DATA PROTECTION

Lab 2 / Merkle Trees

Team: Oriol Lalaguna Royo, Sofia Tsagiopoulou

The python version used to develop the following scripts is >3.0.0.

1. Build the tree

   We developed a script to build a Merkle tree with n docs. These documents are the ones that will be hashed. The user enters the number of the docs and the names of the documents.

   A prefix is added for each document (docs and nodes) to prevent attacks. The hashes of the docs are nodes in layer 0 and they are as many as the docs. For example, if the input is for 4 docs the nodes (hash(docs)) will be node0.0, node0.1, node0.2, node0.3.

```
#prefixes
os.system("echo -n ''\x35\x35\x35\x35\x35\x35'' > doc.pre")
os.system("echo -n '\xe8\xe8\xe8\xe8\xe8\xe8' > node.pre")
```

```
for i in range(doc_num): #hash the documents
    file.append(input("Enter the document" + str(i+1) + ":"))
    os.system("cat doc.pre " + file[i] + " | openssl dgst -sha1 -binary | xxd -p > node0." + str(i))
    os.system("echo -n '" + str(0) + ":" + str(i) + ":' >> merkletree.txt")
    os.system("cat node" + str(0) + "." + str(i) + " >> merkletree.txt")
```

   As the first layer (0) is created, the program checks if it is necessary to build another layer to reach the root. To build another layer we have to hash the nodes nodei.j and nodei.j+1. In this case it is possible to face the problem of no existing node. For example, if the docs are 3, we will have 3 nodes (node0.0, node0.1, node0.2) but it is not possible with this structure to build the next layer as we need to hash two nodes to create the node that is in the next layer. For this reason, this node is replaced by the empty file in the computation of the next node. Or simply, the missing file is removed from the hash computation.

```
if i == hashes_num-1 and hashes_num%2 == 1:
    os.system("cat node.pre node" + str(layer) + "." + str(i) + " | openssl dgst -sha1 -binary | xxd -p > node" + str(layer+1) + "."
else:
    os.system("cat node.pre node" + str(layer) + "." + str(i) + " node" + str(layer) + "." + str(i+1) + " | openssl dgst -sha1 -binar
```

   Then, we create the root hash which is the last node that is created. After this step, we build the public information including the hash algorithm used (e.g., sha1), the two selected prefixes (in hexadecimal), the number of documents (decimal number), n, the depth of the tree (the number of layers including the leaves and the root, as a decimal number) and the root hash (the contents of the tree root in hexadecimal).

   The description of the tree is stored in a file and below we can see an example of our script and its output.

```
┌──(kali㉿kali)-[~]
└─$ sudo python buildtree2.py
Enter the number of the doc files:3
Enter the document1:doc0.dat
Enter the document2:doc1.dat
Enter the document3:doc2.dat

┌──(kali㉿kali)-[~]
└─$ cat finalmerkletree.txt
MerkleTree:sha1:353535353535:e8e8e8e8e8e8:3:3:ef97fb3807576940bd248cabad054c186016cc70
0:0:7ca4b5ba02dfce15eaf08d4ec099a1a94ca9a08c
0:1:4284521787346c2e6592227e7552eacf1064b5ed
0:2:0dc8a1be1b78b82440159ed18ab133d894182b43
1:0:a5af2ad002141f14a0895e44eeabab0df6167f33
1:1:16676ecaf9daa464badeeea81c0e3452ffd5c499
2:0:ef97fb3807576940bd248cabad054c186016cc70
```

2. Insert a new document

We developed a script for inserting a doc in the existing tree, adding, and modifying the necessary nodes.

The user enters the new file and then the script looks into the text file and gets the public information of the tree to modify it and the number of nodes. Then, we write the information about the old nodes in another txt file, we create the hash of the new doc as we did it in the first script, and we enter it in the text file. After that, we follow the same logic with the first script but based on the number of nodes not the docs.

Below we can see an example with the previous tree being modified. It is obvious that the nodes that are affected are the node1.1 and the root node. Also, the node0.3 is the new one.

```
┌──(kali㉿kali)-[~]
└─$ sudo python add2.py
[sudo] password for kali:
Enter the new file:doc3.dat

┌──(kali㉿kali)-[~]
└─$ cat finalmerkletree.txt
MerkleTree:sha1:353535353535:e8e8e8e8e8e8:4:3:203eb7d477371bb3c1cd1883e39d7f3fa43975ee
0:0:7ca4b5ba02dfce15eaf08d4ec099a1a94ca9a08c
0:1:4284521787346c2e6592227e7552eacf1064b5ed
0:2:0dc8a1be1b78b82440159ed18ab133d894182b43
0:3:0ed921bcb8b21086ad813d3f3dba63099570c7d2
1:0:a5af2ad002141f14a0895e44eeabab0df6167f33
1:1:1b55819c71c5756848f0400294e40210447c8913
2:0:203eb7d477371bb3c1cd1883e39d7f3fa43975ee
```

3. Proof of membership

In this script the user enters the document at position k to produce a proof of membership verifying in the hash tree.

First, we read the public info of the tree given in advance. This document must be located inside the directory of the running script. With that, we can check if the document to verify is in the same position inside the tree.

After this, we compute the hash of the document to verify it.

```
for i in f: #Checks if the Document and position is found in the tree
    if os.popen("cat doc.pre " + nameDoc + " | openssl dgst -sha1 -binary | xxd -p").read() in i and layer == int(i[0]) and p == int(i[2]):
        found = True
        break
f.close()
```

If the verification of the document and position are correct, it prints the information to verify the proof of membership. If the document and position are incorrect, an error is shown.

```
┌──(kali㉿alice)-[/media/sf_shared/DProtection/LAB2/5]
└─$ sudo python proofmember.py
Document to generate the proof of membership:doc0.dat
Position of the Document to generate the proof of membership:0
0:1:4284521787346c2e6592227e7552eacf1064b5ed

1:1:16676ecaf9daa464badeeea81c0e3452ffd5c499

2:0:ef97fb3807576940bd248cabad054c186016cc70
```

```
┌──(kali㉿alice)-[/media/sf_shared/DProtection/LAB2/5]
└─$ sudo python proofmember.py
Document to generate the proof of membership:doc1.dat
Position of the Document to generate the proof of membership:1
0:0:7ca4b5ba02dfce15eaf08d4ec099a1a94ca9a08c

1:1:16676ecaf9daa464badeeea81c0e3452ffd5c499

2:0:ef97fb3807576940bd248cabad054c186016cc70
```

4. Verify the proof of membership

The function of this script is to check and show in screen if a node belongs to a hash tree or not. To do it, it requests the public info of the tree given in advance, the document to verify and the filename where the necessary nodes are located to verify it. The nodes are stored in the following format example: [*node0:1*:*node2.0*:*node2.1*].

We must verify if the root hashes are the same or if the hash of a node needs another node to get the upper node and then print the result. This for all the layers.

```
┌──(kali㉿alice)-[/media/sf_shared/DProtection/LAB2/5]
└─$ sudo python verifyproof.py
Public info of the Merkle tree:MerkleTree:sha1:353535353535:e8e8e8e8e8e8:3:3:
ef97fb3807576940bd248cabad054c186016cc70
Document to verify the proof of membership:doc0.dat
Filename with the necessary nodes to verify:nodeinfo.txt
Proof of membership verified
```