Forensic Science International: Digital Investigation 40 (2022) 301337

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS 2022 EU - Selected Papers of the Ninth Annual DFRWS Europe Conference

SSHkex: Leveraging virtual machine introspection for extracting SSH keys and decrypting SSH network traffic

Stewart Sentanoe [a, *], Hans P. Reiser [a, b]

[a] *University of Passau, Innstr. 43, 94 032, Passau, Germany*
[b] *Reykjavík University, Menntavegur 1, 102 Reykjavík, Iceland*

ARTICLE INFO

Article history:

Keywords:
Virtual machine introspection
Secure shell
Session keys
SSH Network packet decryption

ABSTRACT

Nowadays, many users are using an encrypted channel to communicate with a remote resource server. Such a channel provides a high degree of privacy and confidentiality. Secure Shell (SSH) is one of the most commonly used methods to connect to a server remotely. SSH provides privacy and confidentiality by encrypting network traffic between the client and the server. The encryption makes the learning process of malicious activities over SSH is challenging, especially by just analyzing the network traffic. To overcome the problem, we can leverage Virtual machine introspection (VMI). VMI allows direct memory access of a virtual machine (VM) including accessing data of an SSH process. However, the current prototype suffers from high overhead since it extracts every single plain text SSH network payload from memory and the extraction process requires the virtual machine (VM) to be momentarily paused. In this paper, we introduce *SSHkex*, a tool that also leverages VMI to extracts SSH's session keys from a server's memory. Our approach only needs to pause the VM twice to extract the session keys for each SSH session and does passive network monitoring where does not have any noticeable impact on the ongoing connection. To use *SSHkex*, zero modification needs to be done to the server. Thus, it is suitable for intrusion detection systems and high-interaction honeypot where the server shall not be modified.

1. Introduction

Encrypted communication has become a de-facto standard on the Internet. This development has significant advantages in terms of IT security, but creates novel challenges for digital forensics. A large amount of prior work has focussed on the TLS protocol on the Internet, and a number of approaches and tools are available for intercepting, decrypting, and investigating TLS connections.

Secure Shell (SSH) is a network protocol that enables secure remote access to computers. It supports applications such as command-line login, remote command execution, secure file transfer, and also arbitrary tunnelling of network traffic. This broad application spectrum makes it a versatile tool that is often used on the Internet.

SSH plays, however, an important role in the context of malicious activities on the Internet as well. The SSH service can be a target and entry point for attackers. Poorly configured servers might become victims of brute-force password guessing attacks. Often it is even possible to find devices deployed in the public Internet, with default credentials for administrative logins in place. Attackers can also use SSH for transferring second-stage attack payloads or interacting with remote command&control servers. Investigating encrypted SSH communication can thus be an essential part of digital forensics and the analysis of malicious attacks. It can also be an essential part of a high-interaction honeypot and potentially useful in an automated intrusion detection system.

This paper focuses on the investigation of encrypted SSH communication. In the context of a forensic investigation, there are a number of requirements for such investigation:

- Stealthiness: The attacker should not (or not easily) be able to detect the ongoing investigation.
- Preservation: The investigation should avoid modifications on the system under investigation.
- Evidence integrity: The investigation should yield evidence of verifiable integrity.

* Corresponding author.
E-mail addresses: se@sec.uni-passau.de (S. Sentanoe), hr@sec.uni-passau.de (H.P. Reiser).

https://doi.org/10.1016/j.fsidi.2022.301337

There are several potential approaches for obtaining decrypted SSH traffic. The most well-known methods are *active man-in-the-middle* and *binary manipulation*. These two approaches, however, have each specific weaknesses and limitations.

Active man-in-the-middle (MITM) is based on a proxy server between the client and the server (Kaiser, 2020). The MITM server has to decrypt and re-encrypt every SSH packet that is transmitted between the client and server. It is easy to deploy, but it also has a severe disadvantage: The user can easily detect the approach. SSH contains a mechanism for verifying host keys, and if someone connects to the MITM proxy server instead of the real remote server, the different host key is easily detectable.

The binary manipulation approach is based on modifications to the SSH server (and/or client) implementation in use. With the addition of code that exfiltrates all decrypted data packets, a complete communication dump can be generated. While such modification can be done with just a few lines of code in the case of an open-source SSH implementation, the difficult part is the deployment. The modified SSH binary needs to be deployed on the machine under investigation, which is an intrusive operation, and which requires privileged access on the target system.

In this paper, we explore a different approach based on the extraction of SSH session keys from the main memory. Our *SSHkex* prototype detects the establishment of SSH connections, extracts information about which cryptographic algorithms are used by SSH, and then leverages virtual machine introspection (VMI) to read the memory of a virtual machine (VM). To extract the SSH session keys, *SSHkex* uses knowledge about the SSH implementation and is able to directly pinpoint the location of the SSH keys in memory and extract them efficiently. We also passively capture the encrypted network traffic of SSH in standard PCAP files and then are able to decrypt the traffic using a Wireshark plugin that makes use of the extracted session keys.

