Digital Investigation 16 (2016) S114–S123

Contents lists available at ScienceDirect

Digital Investigation

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

DFRWS 2016 Europe — Proceedings of the Third Annual DFRWS Europe

TLSkex: Harnessing virtual machine introspection for decrypting TLS communication

Benjamin Taubmann[*], Christoph Frädrich, Dominik Dusold, Hans P. Reiser

University of Passau, Innstr. 43, 94032 Passau, Germany

A B S T R A C T

Keywords:
Virtual machine introspection
Transport layer security
Decryption
Malware analysis
Virtualization
Semantic gap

Nowadays, many applications by default use encryption of network traffic to achieve a higher level of privacy and confidentiality. One of the most frequently applied cryptographic protocols is Transport Layer Security (TLS). However, also adversaries make use of TLS encryption in order to hide attacks or command & control communication. For detecting and analyzing such threats, making the contents of encrypted communication available to security tools becomes essential. The ideal solution for this problem should offer efficient and stealthy decryption without having a negative impact on over-all security. This paper presents TLSkex (TLS Key EXtractor), an approach to extract the master key of a TLS connection at runtime from the virtual machine's main memory using virtual machine introspection techniques. Afterwards, the master key is used to decrypt the TLS session. In contrast to other solutions, TLSkex neither manipulates the network connection nor the communicating application. Thus, our approach is applicable for malware analysis and intrusion detection in scenarios where applications cannot be modified. Moreover, TLSkex is also able to decrypt TLS sessions that use perfect forward secrecy key exchange algorithms. In this paper, we define a generic approach for TLS key extraction based on virtual machine introspection, present our TLSkex prototype implementation of this approach, and evaluate the prototype.

Introduction

The proliferation of encrypted communication in the Internet has led to an increase in security, but also to an increase in the difficulty of performing forensic investigations and analysis of malicious activity. Today's defacto standard for securing communication is transport layer security (TLS), which is used in a large variety of applications, including e-mail, instant messaging and VoIP communication. A recent NSS Labs report concluded that encryption using TLS "actually reduces security on the corporate network by creating blind spots for corporate security infrastructures" (Pric, 2013). It is likely that an attack against a web-based server uses TLS-based communication channels. Similarly, malware can use TLS channels for protecting information leakage or for communicating with a command & control server. A detailed investigation of such problems can benefit from the ability to decrypt the TLS-encrypted network traffic.

Among existing approaches for TLS decryption, which we discuss in more detail in the Section Related work, active TLS proxies are most likely the most practical approach. Such a proxy acts as a "man-in-the-middle", decrypting and re-encrypting the network traffic. In this paper, we investigate whether TLS-encrypted network traffic can be decrypted if using such a proxy is not feasible. Ideally, TLS decryption should work in a *non-intrusive* and *universal* way. Being non-intrusive implies the following two requirements:

[*] Corresponding author.
E-mail address: bt@sec.uni-passau.de (B. Taubmann).

http://dx.doi.org/10.1016/j.diin.2016.01.014

- *No active manipulation of communication*: The communication should be monitored passively without modifying the contents of the communication (we do not exclude a possible impact on the timing of messages).
- *No modification of application*: The decryption should work without internal modifications to the communicating applications (such as exporting the session key to a file).

Being universal implies the following four requirements:

- *Independence of specific key exchange*: The decryption key extraction should work for any key exchange algorithm.
- *Independence of encryption algorithm*: The decryption and key extraction should work independently of a specific cryptographic algorithm.
- *Independence of client/server role*: It should work for local applications that operate as a server, as well as for local client applications that connect to a remote server.
- *Independence of the implemenation*: The key extraction should work for every TLS implementation.

This paper proposes a novel approach for TLS decryption based on virtual machine introspection (VMI) and presents details of TLSkex, a prototype implementation of this approach. The proposed approach makes the following contributions. First, it proposes a method for inspecting network traffic to detect TLS connections and extracting essential information out of it. Second, it defines a strategy for minimizing the size of memory snapshots that potentially contain TLS session keys. Third, it proposes and compares various heuristics to minimize the time required for a brute-force search of cryptographic session keys in memory snapshots. Finally, it presents an evaluation of the prototype implementation and discusses benefits and limitations.

