SECURITY

Assignment 7, Saturday, October 27, 2017

S1013793 Carlo Jessurun
S1013792 Tony Lopar
Radboud University

Teaching assistant: Joost Rijneveld

Assignment 1

(35 points) Hash functions play an important role in the BitTorrent protocol. When using BitTorrent, users receive pieces of a large file from different providers (typically referred to as 'seeders'). When a user wants to obtain a certain file, he needs a '.torrent' file. Amongst other things, this file can contain a list of hashes. These hashes are used to guarantee the integrity of the individual pieces, allowing a user to check each piece when he receives it.

(a) An attacker tries to replace one piece of the original file with a new one. What property of the hash function is important here so that the attack is detectable? Note that the attacker also has access to the original file.

(b) Say Alice wants to download a file of 2 GB (e.g. an authentic Linux distribution), and it is split in pieces of 16 KB each. Suppose that the .torrent file she uses contains a list of SHA-1 hashes. What is the minimum size of that .torrent file? (Hint: A SHA-1 hash is 160 bits = 20 bytes)

As (b) shows, that can be quite a large file to deal with. We could resolve this by increasing the size of each piece so that we do not need as many hashes. That would however make it more difficult for peers to quickly share their pieces. An alternative approach is to use *binary hash trees*.

Consider Figure 1. On the circles along the bottom (the leaf nodes), we place the hash values of the different pieces: $N^0_0 = h(piece_0)$, $N^0_1 = h(piece_1)$, $N^0_2 = h(piece_2)$, …. Each of the higher nodes in the tree contains the combined hash of its children. For example, the value in the first node of the next layer would be $N^1_0 = h(N^0_0 || N^0_1) = h(h(piece_0)||h(piece_1))$. The node above that would be $N^2_0 = h(N^1_0 || N^1_1)$, etc. We continue doing this all the way to the top of the tree. The top of the tree is called the root R, which is always contained in the .torrent file.

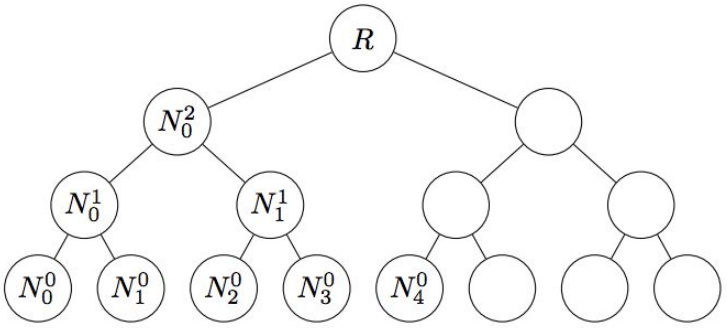

Figure 1: binary hash tree

(c) Suppose the .torrent file does not contain a list of all hashes, but only the root hash. At which stage would you be able to check the integrity of the pieces you receive from others? Why?

Instead of just sending you the piece, a seeder should now also send you a few of the nodes in the tree. This guarantees that you have enough information to calculate the root node, using his piece and the hashes he sends along. You can then compare it to the root node in the '.torrent' file. If it matches, the piece was authentic!

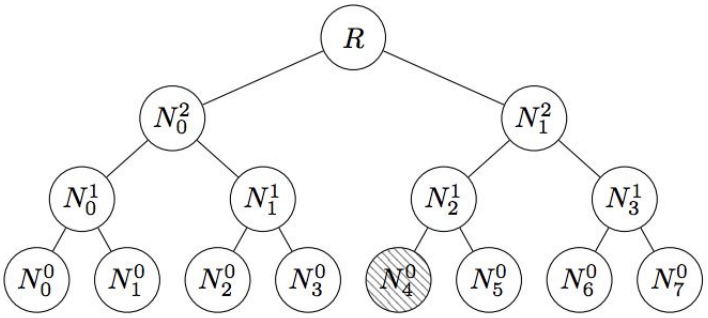

Figure 2: binary hash tree with a marked leaf node

(d) Suppose someone sends you piece4 . Now you can compute $N^0_4 = h(piece_4)$, marked gray in Figure 2. Looking at the figure, what other N-values (nodes) would he need to send you before you can compute R, to check if the piece was correct?

