Review of: Plaintext Recovery Attacks against SSH

Raoul Estourgie
Radboud University Nijmegen
s3022420

Ben Brücker
Radboud University Nijmegen
s0413291

Abstract

This is a review/summary of the article "Plaintext Recovery Attacks against SSH" by Martin R. Albrecht, Kenneth G. Paterson and Gaven J. Watson [1]. We discuss their findings concerning a possible attack on the OpenSSH implementation of the SSH BPP protocol. This attack enables an attacker to recover 14 bits of plaintext with probability 2^{-14} and 32 bits with probability 2^{-18}. We also cover their suggestions for preventing thse attacks.

Secure Shell (SSH) connects computers securely over insecure network connections [2]. This protocol was released in 1995 and was designed to replace rlogin, rsh, Telnet and similar insecure protocols. The SSH protocol covers authentication, confidentiality and integrity [3]. Our review article "Plaintext Recovery Attacks against SSH" paper [1] focuses their attack on the OpenSSH implementation of the SSH Binary Packet Protocol (BPP).

I. The SSH-BPP protocol

The Binary Packet Protocol (BPP) of SSH encrypts a plaintext and then protects it's integrity by appending a MAC value [2].

Before encryption, the message is encoded by prefixing a 4 byte packet-length field and a 1 byte padding-length field. At least 4 bytes of randomized padding must be added as a suffix with a maximum of 255 bytes [1]. The message is then encrypted with a cypher of choice, for example aes128-cbc. After that, a MAC value is added. This MAC is computed from the cyphertext and a 32-bit packet sequence number. (See figure 1)

The packets then form a data stream since the encryption is in CBC mode. Every packet $i-1$ on a connection will be the initialization vector (IV) for packet i on the same connection [1].

For decryption it is essential that the receiver decrypts the first ciphertext block to be able to read the length field. Without this he does not know when a complete packet has arrived and when to perform the MAC check. Most SSH implementations wait for a time until enough data has arrived to complete the packet [1].

The SSH protocol [2] also specifies error handling for the BPP protocol. The connection should terminate whenever a transmission error occurs or MAC verification fails. When such a termination happens,the connection should be re-established. Implementations are free to send error messages to their peer when an error occurs.

II. Open SSH implementation of the BPP protocol

Open SSH follows the guidelines for BPP fairly close [1]. But the details about the specific implementation are important, since these details enable the attack by Albrecht et al. [1].

After receiving the first ciphertext block, Open SSH first performs a length check. If the length given in the length field is not between 5 and 2^{18} it sends a *SSH2 MSG DISCONNECTED* error back to the sender.

Next OpenSSH checks that the total number of bytes expected is indeed a multiple of the

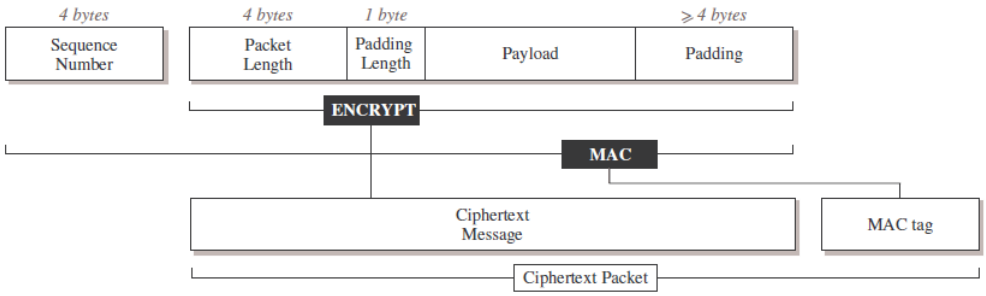

Figure 1: SSH BPP packet format and cryptographic processing [1]

block size. When this check fails, the TCP connection will terminate without an error message [1].

When all data for the package has arrived, OpenSSH performs a MAC check. If this check fails, a *Corrupted MAC on input.* error message is returned to the sender.

Albrecht et al. [1] note that further checks performed by OpenSSH are not of interest for their attack. Also it is important to note that each of the performed checks has a distinct type of error behavior, that can be used to facilitate the plaintext recovery attack.