The remainder of this paper is organized as follows. In Section TLS internals we present some background about the TLS protocol. The basic concepts of TLSkex are discussed in Section Conceptual approach. The implementation details are presented in Section TLSkex implementation and evaluated in Section Evaluation. In Section Related work we compare our approach to related work in the fields of TLS decryption, cryptographic key extraction of main memory and VMI. We finally conclude our elaborations in Section Conclusion.

TLS internals

TLS is the successor of secure sockets layer (SSL). These cryptographic protocols provide a secure communication channel.

TLS uses cryptographic certificates based on asymmetric cryptography for server authentication and – optionally – client authentication. This means that all decryption attempts based on active man-in-the-middle intercepts with fake certificates can be detected, unless the interceptor has access to the private keys of the original endpoints.

After endpoint authentication, TLS negotiates a symmetric session key (master secret). This negotiation can be done with RSA encryption, or with the Diffie-Hellman (DH) or elliptic curve Diffie-Hellman (ECDH) algorithm. Nowadays, DH or ECDH should be the preferred way to negotiate a session key. In contrast to RSA, they have the advantage that even if an attacker obtains the private RSA key it is not possible to decrypt already captured connections as it is not possible to extract the session key from the network traffic. This property is called perfect forward secrecy (PFS).

Finally, the communication between the endpoints is protected with symmetric encryption and message authentication based on symmetric keys derived from the master secret. TLS does not rely on a single fixed cryptographic algorithm; instead it provides a flexible framework that can support a large variety of encryption algorithms. At the beginning of a TLS session, the endpoints negotiate which algorithms and parameters to use.

TLS records

Internally, TLS is composed of several sub-protocols. The lower protocol layer is the TLS record protocol, which is used to exchange control and data messages between the communication partners. Each message of the record protocol contains the content type of a record, the TLS version, the length of a data fragment, and the data fragment (compressed, integrity protected and encrypted using the negotiated algorithms). At this layer, only the data fragment is encrypted, whereas all other fields (record type, TLS version and length) are exchanged in plain text.

Key negotiation and derivation

The key negotiation process is used to exchange the cryptographic parameters of the upcoming encrypted network session. The client initiates the protocol by sending a TLS record with the content type Client Hello (CH) and the server responds with a Server Hello (SH). These messages do not only define which algorithms to use, but are also used to exchange a client and server random value. Afterwards, the client and server define a premaster secret, for example by using the DH algorithm. In the initial handshake of TLS, no encryption is used, and thus the client and server random value are exchanged in plain text. If keys are renegotiated on an already encrypted TLS channel, the parameters will be encrypted with the still active configuration.

The premaster secret, client random, and server random are used to calculate the master secret of a connection. The master secret is used together with the server and client random to compute the derived keys using a pseudo random function (PRF). Typically, they comprise a MAC secret key used to verify the integrity of a TLS record (*write_MAC_key*), the data encryption key used to encrypt the payload of a TLS record (*write_key*), and an initialization vector (*write_IV*).

In the TLS protocol, a dedicated message is used to signal the transition to new cryptographic parameters. After completing the computation of the master secret, each communication partner sends a Change Cipher Spec

(CSP) message to the other endpoint and starts encrypting all subsequent messages using the new cryptographic parameters. The CSP payload message itself, which consists of a single constant byte, is encrypted using the previous parameters. Note, that the message is sent using the record layer protocol using a dedicated record type, which makes it possible to detect any CSP message in an encrypted TLS channel without decryption.

MAC computation

TLS protects the integrity of each TLS record using a message authentication code (MAC). For not encrypted messages or messages encrypted with a standard stream cipher or CBC block cipher, TLS appends an HMAC (Krawczyk et al., 1997) to the payload data. The HMAC is keyed with a dedicated MAC secret derived from the master secret (as described above), and is computed over the implicit sequence number of the TLS record, the type of the record, the TLS version, the length and the unencrypted data of the record. To verify the integrity of a TLS record, the receiver can recompute the HMAC of a decrypted record and check whether it matches the sent one. For authenticated encryption with associated data (AEAD) ciphers, such as counter with CMC-MAC (CCM) or Galois/Counter Mode (GCM) an authentication tag is used to verify the integrity of a TLS record instead of a separated MAC.