Our approach has a number of advantages: It does not intercept communication in a potentially detectable way. It also does not require binary modifications of the SSH service in the target system. The approach produces a small overhead on the SSH session. Our performance evaluation shows that *SSHkex* produces an additional less than 100 ms delay of an SSH connection and takes a small amount of time ($<0.1ms$) to decrypt a network packet. Our approach is suitable for advanced intrusion detection systems and high-interaction SSH honeypots where there should be no modification inside the server.

The main contributions of this paper are:

1. A systematic method for extracting SSH session keys from the main memory of a target system using VMI, exploiting a priori knowledge about user-level data structures of an SSH server;
2. A stand-alone application and a Wireshark plugin that allow decrypting SSH network packets in a standard PCAP network dump using the extracted session keys.

The rest of the paper is organized as follows: Section 2 introduces background knowledge used in the following sections; Section 3 describes an overview design of *SSHkex*. Section 4 discusses our implementation in detail. Section 5 discusses the results and the performance of our methods; Section 6 reviews related work, and Section 7 concludes the paper.

2. Background

This section provides relevant background for this paper. We explain the relevant details of the SSH protocol, in particular the session key management, and specific implementation details of OpenSSH, which we use in our prototype implementation. In addition, we give a short introduction to virtual machine introspection (VMI), as we use this approach for extracting the SSH session keys from main memory.

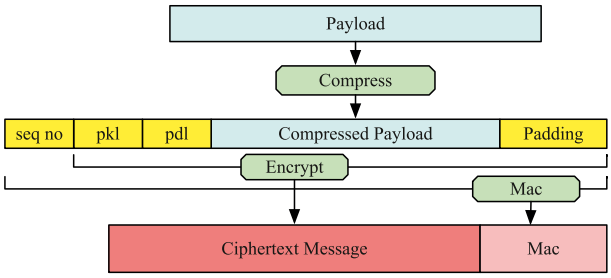

Fig. 1. SSH transport protocol packet layout.

2.1. Secure Shell (SSH)

Secure Shell (SSH) is a network protocol to establish secure remote connections over an insecure network. SSH was designed to replace telnet and unsecured remote shell protocols such as *rlogin*, *rsh* and *rexec*.

When an SSH connection is established, the client and the server will start to exchange data (referred to in the following as packets) over a TCP connection. Each payload message is first compressed and then concatenated with the following elements (see also Fig. 1):

Packet length (pkl) holds the length of the packet itself, not including the packet length field or the message authentication code (MAC) field
Padding length (pdl) holds the length of the padding field
Compressed payload holds the actual payload data of the packet. If compression has been negotiated, this field is compressed.
Padding contains random bytes of padding so that the total length of the packet (excluding the MAC) is a multiple of the cipher block size or 8 bytes for a stream cipher
Message authentication code (MAC) holds the value of the MAC, in case its use has been negotiated. It is calculated over an (implicit) sequence number and all other (unencrytped) elements.

2.2. SSH key exchange and session keys

The SSH key exchange procedure is defined in RFC 4253 (Ylonen and Lonvick, 2006). As a result of the key exchange, the server and the client derived a master key K and a hash value h. The client can verify the authenticity of the server with the server public key, as the message exchange is signed by the server with its private key. However, the server cannot deduce anything about the client's authenticity. To overcome this problem, the SSH authentication protocol is used, which supports several authentication methods such as password authentication and public key authentication. The value of h of the first key exchange serves as a session ID for this connection. Later re-key operations can refresh the key material, but will maintain the session ID.

From the key exchange step, the values K and h and the session ID (denoted *session_id* below) are used to derive the SSH session keys. Those keys will be used to encrypt the network traffic. The keys are computed as follows (Ylonen and Lonvick, 2006):

1. Key A (Initialization vector, client to server)
   $IV_{client2server} = Hash(K, h, \text{“A”}, session\_id)$
2. Key B (Initialization vector, server to client)
   $IV_{server2client} = Hash(K, h, \text{“B”}, session\_id)$
3. Key C (Encryption key, client to server)
   $EK_{client2server} = Hash(K, h, \text{“C”}, session\_id)$
4. Key D (Encryption key, server to client)
   $EK_{server2client} = Hash(K, h, \text{“D”}, session\_id)$
5. Key E (Integrity key, client to server)
   $IK_{client2server} = Hash(K, h, \text{“E”}, session\_id)$
6. Key F (Integrity key, server to client)
   $IK_{server2client} = Hash(K, h, \text{“F”}, session\_id)$

For decrypting encrypted SSH communication, it is thus either necessary to extract knowledge about the values used in the session key calculation (K, h, and $session_{id}$), or directly all the encryption keys (note that the two communication directions use different keys).

2.3. OpenSSH

OpenSSH is a freely available and widely used software implementation of the SSH protocol. OpenSSH includes a server component (*sshd*) and client tools (in particular, the remote shell client *ssh* and the secure file transfer *scp*).

OpenSSH used privilege separation such that a bug in an unprivileged child process does not result in a system compromise (Provos et al., 2003). Privilege separation uses two processes: A privileged parent process monitors the progress of the unprivileged child process as shown in Fig. 2 and Fig. 3 left. The unprivileged child process has restricted access to the file system and handles all network data originating from the client during authentication. The parent process will decide whether the authentication is successful or not. When the user is authenticated, the privileged parent (monitor) forks a new process that changes its user id and privileges to that of the authenticated user (Provos et al., 2003), as shown in Fig. 3 right. This knowledge is essential for the approach presented in this paper since we extract data from the memory that might exist only inside the privileged process and not inside the unprivileged process or vice-versa.