III. The Attack

General overview

The fact that the first block of ciphertext in a SSH packet includes the 4 byte packet-length field can be exploited to recover bits of plaintext from an encrypted message. The attacker simply has to inject a chosen ciphertext block as the first block of a new SSH packet to induce the SSH server to treat the resulting plaintext length-field as the length of this SSH packet.

The OpenSSH implementation of BPP only supports lengths up to 2^{18} bits. But since the length field is 4 bytes long, a packet will only pass the length check if the first $32 - 18 = 14$ bits are all 0. Because only then can the length-field have a value between 5 and 2^{18}. So if the attacker does not receive a *SSH2 MSG DISCONNECTED* error, he knows the first 14 bits of plaintext. Because the ciphertext block also passed the block-length check, the attacker knows that in case of $L = 16$ (as with AES) the last 4 bits encode value 12, increasing the known bits of plaintext to 18 (So $L = 8$ reveals 3 bits for a total of 17).

When the attacker succeeds in the first part of the attack and the SSH connection enters a wait state. He can iterate the attack by feeding random cyphertext blocks into this connection and waiting after each block. When the target returns a *Corrupted MAC on input.* error we know that it received enough blocks to trigger a MAC check. At this point we know exactly how long the packet length is, and therefore all 32 bits of the length-field. Because of the chaining property of the CBC mode these 32-bits leak information about the rest of the ciphertext [1].

Formal Notation

Let us first define some notations. K is the key of our block cipher, which is fixed for the duration of a connection. F_k and F_k^{-1} are the encryption and decryption operations of the block cipher. L is the block size of the block cipher in bytes. The CBC mode in SSH BPP then operates as follows: given a sequence $p_1, p_2, ..., p_n$ of plaintext blocks making up a packet, we have: $c_i = F_k(c_{i-1} \oplus p_i), i = 1, 2, ..., n$ where c_0, the Initializing Vector (IV), is the last block of the previous ciphertext. Decryption works as follows: $p_i = c_{i-1} \oplus F_k^{-1}(c_i), i = 1, 2, ..., n$

The attack

The attacker collects a target ciphertext block c_i^* from an established SSH connection. Let c_{i-1}^* denote the preceding target block and p_i^* denote the target plaintext of c_i^*. $p_i^* = c_{i-1}^* \oplus F_k^{-1}(c_i^*)$ The attacker now injects c_i^* as the first block of a new packet in the SSH connection. Let c_n denote the last ciphertext block of the preceding packet on the connection. This block is used as the IV for the new packet, so it will receive $p_1^{'}$ after decryption because: $p_1^{'} = c_n \oplus F_k^{-1}(c_i^*)$

By combining these equations, we get: $p_i^* = c_{i-1}^* \oplus p_1^{'} \oplus c_n$ (1) After receiving c_i^*, there are two options. Either a termination of the TCP connection over which the connection is running and sending a *SSH2 MSG DISCONNECTED* error, or the SSH connection enters a state in which it is waiting for more data. When it enters this waiting state, we know that $p_1^{'}$ has passed the length check. This only occurs if the packet length field in $p_1^{'}$ lies between 5 and 2^{18}, which only occurs if the first 14 bits of $p_1^{'}$ are zero, we can use this and the equation in (1) to calculate the first 14 bits of p_i^*. $p_1^{'}$ is a random 32-bit value, therefore the length check will pass with probability $2^{-14} - 5/2^{18} \approx 2^{-14}$. Which is a low chance but will give us the first 14 bits of p_i^*, and still is a way better chance then guessing.

Complications of the attack

OpenSSH checks if the packet length field is at most 2^{18} and then checks if it has a certain divisibility property. The connection is torn down if either check fails. The effect of the length checking will reduce the success of the attack to about 2^{-18}. However the effect is that it reduces the maximum number the attacker has to inject to only 2^{14}.