Session resumption

TLS supports the resumption of previously established TLS sessions. In session resumptions, parameters and keys of a previous TLS session are reused instead of negotiating new values, resulting in a faster initial handshake. The session state includes the choice of cryptographic parameters as well as the values for client random, server random, and master secret. The session state can be stored either on the server and is identified using a unique ID that the client includes in its initial CH message (session ID), or on the client (session tickets) (Salowey et al., 2008).

Conceptual approach

This section presents all the components that are necessary to decrypt TLS secured connections (see Fig. 1) The TLS decryption process can be separated into two stages – an online and an offline part. The online functions must be executed synchronous to the TLS communication and include capturing of the network traffic, detecting TLS sessions, and taking a snapshot of the memory in which the key is stored, e.g., a snapshot of the whole virtual machine main memory. The offline part can be executed later, whenever the decrypted content of a TLS connection needs to be accessed. The process of extracting the master secret from memory uses the information captured by the online parts.

Network logging

Network logging is responsible for capturing the network traffic of all TLS connections that shall be monitored. This task can be executed by any standard network logging tool, such as a dedicated device on a promiscuous network switch port or a local capturing process on the host running a virtual machine.

Trigger mechanism

The acquisition of virtual machine memory must be triggered at the right point in time, when the key material is present in main memory. The master secret and the derived key material is available as soon as the TLS handshake for key negotiation has finished and the key calculation has been executed. According to the TLS protocol, a CSP message is sent when a node is ready to use new key material in subsequent messages. This means that the right moment for snapshot creation can be detected by monitoring the network traffic for TLS records that contains the CSP message.

During a TLS session the key material can be renegotiated. This means that the key extraction routine must be triggered each time when new cryptographic parameters are exchanged. We can detect the subsequent CSP messages even if these messages themselves are encrypted, because they are sent with a unique payload type in the TLS record layer header, which is not encrypted (as explained in the previous section).

Memory acquisition

The memory acquisition must be performed synchronous to the triggered snapshot request and network traffic. If it is performed asynchronously, the connection or the program might be terminated and the key could be gone. Thus, it is important that the memory acquisition is executed synchronous. Additionally, it is important to take the snapshot fast in order to decrease the impact on the timing of the network communication.

There are several ways how the time required to take a snapshot can be decreased. For example, LibVMI supports copy on write snapshots of virtual machines. But this feature is currently not implemented for Xen (Xu et al., 2013).

The other way to decrease the time is to reduce the size of the snapshot. However, this requires contextual knowledge about the guest operating system. This includes for example to know which process is communicating and where its memory is located in physical memory.

Key extraction

There are several ways how the master secret can be obtained from a memory snapshot. In contrast to RSA keys, there is no standardized way to store the master secret of a TLS session. Thus, there is no general approach to find it with a simple pattern matching approach. But there are several other ways how this can be achieved.

One way is to parse the structures of a process in order to find the TLS session structure. This requires contextual knowledge about the program. Thus, this approach is not feasible for unknown programs such as malware. However, this approach is very fast because no complex computation is required. The search routine only needs to follow pointers in memory.