(e) Suppose we have a file that consists of 1024 pieces. How many hashes would need to be sent along with each of the pieces? This exercise was only a very slight abstraction from reality. Have a look at http://bittorrent.org/beps/bep_0030.html if you are interested in what this looks like in practice.

Solutions 1

A. It is important that the hash function has a good second preimage resistance. In this case the attacker will not be able to construct a file with the same hash. In the case an attacker changes a file, we should be able to detect it when checking it with the .torrent-file, since the hashes should be unequal.

B. 2GB = 125 000 x 16kb = 2.000.000 bytes. 125.000 pieces with SHA-1 hash of 20 bytes. The torrent file will contain 125.000 hashes, so the filesize will be 125.000 x 20 = 2.500.000 bytes = 2,5 MB

C. You can check the integrity of the pieces when you know the hash of both children of the root. This is because you need the hash of both children to compute the hash of the root. This computed value can be compared to the root in the .torrent-file with which we can check the integrity.

D. He should also send the N^0_5 in order to find out what N^1_2 is. Further, he needs to send hash N^1_3 so we may calculate N^2_1 which is the right child of the root. Now we know the hash of the right child of the root we should also have the left child N^2_0 to compute the root. This means we also need to receive N^0_5, N^1_3 and N^2_0 from this person.

E. Since there are 1024 pieces, the tree would have 1024 leaves. We know that $2^{10} = 1024$, so the root will be the 10^{th} node from the leaves. At the leaves we should know two values of N to compute the parent. At each other intermediate height, we should have one other value of N to compute the parent of the current node. This means there should be 9 values of N more known. So every piece should include 11 hashes in order to calculate the root.

Assignment 2

(35 points) Lately there have been a number of high-profile database leaks, with millions of accounts hitting the internet. For example, last year saw leaks resulting from attacks on Last.fm and Dropbox[1] that took place back in 2012.

(a) Last.fm simply stored MD5 hashes of passwords. Suppose you can compute 9 billion hashes per second, and you know that someone's password only contains digits and letters (both lowercase and uppercase). How long would it take to break a fully random 11-character password?

(b) Suppose you're not interested in one specific account, but you just want access to any account. Knowing that the leak included roughly 43 million entries, how long would it take on average before you find a match? Assume that comparing two hashes is free.

(c) Luckily, Dropbox had 'salted' their passwords before hashing them, preventing such a multi-target attack. On top of that, they used the bcrypt hash function, which is designed to be slow. Assume this allows you to do only 10 hashes per second. How long would it take to break a fully random 7-character alphanumeric[2] password?

For simplicity, we will briefly ignore salts and multi-target attacks for the remainder of this exercise.

In a typical (web) application, users are asked to submit their password when logging in. This is then hashed and compared to a stored value H(password). The plaintext password itself is never stored.

(d) Dropbox was transitioning from SHA1 to bcrypt hashes: roughly half the passwords were still stored as SHA1 hashes. In order to upgrade from a SHA1 hash to a bcrypt hash, the user had to log in at least once. Explain why this is necessary: which property of SHA1 (i.e. P, P2 or C) prevents Dropbox from simply updating the hashes themselves, without the user logging in?

(e) The reason why Dropbox wanted to upgrade to bcrypt is because it is much more timeconsuming to brute force bcrypt hashes, making it less of a problem if they ever leak. Assume that some users never log in, but Dropbox does want to protect all passwords using bcrypt. How could they have done this, assuming they only have the SHA1 hash of the passwords? Explain what they would need to compute, and mention what they need to check when a user logs in.

