supports key sizes that match the output of our Diffie-Hellman exchange. We used the CBC mode of operation, which is also a commonly used standard. The initialisation vector needs to be both random and shared, and so in order to do this, we used the first 16 characters of the shared secret.
This is the cipher was initialized:

```python
from Crypto.Cipher import AES
if self.server or self.client:
    # Initialize public and private keys using code above
    my_public_key, my_private_key = create_dh_key()
    # Send them our public key
    self.send(str(my_public_key))
    # Receive their public key
    their_public_key = int(self.recv())
    # Obtain our shared secret
    self.shared_hash = calculate_dh_secret(their_public_key, my_private_key)
    # Encode it properly
    self.shared_hash = bytes.fromhex(self.shared_hash)
# Initialisation vector is set to the first 16 bytes of our key
iv = self.shared_hash[:16]
# Cipher initialised with settings
self.cipher = AES.new(self.shared_hash, AES.MODE_CFB, iv)
```

And this is how data was encrypted using the cipher:

```python
encrypted_data = self.cipher.encrypt(data)
```

3 Integrity

To prevent tampering of the message we use data origin authentication. To do this, we implement HMAC-MD5 which hashes the message using the previously established shared secret as a key. Since this key is known only to the original sender and the recipient an attacker is unable to replicate this or tamper with the hashed message. Although MD5 has known issues, it is known that these problems do not carry on in their use in a HMAC. These authentication codes are attached to messages sent across the network, which can be used to verify the authenticity of messages received by re-computing the HMAC and comparing it to the one sent across.

When a message is sent:

```python
# Create a HMAC and prepend it to the message
h = HMAC.new(self.shared_hash)
h.update(data)
# Prepend it to the data
mac_data = bytes(h.hexdigest() + data.decode("ascii"),"ascii")
```

When a message is received:

```python
# createe the HMAC
h = HMAC.new(self.shared_hash)
# split the hmac from the front of the data
hmac = data[:h.digest_size*2]
data = data[h.digest_size*2:]
h.update(data)
# check that the hmac matches
if h.hexdigest() != str(hmac, 'ascii'):
    raise RuntimeError("Bad message: HMAC does not match")
```

4 Preventing Replay

Replay attacks occur when an attacker takes a previously sent encrypted message and sends it to the server again. To prevent this, the user appends a timestamp at the beginning of the message before it is encrypted and when the server decrypts it, it checks that the timestamp of the current message is later than the timestamp of the previous message received. When a message is sent:

```python
# Add a timestamp to the message
current_time = datetime.datetime.now()
# Use the following code to test if it works: subtract some time from now
#current_time = self.last_message_time - datetime.timedelta(1,0)
timestr = datetime.datetime.strftime(current_time, timestamp_format)
mac_data = bytes(timestr, 'ascii') + mac_data
```

When a message is received:

```python
# Get the timestamp from the message
msg_time = datetime.datetime.strptime(tstamp, timestamp_format);
# Check that it is newer than the last message
if msg_time <= self.last_message_time:
    raise RuntimeError("Bad timestamp: message not newer than last recieved one")
# Update the time stamp
self.last_message_time = msg_time
```

5 Peer to Peer file transfers

Peer-to-peer file transfers allow botnets to function without relying on a central server to be always operational. Such a transfer mechanism has the advantage that there is no single point of failure, and similarly no single point which would present itself as a target for attackers. Additionally it would also allow a bot net to function even when the central server is inaccessible or blocked. However, a central web server provides a source of authority for all bots in our network, and there is no possibility of outdated files or instructions being distributed across the network like there can be for a peer-to-peer system.

6 In the Real World

In the real world, bot nets could be used to:

- Distributed denial of service attacks (dDoS) by asking all computers connected to the botnet to ping a server.
- Sending spam emails
- Distributing malware through p2p transfers
- Brute-forcing by asking all computers to conduct a brute force attack on with different starting points

There are two methods for detecting a bot net: static analysis and behavioral analysis. Static analysis involves checking items against a known list of malicious items. It is difficult to keep this list up to date, making this method not very effective. Behavioural analysis involves monitoring communications in a network for behaviour similar to a bot net. This includes monitoring network for unusual traffic (such as at unusual times, of unusual volume, protocols such as IRC, SMTP, UDP) and monitoring computers to see if processes are doing more work than expected. However the problem with this method is that it may be legitimate for a computer to be exhibiting these behaviors.