2.4. Virtual machine introspection (VMI)

Virtual machine introspection is the process of examining and monitoring the virtual machine (VM) from the hypervisor's point of view. The hypervisor allows a guest operating system to run and maintain control of the resources (Pfoh et al., 2009). Such architecture allows the hypervisor to have a complete and untainted view of all guest VM's state information (Pfoh et al., 2009). With such capability, the hypervisor can observe and analyze the guest operating system from the outside (Garfinkel and Rosenblum, 2003).

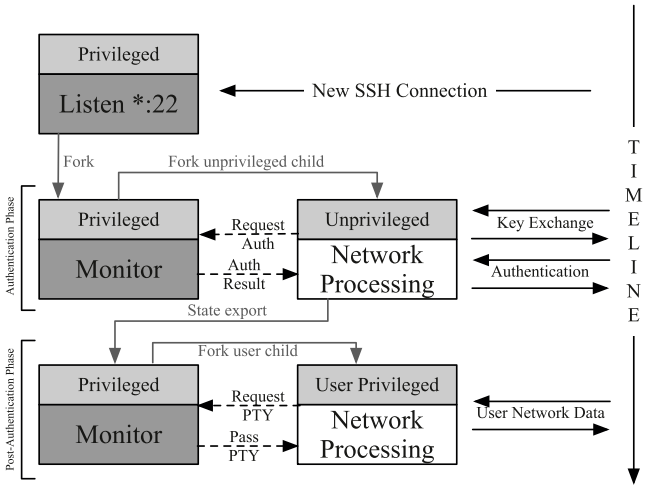

Fig. 2. OpenSSH privilege separation flow.

```
root  /usr/sbin/sshd -D
root          \_ sshd: user1 [priv]
sshd          \_ sshd: user1 [net]
```

```
root /usr/sbin/sshd -D
root  \_ sshd: user1 [priv]
user1       \_ sshd: user1@pts/2
user1          \_ -zsh
```

Fig. 3. OpenSSH processes during and after the successful authentication phase.

There are several ways to detect activities on a VM using VMI. Two of those are monitoring the execution of system calls (Sentanoe et al., 2017) and monitoring function calls (Sentanoe et al., 2018). By intercepting system calls or function calls, and analyzing the parameters, we can get an overview of what is happening inside a VM (Hofmeyr et al., 1998; Kosoresow and Hofmeyer, 1997). There are some tools that do VMI such as: LibVMI (Payne, 2012) and Volatility (Volatility Foundation, 2007).

3. Design of *SSHkex*

In this section, we explain the high-level goals of our approach, the assumptions we make, and the overall design of *SSHkex*.

3.1. Goals and assumptions

The core goal of *SSHkex* is to use standard approaches for capturing network traffic and to combine them with dynamic SSH session key extraction from the main memory using virtual machine introspection (VMI).

Our first assumption is that means for passively monitoring the network traffic exist. All network traffic of communication with the SSH server can be captured in the standard PCAP format for further processing. This assumption can easily be met by almost any system.

The second assumption is stricter: We assume that we know which SSH implementation is running in the server. In the next section, we will explain in detail how *SSHkex* exploits knowledge about data structures of the SSH service in order to extract SSH session keys from the main memory target system efficiently. Specifically, we assume to have knowledge about which version of the SSH implementation is used in our target system, and we assume that we have information about debugging systems of that implementation available.

The second assumption is usually satisfied if the target system is running an off-the-shelf version of the SSH software from a standard Linux distribution. In the case of a custom, possibly modified SSH software in the target system, some process for passing the necessary debug symbol information to our software is needed. The main focus of our work, however, is on target systems that use well-known standard software. It is, therefore, a valid assumption in most realistic use cases.

The third assumption is that we have VMI functionality available to access the target systems. Our prototype implementation is based on the well-known LibVMI, which, for example, supports introspection on virtual machines on the Xen and KVM hypervisor.

3.2. Main components

Fig. 4 illustrates the high-level design of *SSHkex*. The main components are the following:

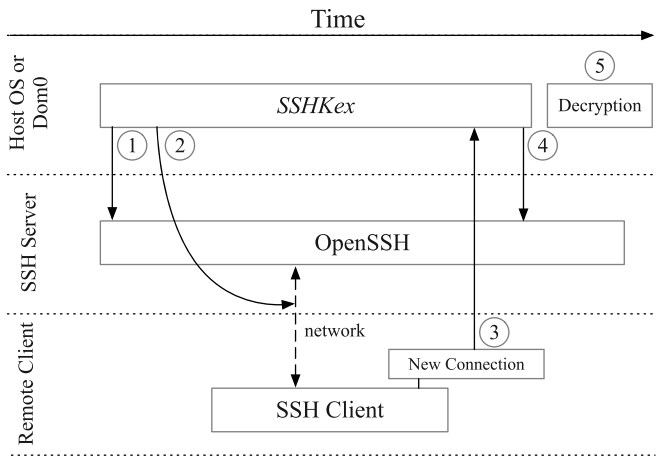

Fig. 4. *SSHkex*'s high-level design.