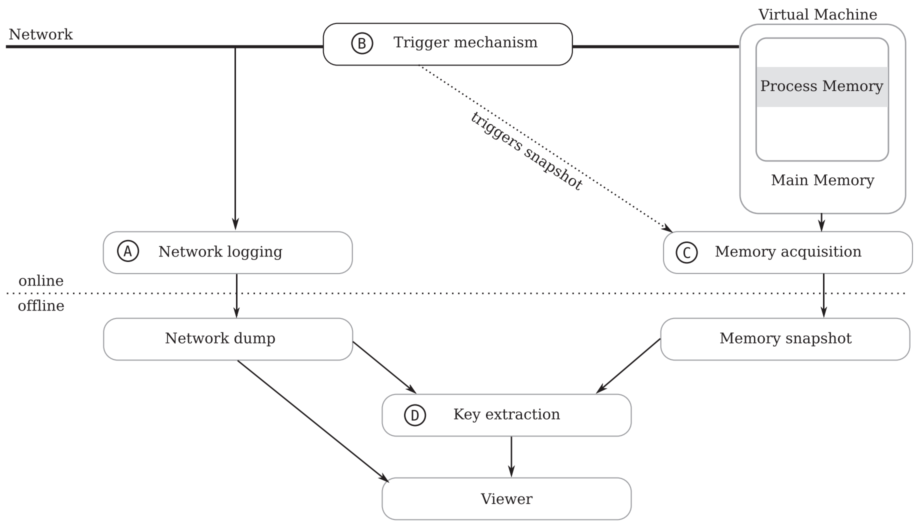

Fig. 1. The main steps for decrypting TLS connections: network logging, TLS detection, memory acquisition, and key extraction.

A similar alternative is to search for well-known TLS session structures of different implementations that include the master secret. These sessions structures often contain values such as the TLS version or the IP address of the communication partner. Thus, these structures can be found easily when some parts of them are known (Homan, 2013). This approach requires searching for parts of the key structure in the whole address range of a process but no complex computation is required to identify the key. However, the key structure must be known a priori. If malware uses an unknown TLS implementation, this approach does not work.

When no a priori knowledge about a process is given, there is still the option to try every byte sequence as a potential master secret. This approach is slow as the testing of a byte sequence includes the key derivation and the decryption of a data block. Thus, first of all, the size of the snapshot should be reduced to memory areas which potentially contain the key. For example, in most of the cases it would not make sense to search the master secret in the read-only mapped text segment of a process. With a high probability it is stored in a memory region that is write able as it is negotiated dynamically at runtime. This approach can be further optimized with heuristics that filter byte sequences that are no potential keys, e.g., by checking the entropy.

After a potential master secret was found in memory, it must be tested whether it fits to the corresponding connection. This can be achieved by decrypting a TLS record and verifying the HMAC that is included in every TLS record. As the HMAC is computed over the sequence number of a TLS record, this number must be known. Thus, the first CSP message should be used as it has always the sequence number zero.

TLSkex implementation

TLSkex implements the concepts described in the previous section. It is a framework written in the programming language C that uses LibVMI in order to access the memory of a virtual machine running under the XEN hypervisor.

TLSkex has been created with the focus on the following goals: It shall obtain the master secret of a virtual machine without active manipulation of the TLS channel itself, and without manipulation of the communicating application. TLSkex shall work independent of the specific key exchange mechanism and selected cryptographic algorithms and independent of whether a client or a server application runs within the virtual machine.

Trigger mechanism

We need to trigger the snapshot process after observing a network packet containing a CSP message, but before the connection is closed and the master secret removed from main memory. It is not sufficient to monitor the network passively and trigger the key extraction process asynchronously. In such a passive approach, triggering the creation of the memory snapshot might take longer than the lifespan of the TLS connection, and thus fail to capture the master secret. This is especially a problem for very short living connections. Instead, we implemented an *active network monitoring* approach.

Thus, TLSkex includes an active network monitoring component that is able to analyze every packet coming from or to the monitored virtual machine. The network analyzer forwards packets only after they have been inspected. It is equipped with two virtual TUN network interfaces. One interface is bridged to the virtual machine and the other interface is bridged to the rest of the network. In general, the network analyzer simply receives a packet from one interface, analyzes its content, and writes it to the other one.

The network analyzer must recognize various types of TLS messages. These messages together with the corresponding actions of the network analyzer are depicted in Fig. 2. The most important message is the CSP because it triggers the memory snapshot. As both communication partners send a CSP message, the snapshot is triggered when the monitored virtual machine sends it. The corresponding packet is not passed to the destination interface until a memory snapshot has been taken.

Memory acquisition

Every time the trigger mechanism detects a new TLS connection, the master secret that is present in VM memory needs to be recorded. This can either be achieved by directly searching the memory for the key, or by taking a snapshot of (parts of) the virtual machine and extract the key from the snapshot later when the communication needs to be decrypted.