Solutions 2

A. First we have to calculate how many different characters may be possible. These are 10 digits, 26 lowercase letters and 26 uppercase letters. The sum of this is 62 characters. So, for all characters in the password we have 62 possibilities. This means that to break a fully random password would take at most 62^{11} hashes. Dividing this value by 9 billion results in $5{,}8 \times 10^9$ seconds. This is approximately 183.3 years
B. You could enter a random password and compare it's hash to the hashes in the entries

C. Like in A, there are 62 possible characters for each position. So this gives us 62^7 different possible passwords which are $3.5 * 10^{12}$. This means we would need $3.5 * 10^{11}$ seconds for this. Recalculating this to years results in 11166.96 years.
D. The property of preimage resistance of the hash, since this property prevents to find out the plaintext password only knowing the hash. This means dropbox should not be able to figure out the plaintext passwords which have to be hashed with bcrypt. Therefore all users have to login at least once to rehash the password with bcrypt.
E. They could use the known SHA-1 hash as input of the bcrypt hashing. To save computing time, this could be done with users that did not log in for an extended period of time. With this solution, all passwords will be stored as bcrypt hashes. We then have the following two scenarios:
   When a user changes his password, then this new password should be stored only in bcrypt.
   When a user that hasn't changed it's password changed logs in, then we should first compute the SHA-1 hash of the entered password and hash this SHA-1 hash with bcrypt to compare it with the stored hash.

Assignment 3

(30 points) On 23 February 2017, researchers from CWI Amsterdam and Google Researchers broke SHA-1 in practice. In particular, they derived two different messages M_1 and M_2 that, if preceded by a common prefix S, resulted in SHA-1(S||M_1) = SHA-1(S||M_2). The messages *S, M_1, M_2* are given at http://www.sos.cs.ru.nl/applications/courses/ security2017/sha1, alongside a SHA-1 digest computation tool.

We will not dive into the technical details of the SHA-1 hash function, but will only recall that SHA-1 suffers from the *length extension weakness*: Given SHA-1(M) and a message string M' , it is possible[3] to compute SHA-1(M||M') without knowing M.

(a) Compute SHA-1(S||M_1) and SHA-1(S||M_2).

(b) Consider M_a := C0 FF EE. Find a prefix P for M_a and a second message M_b such that SHA-1(P||M_a) = SHA-1(M_b), where P||M_a and M_b are distinct.

(c) Compute the corresponding SHA-1 hash digest.

(d) The prefix S is not a random bit string, but rather a carefully chosen string. Discover the main motivation behind S.

Solutions 3

A. Using the SHA-1 computation tool and the M_1, M_2 and S in the description of the tool we computed the following hashes:

S	$(S\|\|M_1)$	$(S\|\|M_2)$
25 50 44 46 2D 31 2E 33 0A 25 E2 E3 CF D3 0A 0A 0A 31 20 30 20 6F 62 6A 0A 3C 3C 2F 57 69 64 74 68 20 32 20 30 20 52 2F 48 65 69 67 68 74 20 33 20 30 20 52 2F 54 79 70 65 20 34 20 30 20 52 2F 53 75 62 74 79 70 65 20 35 20 30 20 52 2F 46 69 6C 74 65 72 20 36 20 30 20 52 2F 43 6F 6C 6F 72 53 70 61 63 65 20 37 20 30 20 52 2F 4C 65 6E 67 74 68 20 38 20 30 20 52 2F 42 69 74 73 50 65 72 43 6F 6D 70 6F 6E 65 6E 74 20 38 3E 3E 0A 73 74 72 65 61 6D 0A FF D8 FF FE 00 24 53 48 41 2D 31 20 69 73 20 64 65 61 64 21 21 21 21 21 85 2F EC 09 23 39 75 9C 39 B1 A1 C6 3C 4C 97 E1 FF FE 01	25 50 44 46 2D 31 2E 33 0A 25 E2 E3 CF D3 0A 0A 0A 31 20 30 20 6F 62 6A 0A 3C 3C 2F 57 69 64 74 68 20 32 20 30 20 52 2F 48 65 69 67 68 74 20 33 20 30 20 52 2F 54 79 70 65 20 34 20 30 20 52 2F 53 75 62 74 79 70 65 20 35 20 30 20 52 2F 46 69 6C 74 65 72 20 36 20 30 20 52 2F 43 6F 6C 6F 72 53 70 61 63 65 20 37 20 30 20 52 2F 4C 65 6E 67 74 68 20 38 20 30 20 52 2F 42 69 74 73 50 65 72 43 6F 6D 70 6F 6E 65 6E 74 20 38 3E 3E 0A 73 74 72 65 61 6D 0A FF D8 FF FE 00 24 53 48 41 2D 31 20 69 73 20 64 65 61 64 21 21 21 21 21 85 2F EC 09 23 39 75 9C 39 B1 A1 C6 3C 4C 97 E1 FF FE 01 7F 46 DC 93 A6 B6 7E 01 3B 02 9A AA 1D B2 56 0B 45 CA 67 D6 88 C7 F8 4B 8C 4C 79 1F E0 2B 3D F6 14 F8 6D B1 69 09 01 C5 6B 45 C1 53 0A FE DF B7 60 38 E9 72 72 2F E7 AD 72 8F 0E 49 04 E0 46 C2 30 57 0F E9 D4 13 98 AB E1 2E F5 BC 94 2B E3 35 42 A4 80 2D 98 B5 D7 0F 2A 33 2E C3 7F AC 35 14 E7 4D DC 0F 2C C1 A8 74 CD 0C 78 30 5A 21 56 64 61 30 97 89 60 6B D0 BF 3F 98 CD A8 04 46 29 A1	25 50 44 46 2D 31 2E 33 0A 25 E2 E3 CF D3 0A 0A 0A 31 20 30 20 6F 62 6A 0A 3C 3C 2F 57 69 64 74 68 20 32 20 30 20 52 2F 48 65 69 67 68 74 20 33 20 30 20 52 2F 54 79 70 65 20 34 20 30 20 52 2F 53 75 62 74 79 70 65 20 35 20 30 20 52 2F 46 69 6C 74 65 72 20 36 20 30 20 52 2F 43 6F 6C 6F 72 53 70 61 63 65 20 37 20 30 20 52 2F 4C 65 6E 67 74 68 20 38 20 30 20 52 2F 42 69 74 73 50 65 72 43 6F 6D 70 6F 6E 65 6E 74 20 38 3E 3E 0A 73 74 72 65 61 6D 0A FF D8 FF FE 00 24 53 48 41 2D 31 20 69 73 20 64 65 61 64 21 21 21 21 21 85 2F EC 09 23 39 75 9C 39 B1 A1 C6 3C 4C 97 E1 FF FE 01 73 46 DC 91 66 B6 7E 11 8F 02 9A B6 21 B2 56 0F F9 CA 67 CC A8 C7 F8 5B A8 4C 79 03 0C 2B 3D E2 18 F8 6D B3 A9 09 01 D5 DF 45 C1 4F 26 FE DF B3 DC 38 E9 6A C2 2F E7 BD 72 8F 0E 45 BC E0 46 D2 3C 57 0F EB 14 13 98 BB 55 2E F5 A0 A8 2B E3 31 FE A4 80 37 B8 B5 D7 1F 0E 33 2E DF 93 AC 35 00 EB 4D DC 0D EC C1 A8 64 79 0C 78 2C 76 21 56 60 DD 30 97 91 D0 6B D0 AF 3F 98 CD A4 BC 46 29 B1

a. SHA-1($S||M_1$) f92d74e3874587aaf443d1db961d4e26dde13e9c
b. SHA-1($S||M_2$) f92d74e3874587aaf443d1db961d4e26dde13e9c

We can conclude that both resulting hashes are equal.

B. So in this case we added the prefix + m1 to the block “C0 FF EE”. In the second one, assuming that we can use any message possible we took the prefix, added m2 and the block “C0 FF EE” at the end.