① Setup: The setup is responsible for setting up the monitoring of SSH processes in the target VM using active VMI.
② Network logging: The basis of our analysis is a direct capture of the network traffic. We can use any standard tools that record the traffic into PCAP files. Capturing network traffic is also a standard approach in digital forensics, and established means for protecting the integrity of captured traffic can be used directly.
③ Key capture trigger: We have to extract the keys at the right moment in time. Only a short time after the initial network layer connection, the key material will have been generated and be present in the main memory, and after the connection is closed, the key material may rapidly be cleared from the main memory. In *SSHkex*, we use control flow tracing of the SSH process to extract information about memory locations used for storing the SSH session keys and determining when the key generation has been completed.
④ Key extraction: Based on information collected in the preceding step, we can directly extract the SSH key material from main memory and store it in a file for later network traffic decryption.
⑤ Decryption: For our research, the decryption is done separately. To make it easy to use, we have developed two decryption tools: a stand-alone Python program and a Wireshark plugin.

4. *SSHkex* implementation details

This section explains in more detail our implementation, how we extract the SSH session keys efficiently from the memory of a Linux machine and how we decrypt the SSH network packets. The full source code is available at: https://github.com/smartvmi/SSHKex.

4.1. Using data structure information

The goal of *SSHkex* is not only to locate the key material in the main memory (this could, for example, be done with some brute-force search), but also to do so fast and efficiently. For that purpose, we leverage knowledge about the data structures used to store the key material.

We assume that we have debugging symbol information available that matches the version of the SSH implementation in the target system. We use libdwarfparser (Kittel, 2014) to extract information about data structures and obtain offset values of specific data structure elements within a structure.

Our prototype has been developed and tested with OpenSSH. For that implementation, the following data structures are of particular interest:

1. *struct ssh* is declared in *packet.h* file and contains remote IP and port (variable name: *remote_ipaddr* and *remote_port*), local IP and port (variable name: *local_ipaddr* and *local_port*), the memory address of *session_state* and *sshbuf*.
2. *struct session_state* is declared in *packet.c* file and contains the session keys (variable name: *newkeys*) for a particular SSH session.
3. *struct newkeys* is declared in *kex.h* file and contains the *struct sshenc* that holds information about the encryption keys
4. *struct sshenc* is declared in *kex.h* file and contains the encryption method (variable name: *name*), the initialization vector (variable name: *iv*) and encryption keys (variable name: *key*)

Note that the type information about data structure does not tell us directly where in the main memory to find the desired information. It tells us only where the elements of interested are located, relative to the beginning of the data structure.

4.2. Tracing OpenSSH functions

For efficiently locating the data structures and extracting the keys at the right point in time, we make use of tracing two OpenSSH functions that are used by the unprivileged sshd process (see Fig. 2) that handles client authentication and key generation:

4.2.1. kex_derive_keys

The function is declared in *kex.c* file. The function is called when the generation of session keys process about to begin. The input parameters are the memory address of *ssh* struct, hash (*h*), hash length and the address of *sshbuf* that contain the shared secret (*K*).

4.2.2. do_authentication2

The function is declared in *auth2.c* file. The function is called when the user authentication phase is about to begin. The input parameter is the memory address of struct *Authctxt*.

4.3. Breakpoints injection and key extraction

A software breakpoint is an intentional suspension of the program execution at a specific point in the execution flow, which can be used to explore the program state for debugging purposes (Zhang et al., 2013). Our approach makes use of breakpoints placed at the beginning of functions to implement function tracing. We set up these breakpoints using VMI, and for that purpose we need the information about where the two relevant functions (see Section 4.2) are located in the memory.

We obtain this information from the OpenSSH debugging symbols retrieved from the package repository of the Linux distribution in use. The breakpoint injection steps are as follows:

1. Read the offsets of the functions from the debugging symbol;
2. Find the executable memory region of the *sshd* process;
3. Add the start of the executable region to the offset to get the exact location of the functions in the process memory;
4. Read the original opcode byte at that location, save it, and then replace it with the opcode *0xCC* (*int3*);
5. When the breakpoint is hit, extract the needed data from the main memory;
6. Replace the *0xCC* with the original byte;
7. Proceed with a single-step operation;
8. Replace the original byte with *0xCC* again;
9. Resume the execution.

4.4. Key extraction

When the *kex_derive_keys* function is called, we extract and store the address of the *ssh* struct. Then, we let the process continue until *do_authentication2* function is called. Next, we extract the keys from the memory by following the struct's structure as shown in Fig. 5. OpenSSH stores the *client2server* and *server2client* keys on the different index of *newkeys* (index 0 and 1, respectively). Key A and C are stored in index 0 in the *iv* and *key* fields, respectively. Key B and D are stored in index 1, also in the *iv* and *key* fields. Our prototype also extracts key E and F, but currently these keys are not used for further steps. OpenSSH has also a *rekeying* feature where a new set of keys are generated and used in the middle of a long session. We also do not address this feature on this research but, to handle this we need to intercept the function that responsible to do the *rekeying*.

To handle multiple connections, we extract each keys based on their own process ID since OpenSSH creates a child process to handle each connection.

4.5. Decryption

We make two decryption tools for this research. The first is Python-based with the Pycryptodome library, and the second is a Wireshark plugin.