In both cases, the snapshot process has to be executed fast in order to keep the delay impact on the connection as low as possible. Therefore, we decrease the time required to take the snapshot by minimizing its size. Thus, we have to find out where the master secret is stored in the main memory of a virtual machine. First of all we restrict the snapshot to the memory of the process that handles the TLS session. Therefore, we parse the kernel task structures of each process in order to find the process that handles the connection. To get this information we parse the file descriptor table of each process in the task structure and compare the source and destination IP/port combination. For this purpose, we wrote a custom utility extracting the required information from the guest Linux kernel as we found existing tools like Volatility or Rekall as too slow for that time-critical operation.

Additionally, we consider only those pages of a process that are mapped write able and anonymous. Anonymous pages do not have a reference to a file in their description structure. This usually holds for the heap and stack of a process as they are allocated dynamically.

Furthermore, we decrease the size of the snapshot by considering only pages that have been altered between the establishment of the connection and the key negotiation of the session key. Therefore, we register memory access handlers that monitor the process memory between the sending/receiving of the SH record and the CSP message. The occurrence of the SH marks the last point in time where the session key is not existing and the CSP message indicates that the key was computed. Thus, the key can be found in pages that have been modified or newly allocated during this period.

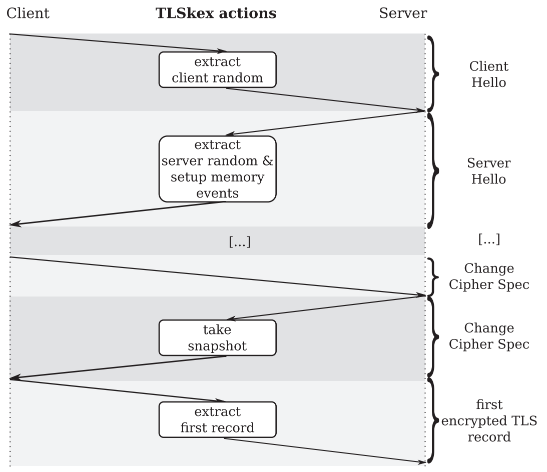

Fig. 2. TLS key negotiation process and the corresponding TLSkex actions.

This approach does not work, when the session key is stored in main memory before the connection is established, e.g., for resumed sessions. In this case, we can assume that we have captured the first key negotiation and do not need to extract it again. If this is not the case, only a snapshot of the whole address space of a process works.

Key extraction

TLSkex implements a brute force approach to find TLS master secrets in main memory. Therefore, it takes each 48 byte sequence in the snapshot as a master key and checks whether its derived *write_key* can successfully decrypt a TLS record of the connection. To test whether the decryption was successful we compute the HMAC of a decrypted TLS record and compare it with the given one. If they match, the key is correct. All necessary parameters, e.g. the server random, are extracted from the network flow by the proxy component.

As the key validation process is slow it is important that we do not try every byte sequence as a master secret. Therefore, we have implemented several strategies to precheck whether a byte sequence is a potential key. The first optimization is that we assume that a key is stored four byte aligned in memory. This increases the speed by a factor of four. The second optimization is that we look at the stochastic properties of a byte sequence. As the master secret is generated by a pseudo random function we assume that it contains about the same amount of zero and one bits and the amount is distributed binomial. For example, a 48 byte sequence that is randomly generated and binomial distributed has with a probability of about 90 percent between 176 and 208 one bits. The parameter μ is the expected count of one bits in the master secret. The TLS master secret is 48 bytes long and the probability p that a bit is one is 0.5. Thus, μ is $0.5 \times 48 \times 8 = 192$.