$P\|\|M_a$	M_b
25 50 44 46 2D 31 2E 33 0A 25 E2 E3 CF D3 0A 0A 0A 31 20 30 20 6F 62 6A 0A 3C 3C 2F 57 69 64 74 68 20 32 20 30 20 52 2F 48 65 69 67 68 74 20 33 20 30 20 52 2F 54 79 70 65 20 34 20 30 20 52 2F 53 75 62 74 79 70 65 20 35 20 30 20 52 2F 46 69 6C 74 65 72 20 36 20 30 20 52 2F 43 6F 6C 6F 72 53 70 61 63 65 20 37 20 30 20 52 2F 4C 65 6E 67 74 68 20 38 20 30 20 52 2F 42 69 74 73 50 65 72 43 6F 6D 70 6F 6E 65 6E 74 20 38 3E 3E 0A 73 74 72 65 61 6D 0A FF D8 FF FE 00 24 53 48 41 2D 31 20 69 73 20 64 65 61 64 21 21 21 21 21 85 2F EC 09 23 39 75 9C 39 B1 A1 C6 3C 4C 97 E1 FF FE 01 7F 46 DC 93 A6 B6 7E 01 3B 02 9A AA 1D B2 56 0B 45 CA 67 D6 88 C7 F8 4B 8C 4C 79 1F E0 2B 3D F6 14 F8 6D B1 69 09 01 C5 6B 45 C1 53 0A FE DF B7 60 38 E9 72 72 2F E7 AD 72 8F 0E 49 04 E0 46 C2 30 57 0F E9 D4 13 98 AB E1 2E F5 BC 94 2B E3 35 42 A4 80 2D 98 B5 D7 0F 2A 33 2E C3 7F AC 35 14 E7 4D DC 0F 2C C1 A8 74 CD 0C 78 30 5A 21 56 64 61 30 97 89 60 6B D0 BF 3F 98 CD A8 04 46 29 A1 C0 FF EE	25 50 44 46 2D 31 2E 33 0A 25 E2 E3 CF D3 0A 0A 0A 31 20 30 20 6F 62 6A 0A 3C 3C 2F 57 69 64 74 68 20 32 20 30 20 52 2F 48 65 69 67 68 74 20 33 20 30 20 52 2F 54 79 70 65 20 34 20 30 20 52 2F 53 75 62 74 79 70 65 20 35 20 30 20 52 2F 46 69 6C 74 65 72 20 36 20 30 20 52 2F 43 6F 6C 6F 72 53 70 61 63 65 20 37 20 30 20 52 2F 4C 65 6E 67 74 68 20 38 20 30 20 52 2F 42 69 74 73 50 65 72 43 6F 6D 70 6F 6E 65 6E 74 20 38 3E 3E 0A 73 74 72 65 61 6D 0A FF D8 FF FE 00 24 53 48 41 2D 31 20 69 73 20 64 65 61 64 21 21 21 21 21 85 2F EC 09 23 39 75 9C 39 B1 A1 C6 3C 4C 97 E1 FF FE 01 7F 46 DC 93 A6 B6 7E 01 3B 02 9A AA 1D B2 56 0B 45 CA 67 D6 88 C7 F8 4B 8C 4C 79 1F E0 2B 3D F6 14 F8 6D B1 69 09 01 C5 6B 45 C1 53 0A FE DF B7 60 38 E9 72 72 2F E7 AD 72 8F 0E 49 04 E0 46 C2 30 57 0F E9 D4 13 98 AB E1 2E F5 BC 94 2B E3 35 42 A4 80 2D 98 B5 D7 0F 2A 33 2E C3 7F AC 35 14 E7 4D DC 0F 2C C1 A8 74 CD 0C 78 30 5A 21 56 64 61 30 97 89 60 6B D0 BF 3F 98 CD A8 04 46 29 A1 C0 FF EE

C. SHA-1: “667e54f8636176e996cbc88f6a345fda4e258648”

D. On the paper of the team behind Shattered: https://shattered.io/static/shattered.pdf. We found that: "*The prefix of the colliding messages was carefully chosen so that they allow an attacker to forge two PDF documents with the same SHA-1 hash yet that display arbitrarily-chosen distinct visual contents. We were able to find this collision by combining many special cryptanalytic techniques in complex ways and improving upon previous work. In total the computational effort spent is equivalent to 263.1 SHA-1 compressions and took approximately 6 500 CPU years and 100 GPU years. As a result while the computational power spent on this collision is larger than other public cryptanalytic computations, it is still more than 100 000 times faster than a brute force search.*"