The Python-based tool serves as a proof-of-concept and to test whether we have the correct keys. To use it, the user has to manually input the keys and the cipher-text. Then, the program will print the plain-text.

For the Wireshark plugin, first, the user has to install the plugin (see on our repository). Second, load a PCAP file or do live network sniffing. Third, insert the SSH session keys. Lastly, the plugin will decrypt and print the plain-text. Wireshark uses data sources for every captured packet in which the relevant data (encrypted packet data, length, header information) is saved. In the plugin implemented here, this data source is extended to include also the decrypted data.

5. Result and discussion

In this section, we discuss the result of our approach to extract the SSH keys from the memory, the decryption result, and the performance benchmark.

5.1. Experimental setup

For this paper, we use Xen 4.11 as the hypervisor. We run two Ubuntu VMs with 256 MB of RAM and 1 vCPU, running OpenSSH 7.2p2. One VM acts as the SSH server that is actively monitored by *SSHkex* (as shown in Fig. 6). The network monitoring is running on the host system that has access to the network interface of the VM.

5.2. Extracted keys

Listing 1 shows the extracted *ChaCha20-Poly1305* keys. Our system extracts the name of the encryption algorithm and all keys (A-F), and we encode it in hex format.

Listing 1. Extracted Chacha20-Poly1305 keys

```
Key 1 name : chacha20-poly1305@openssh.com
Key 2 name : chacha20-poly1305@openssh.com

key : A --- addr : 5588ac9fcde0 len : 64
93D20734C5F83B5BC81DD1816A042BB4624D1296BC1F8E
93FF02EA133CEC2F6B07922928ABEE64EC59DFC4BB11301
7284D6CAC9548BE9A8C6A9AC4EFB0D0035B
key : B --- addr : 5588ac9f8ac0 len : 64
A196F1B4A72308A8EC41AC72A84FA222CD699EF83F3CDE
7789AB37B47113CED57A67396EF07ACA52FE2B5D96F2E3
D5E3AD982B6F32850472AA4B29C1221CDCD4
key : C --- addr : 5588ac9fed90 len : 64
0B77E10E49CAB165B93CF0C861FEB38C097F3F8F8E6AA7
FFF4D979ABC5F3164A8136FE3C48DFE86CF59710BA7998
FA996B40AC21B50E5B2F6A799F08C7C3E183
key : D --- addr : 5588ac9fa810 len : 64
8F84E1EEFD3E09A4F8A338DDF7E47A59808B0C96020595
DA70F073A5830A7ABE666C4FB6BE2F60DD837C63E2DC8
766B6BF62DFBD77BDA98BD1FCEB791E9442B8
key : E --- addr : 5588ac9fce30 len : 64
D446F576112F2B09C6758D1A8ABD70ED4B945A8298CD6
7F8102ABDED23779132C4AC8CB1483C2D22533AC4BE1B
33F838B97FBBA624900A13D19063C99F8E8693
key : F --- addr : 5588ac9fbd00 len : 64
4FFE294F171DBAD11CA45B289BD2E27C3E48BEF73E1D1
C503CDF726BA0071998C71AD9A2066D2613B130CD3122
6F6FDF927BA38202BCCDFE6D25ED380644186C
```

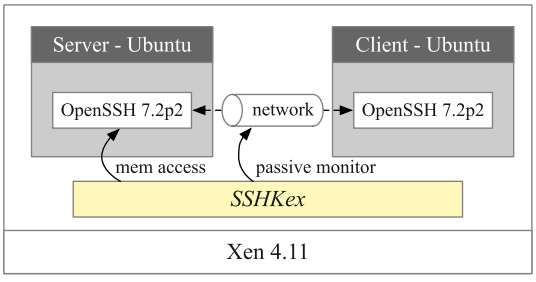

Fig. 6. Experimental setup.

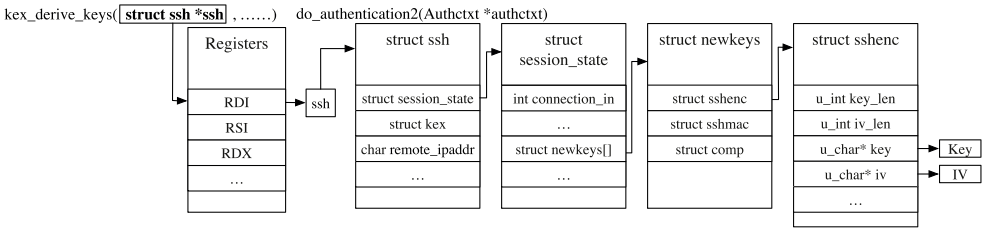

Fig. 5. OpenSSH's key extraction flow.

5.3. Decrypted packet — tcpdump

Using *tcpdump* to capture the network traffic and our Python-based decryption tool, we are able to decrypt the SSH network packets using the extracted session keys. Listing 2 shows an encrypted (with Chacha20-Poly1305) SSH packet and Listing 3 shows the decrypted packet that still has the padding length's value and the padding.