$$\sum_{\mu-k}^{\mu+k} \binom{n}{k} p^k * (1-p)^{n-k} > = 0.89$$

$$k = 16,\ \mu = 192,\ p = 0.5$$

Thus, we first test bytes sequences in the snapshot that have between 176 and 208 one bits. If we do not find a key with these properties we increase the borders and iterate again over the snapshot. Another heuristic to minimize the search space is to check whether a byte sequence consists only of ASCII characters. If so, the highest bit of each byte must be unset. The probability that such a 48 byte sequence with only ASCII characters is a key is about 0.5^{48}, which is negligibly small. The amount of required characters could also be reduced but the chance that a key gets preeliminated will increase. The last heuristic that is implemented in TLSkex is to check whether an eight byte sequence contains either only one bits or only zero bit. This can be tested easily as the bit counting function takes eight bytes as input and returns the amount of one bits. The probability that a key contains at least one sequence with 64 zero or one bits is also negligible ($2^*48^*0.5^{64} \approx 10^{-15}$). TLSkex combines all three heuristics in order to decrease the search space as much as possible without eliminating too many potential keys. If the key was not found, the heuristics can be switched off in order to research the key in the snapshot. After we have found the session key of a connection, we write it into a key log file that serves as in input for Wireshark to decrypt TLS streams.

Network logging & traffic decryption

TLSkex extracts only the master secret of TLS connections from main memory. It does not save the corresponding network traffic. In order to save the network traffic standard tools like tcpdump (Tcpdump contributors) can be used. Depending on the use case, the network sniffer should store only TLS connections in order to save space.

Moreover, TLSkex does not decrypt TLS sessions directly. It only extracts the TLS master secret from main memory. However, it can be used together with Wireshark to decrypt TLS encrypted network connections based on the extracted master key.

Evaluation

In this section we measure the time to take a snapshot of several programs and the time that is required to extract the key out of it. Additionally, we discuss the limitations and stealthiness of TLSkex. All measurements in this section are executed on a machine with an Intel(R) Core(TM) i5-2500 CPU @ 3.30 GHz and 4 GB of RAM. The hypervisor is Xen in version 4.4.1 out of the Debian stable repository. The dom0 and guest operating system is a 64 Bit Debian Linux with kernel version 3.16.0-4.

Memory acquisition

To measure the throughput of the memory acquisition process we read the whole address space of a virtual machine using LibVMI. Therefore, we requested each page sequentially by starting from the virtual address 0. For a guest system with 1024 MB of main memory the snapshot took 2.4 s, with 256 MB it took 0.6 s. These results not include the time for storing the data on a persistent storage. It only includes the address translation and the copy operation but not the write operation.

Table 1 depicts the total amount of pages of a sample set of processes, the amount of pages that are mapped anonymous and writable and the amount of pages that are allocated (new) and altered (modified) during the key negotiation process and dumped as a snapshot. The time t_{snap_start} describes the time that is required to set the memory events and t_{snap_stop} the time that is required to take the snapshot and find newly allocated pages. The time that is required to extract the session key out of a snapshot is denoted t_{search}. All values in this table are based on single run measurements in order to provide the dimension of the timing values. The Apache2 process acted as a server, the others ones as clients.

The first notable observation in Table 1 is that the differential snapshot is very small compared to the size of the anonymous and write able pages. For example, the differential snapshot of the Apache2 process is about one percent as big as the snapshot of the anonymous and writable mapped pages. This decreases the search space for the master secret dramatically.

Additionally, we can see that the time required to take a snapshot does not only correlate to its size. It also dependents on the size of the address space. For example, the snapshot of the s_client process has about the same size as the other ones. However, it is six to ten times faster as the other ones because its address space is smaller. This is caused by the fact, that we have to iterate over the address space in order to set and remove the memory events.

Key extraction

In Table 1 we present the time that is required to extract the TLS master secret from the memory dump of different processes. The different times to find the key by having the same snapshot size are caused by the position of the cryptographic key and the entropy of a snapshot. For example, if the key was stored in the heap of the process, it is in the beginning of the process dump and is found faster. However, when the key is on the stack, it is in the rear part of the memory dump and it takes longer to find it. Additionally, the time depends on the amount of bytes that are filtered out by our heuristics.

Our brute-force implementation was able to test about 131 thousand keys per second. This means that every sequential 48 byte sequence of a memory snapshot with a size of 131 KB can be tested in 1 s. The low throughput is mainly caused by the key derivation of TLS and the decryption process. We assume that we can improve the performance of the key testing algorithm in the future. However, this can be executed offline and does not affect the monitoring process.

In Table 2 we show how the performance of different heuristics can decrease the size of the search space of a memory snapshot by selecting only byte sequences that have the characteristics of a random encryption key. The first row shows the computed probability that a key matches the heuristic.