Recovering 32 plaintext bits

We now continue where we left at the second attack, the attacker retrieved 14 plaintext bits and now knows that the server is in a wait state. When L = 16 (e.g. with aes), this implies that the first 14 bits of the length field in $p_1^{'}$ will all be zero, and that the last 4 bits of this field encode the value 12. So we now have 18 bits we can use to calculate parts of p_i^* using equation (1)(with 3des L = 8 and the last 3 bits of the length field should encode the value 4 which gives us 17 bits in total). When both test pass we no know that the server is waiting for more data until the following condition is no longer satisfied.

```
if (buffer_len(&input) < need + maclen)
return SSH_MSG_NONE;
```

Once the test fails, the MAC check will be triggered. The attacker continues his attack by injecting packets of size *maclen*. The attacker will keep feeding the server packets until it receives a MAC error which is after at most $2^{18}/L$ packets. When this occurs we know the value of *need*, the exact value of $p_1^{'}$.

```
need = 4 + packet_length - block_size
```

Knowing this 32-bit value, the first 32 bits of p_i^* can be recovered using equation (1). The overall success probability is 2^{-18} with the recommended AES-CBC and 2^{-17} with the required 3des-CBC.

IV. Alternatives

Bellare et al. [4] propose a few alternatives to the BPP protocol. SSH-NPC, SSH-$NPC, SSH-CTR, SSH-CTRIVCBC and SSH-EIV-CBC.

SSH-NPC and SSH-$NPC (No Packet Chaining) use CBC mode with random per packet IV's [1]. Albrecht et al show that using these alternatives won't matter as long as the attacker can still distinguish between a packet-length check failure and a MAC failure. The attacker can just inject a random IV block, instead of using the last packet's last ciphertext block.

Furthermore, control over the IV gives an attacker more advantages. For example, he can distinguish whether a message M_1 or message M_2 was encrypted.

Albrecht et al. show that SSH-CTR, SSH-CTRIVCBC and SSH-EIV-CBC are resistant to

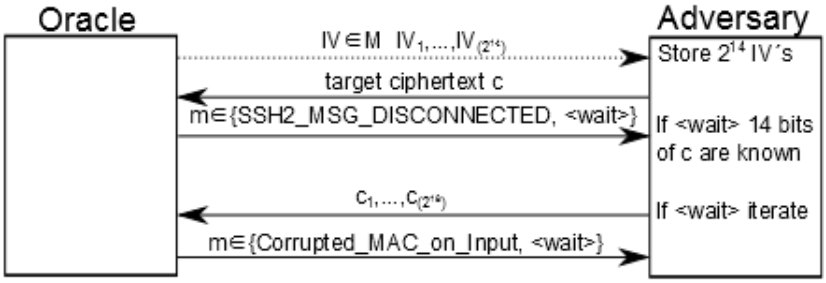

Figure 2: Attack on the SSH BPP protocol [1]

their attack. SSH-CTR because it uses a counter mode. SSH-CTRIVCBC and SSH-EIV-CBC because the attacker is unable to see the IV since it is either the encryption of a counter or encipherment of the last ciphertext block.

OpenSSH 5.2 switched to AES counter mode, but currently no SSH implementations use either SSH-CTRIVCBC or SSH-EIV-CBC.

V. Countermeasures

One of the countermeasures OpenSSH took was to return the same error message when either the length check or the block-length check failed. By not having this distinction, an attacker can't apply the first attack to reveal the first 14 bits. However this doesn't prevent the 32-bit recovery attack. An extra improvement is to randomize the length field if the length check fails. The system then proceeds with the new length until the MAC check fails. This will causes a problem for our attacker since it now doesn't know if the length is accepted and the MAC tag just failed or the length isn't accepted and it just returns the MAC error.

VI. Conclusion

It is possible to recover plaintext bits from a proven secure SSH implementation. Therefore it is also hard to know whether improvements to resolve this issue won't lead to new attacks on these SSH implementations.

References

[1] M. R. Albrecht, K. G. Paterson, and G. J. Watson, "Plaintext recovery attacks against ssh."

[2] T. Ylonen, "The secure shell (ssh) transport layer protocol."

[3] D. J. Barret and R. E. Silverman, *SSH The Secure Shell, The Definitive Guide*.

[4] M. Bellare, T. Kohno, and C. Namprempre, "Breaking and provably repairing the ssh authenticated encryption scheme: A case study of the encode-then-encrypt-and-mac paradigm," *ACM Transactions on Information and System Security*, vol. 7.