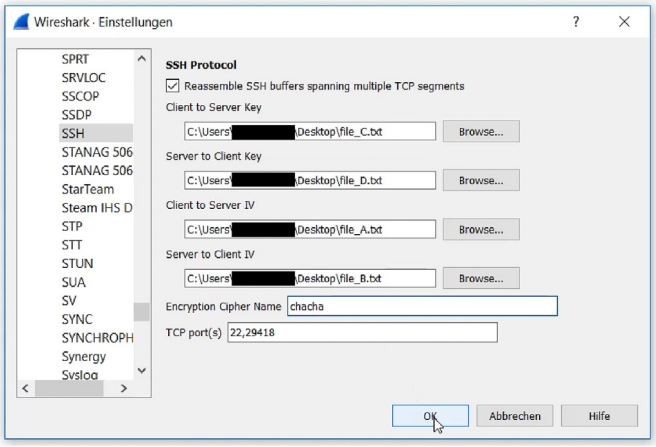

Fig. 7. Wireshark's user interface (key setup).

Listing 2. Encypted SSH Packet with Chacha20-Poly1305,

```
09:55:19 IP 192.168.12.46.ssh > 192.168.12.29.60660:
tcp 76
0x0000:  4548 0080 bc22 4000 4006 e471 c0a8 0c2e
0x0010:  c0a8 0c1d 0016 ecf4 5a30 7e13 4bac 0d86
0x0020:  8018 0487 9a0e 0000 0101 080a 22c4 d660
0x0030:  8cb4 0c8a 1a00 37ab f298 07d5 661e 26ed
0x0040:  e007 2618 0c96 0bc6 35c6 2d06 9a0f 546e
0x0050:  4b7a ae5d 4440 90c2 445c 8da9 c998 0138
0x0060:  bf0b 9220 9256 e3c7 bb18 0ddf 9591 6bcd
0x0070:  8dab 2f73 07ef 0916 1c33 298e 9eb8 1985
```

Listing 3. Decrypted packet with the padding length's value and the padding

```
b'\x04^\x00\x00\x00\x00\x00\x00\x00*This is a
test. Just a test. One two three\x8btf\xc8'
```

5.4. Decrypted packet — Wireshark

Fig. 7 and Fig. 8 show the user interface of Wireshark with our decryption plugin; (1) shows that we filter the SSH connection from a client to our server, (2) shows the packet list, (3) shows the packet list of the connection of the server, (4) shows the packet details where we can see it is an SSH connection, (4) shows the packet in hex representation, (5) shows the decrypted packet, and (6) shows the frame number and the decrypted packet's size.

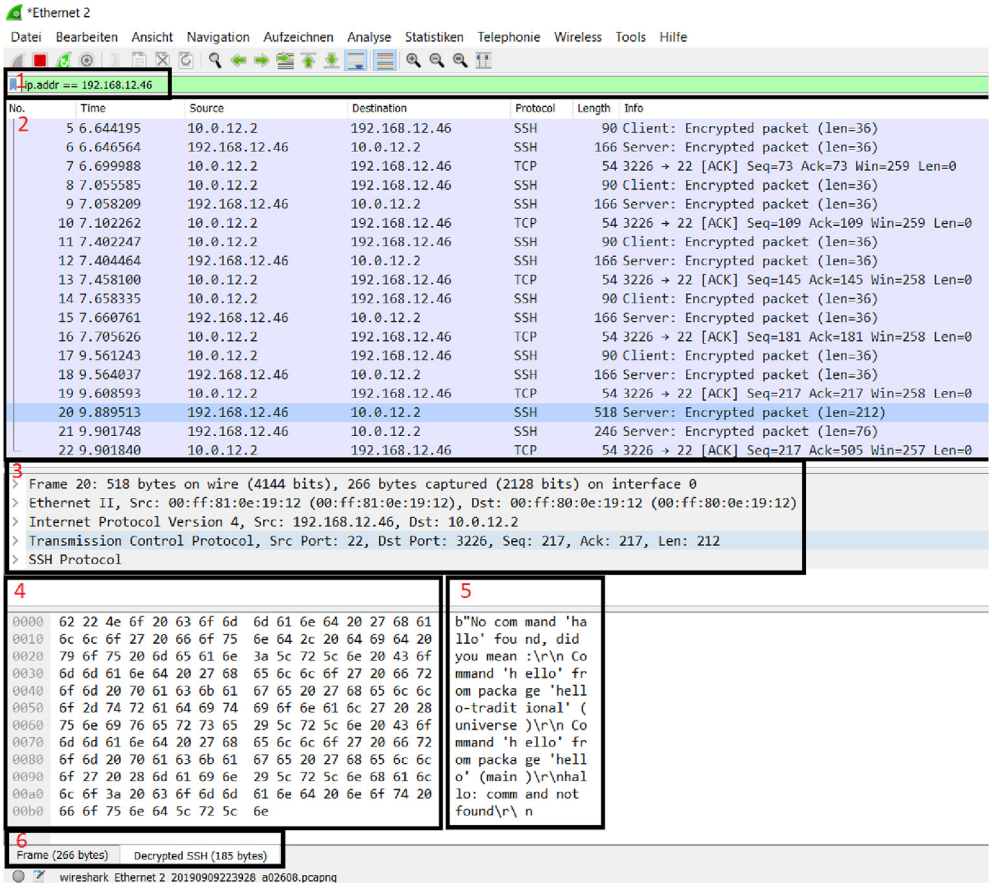

Fig. 8. Wireshark user interface (packet decryption).

5.5. Performance test

We measure the performance of our system by calculating the connection time from the client side with the public key authentication method. There are several scenarios:

S1 Establish a ssh session (with AES128-CTR) and exit immediately.
S2 On the SSH server, we execute the command *git clone https://github.com/wireshark/wireshark.git* and the client ssh connection in parallel. The goal is to produce loads on the monitored server.
S3 Similar to S1 but with AES256-CTR.
S4 Similar to S2 but with AES256-CTR.

We execute each scenario 100 times and the average result is shown in Fig. 9. In the best-case scenario, our system increases the connection time by 50 ms. On the other hand, when the server has more loads, our system produces 85.8 ms overhead. This result shows that our system has a small and unnoticeable impact on the server.

We measure the decryption time using Pycryptodome with multiple ciphers as shown in Table 1. *pkl* and *pyl* denotes packet and payload length in bytes, respectively. The decryption of each packet is fast and since it is done in parallel, it has no impact on the connection performance.

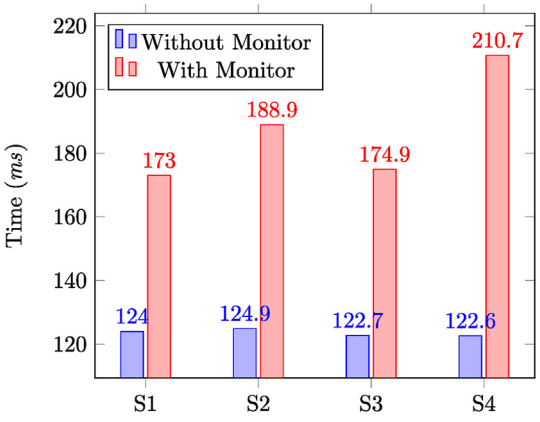

Fig. 9. *SSHkex*'s performance test result.

Table 1
Single SSH network packet decryption time.

pkl	pyl	Time (*ms*)
(a) AES128-CTR		
28	16	0.030
44	32	0.036
76	64	0.043
124	96	0.052
(b) AES256-CTR		
28	16	0.031
60	48	0.037
76	64	0.041
132	32	0.058
(c) ChaCha20-Poly1305		
28	16	0.066
56	16	0.070
76	56	0.069
132	32	0.079

5.6. Honeypot integration

In 2018, we introduced *Sarracenia* (Sentanoe et al., 2018), a high-interaction SSH honeypot. It extracts the decrypted packet from the main memory, requiring the VM to be paused for every incoming or outgoing packet, which degrades the server performance. By using *SSHkex*, instead of extracting each decrypted packet from the main memory, we extract the derived SSH session keys and then decrypt the network packets on a separate process. Hence, the VM is paused less frequently (only during SSH session keys extraction). Thus, this approach significantly reduces the performance degradation of the server performance.

6. Related work

McLaren et al. (2019) introduce *Memdecrypt*. They extract the main memory content and find the SSH session keys and IVs. They assume that the cryptographic keys have high entropy and remain static during the duration of a session. Also, keys and IVs are located close to each other in memory. They run a decryption test for each potential key until they find the correct one. The duration of the memory analysis varies a lot and can take several seconds, but in ideal, optimized conditions, can be as low as about 1 s. In contrast, on our approach, we directly have the precise location of the keys, which enables us to extract the keys in less than 100 ms.

Taubmann et al. (2016) introduce *TLSkex*, which decrypts TLS-based connections by taking a snapshot of the memory during run-time and finding the key by brute-force search of its location in memory. It takes 48 bytes sequences in the snapshot as key candidates and tries to decrypt the packet. If fails, it takes the next sequence until the correct key is found.

Taubmann et al. (2018) introduce *DroidKex*. DroidKex first links data structures using pointer paths that they found during a training phase. Next, it follows the path and extracts they key. Finally, it decrypts the packet using the extracted key.

Sentanoe et al. (2017) rebuild an SSH session by intercepting system calls. This method is effective, but it is not efficient since intercepting the control flow of the target VM for each network packet produces high overhead.

Man-in-the-middle (MiTM) can also be used to rebuild an SSH session (McMurray, 2016; Testa, 2017). It acts like a proxy server and stands between the client and the user. The problem is when a user has connected to the server before, it will receive a warning because now it will connect to a different server (the proxy).

Albrecht et al. (2009) exploit a weakness in a OpenSSH implementation to reconstruct a few plaintext bits using a known ciphertext block. Such an approach does not require knowledge of the decryption keys and does not need to run on virtual environment. However, it does not permit the full recovery of all plaintext, and it will not work with newer versions of OpenSSH that contain mitigations against such attacks and is not able to decrypt newer encryption algorithm such as ChaCha20.

Jelle (Vergeer, 2020) uses a method similar to our approach to extract the SSH keys. It uses a memory dump and scans the entire memory address space of the program and looks for the session keys by pattern matching. In our approach, we directly have the precise location and can be done on a running machine.

7. Conclusion

Monitoring encrypted SSH communications is essential for purposes such as attack analysis and digital forensics. Existing

approaches based on man-in-the-middle interception or modification of SSH binaries have the disadvantage of being visible in the target system and thus detectable by the attacker. Existing VMI-based approaches offer better stealthiness, but are either inefficient (such as brute-force key extraction) or have a high impact on the target system (such as tracing all network I/O operations and extracting plaintext from memory using VMI).

With our paper, we demonstrate that a novel approach offers superior properties while maintaining the stealthiness of VMI-based monitoring: We make use of a priori knowledge about data structures used in the SSH implementation in the target system. By applying function tracing to selected SSH functions used during connection establishment, we limit the (small) VMI overhead to a short moment at the beginning of an SSH session. The interception allows us to directly extract the SSH session keys, which then can be used for decryption.

With our *SSHkex* prototype implementation, comprising a key extraction tool, a stand-alone decryption tool, and a Wireshark plugin for decryption, we have demonstrated that such an approach is feasible with LibVMI and a Linux target system. Based on network traffic captured as PCAP files, our approach is both effective and efficient: *SSHkex* is able to extract the keys accurately (effective) with a minimal overhead: less than 100 ms (efficient).

Our current approach is only feasible if the memory layout of the data structures used by the SSH implementation is known to the key extractor. As future work, we are confident that we will be able to address this limitation as well. We plan to replace the static knowledge about data structure layout with a learning face in which the control flow of the target system is recorded and a generic model is trained using machine learning.

Acknowledgement

This work has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project 361891819 (ARADIA) and the Bundesministerium für Bildung und Forschung (BMBF, German Federal Ministry of Education and Research) – project 01IS18068 (SORRIR). Additionally, we would like to thank Paul Nikolaus for his implementations that supported this project.

References

Albrecht, M.R., Paterson, K.G., Watson, G.J., 2009. Plaintext recovery attacks against SSH. In: 30th IEEE Symposium on Security and Privacy. IEEE, pp. 16–26.
Garfinkel, T., Rosenblum, M., 2003. A virtual machine introspection based architecture for intrusion detection. In: Network and Distributed Systems Security Symposium (NDSS), pp. 191–206.
Hofmeyr, S.A., Forrest, S., Somayaji, A., 1998. Intrusion detection using sequences of system calls. J. Comput. Secur. 6, 151–180.
Kaiser, M., 2020. SSH-MITM – Ssh Audits Made Simple. https://github.com/ssh-mitm/ssh-mitm/ [accessed 2021-09-06].
Kittel, T., 2014. Libdwarfparser. https://github.com/kittel/libdwarfparser/ [accessed 2021-06-06].
Kosoresow, A.P., Hofmeyer, S., 1997. Intrusion detection via system call traces. IEEE software 14, 35–42.
McLaren, P., Russell, G., Buchanan, W.J., Tan, Z., 2019. Decrypting live SSH traffic in virtual environments. Digit. Invest. 29, 109–117.
McMurray, S., 2016. SSHHiPot. https://github.com/magisterquis/sshhipot [accessed 2021-06-06].
Payne, B.D., 2012. Simplifying Virtual Machine Introspection Using LibVMI. Technical Report SAND2012-7818. Sandia report.
Pfoh, J., Schneider, C., Eckert, C., 2009. A formal model for virtual machine introspection. In: Proceedings of the 1st ACM Workshop on Virtual Machine Security. ACM, pp. 1–10.
Provos, N., Friedl, M., Honeyman, P., 2003. Preventing privilege escalation. In: USENIX Security Symposium.
Sentanoe, S., Taubmann, B., Reiser, H.P., 2017. Virtual machine introspection based SSH honeypot. In: Proceedings of the 4th Workshop on Security in Highly Connected IT Systems, pp. 13–18.
Sentanoe, S., Taubmann, B., Reiser, H.P., 2018. Sarracenia: enhancing the performance and stealthiness of SSH honeypots using virtual machine introspection. In: Nordic Conference on Secure IT Systems. Springer, pp. 255–271.
Taubmann, B., Alabduljaleel, O., Reiser, H.P., 2018. Droidkex: fast extraction of ephemeral tls keys from the memory of android apps. Digit. Invest. 26, S67–S76.
Taubmann, B., Frädrich, C., Dusold, D., Reiser, H.P., 2016. TLSkex: harnessing virtual machine introspection for decrypting TLS communication. Digit. Invest. 16, S114–S123.
Testa, J., 2017. SSH MITM. https://github.com/jtesta/ssh-mitm [accessed 2021-06-06].
Vergeer, J., 2020. Decrypting OpenSSH Sessions for Fun and Profit. https://research.nccgroup.com/2020/11/11/decrypting-openssh-sessions-for-fun-and-profit/ [accessed 2021-08-06].
Volatility Foundation, 2007. Volatility Framework – volatile memory extraction utility framework. https://github.com/volatilityfoundation/volatility [accessed 2021-06-06].
Ylonen, T., Lonvick, C., 2006. The secure shell (SSH) transport layer protocol. RFC 4253. RFC Editor. URL: http://www.rfc-editor.org/rfc/rfc4253.txt http://www.rfc-editor.org/rfc/rfc4253.txt.
Zhang, C., Yang, J., Yan, D., Yang, S., Chen, Y., 2013. Automated breakpoint generation for debugging. J. Softw. 8, 603–616.