The first six columns show how many four-byte-aligned 48 byte sequences have between $192 - k$ and $192 + k$ one bits. For $k = 16$ only about two to four percent of the memory meet this condition. However, there is a chance of about 90% that a random master key satisfies this condition. In other words, we find the master key with a probability of about 90% if we just search the small part of the memory that matches this condition. The bit counting heuristic is implemented very efficiently. Thus, the pretesting reduces the time for finding the key dramatically.

In Table 2 we depict as well the performance of two other heuristics. The first one is testing whether a byte sequence is not an ASCII string. This is accomplished by testing if the highest bit of each byte is set. Depending on the application, this simple heuristic reduced the time for

Table 1
Amount of mapped and changed memory pages (4096 bytes) of different processes during the key negotiation procedure and the time to prepare (t_{snap_start}) and take (t_{snap_stop}) a differential snapshot; t_{search} denotes the time to extract a key from a snapshot.

Process	Total	Anon & writeable	New	Modified	Dumped	t_{snap_start}	t_{snap_stop}	t_{search}
Apache2	72,090	3715	0	26	26	4.3 ms	4.4 ms	30 ms
Curl	38,264	3438	15	19	34	3.3 ms	4.0 ms	2 ms
Wget	22,813	1378	16	13	29	4.0 ms	3.5 ms	2 ms
s_client	6114	152	9	22	31	0.4 ms	0.6 ms	8 ms

Table 2
First row: probability that a key is not eliminated by the heuristic. Other rows: percentage of a memory snapshot that contains a 48 byte long and four byte aligned sequence with: a) 192 ± *k* one bits, b) the byte sequence is not an ASCII string c) no 8 byte sequence with only zero or only one bits d) a to c combined.

Process	a						b	c	d
	k = 1	k = 2	k = 4	k = 8	k = 16	k = 32	No string	Not all 0/1	Combined (k = 16)
key included	8.12	16.2	31.6	58.5	89.7	99.9	$1-10^{-15}$	$1-10^{-19}$	87.7
Apache2	0.10	0.28	0.64	1.27	2.33	4.26	85.49	43.54	1.69
Curl	0.15	0.45	1.04	2.11	3.50	4.75	77.53	10.55	3.32
Wget	0.15	0.46	1.06	2.15	3.60	4.91	78.10	10.68	3.38
s_client	0.054	0.18	0.49	0.96	1.89	3.40	56.52	37.35	1.63

the key search in our experiments by between 56% and 85% s. Another approach is to test whether each of the eight byte blocks where we count the bits in has either 0 or 64 one bits. This reduced the search space by between 10% and 44%, again depending on the application process. The last column shows how much of the memory snapshot contains potential keys when all presented heuristics are combined.

Network proxy

The performance of the network is mainly influenced by two factors: (1) The overhead of each packet which is caused by the deep packet inspection proxy that analyzes the contents of the TCP headers in order to check if it belongs to a TLS connection. We currently ignore this overhead in our proof-of-concept implementation as it is a constant factor that affects all packets. The optimization of this proxy is part of our future work.

(2) The overhead that is caused by analyzing the TLS records and the corresponding actions, e.g., the creation of the snapshot depends on the TLS records which are included in a TLS record. For example packets with only application data records are simply forwarded. The only packets that are delayed noticeable are packets with a SH and outgoing CSP records. Packets with a SH record are delayed by t_{snap_start} and CSP messages by t_{snap_stop}. Both values depend on the size of the address space of a process and the amount of changed pages.

Limitations

We have made several assumptions to increase the performance. For example we only take a snapshot of the process that handles the connection. However, malware might spawn a dedicated crypto process that runs the encryption routine. In that case, our approach would not work. Thus, it might be better to save a snapshot of the whole virtual machine. A malware might also obfuscate the key in memory, e.g., by shifting the byte order in order to hide it from TLSkex. But a human operator might notice this problem when the key was not extracted and can implement a custom strategy for the specific use case.

Another way to circumvent the automatic key extraction process of TLSkex is to use a slightly changed version of the TLS protocol, e.g., by modifying the default numbers of some commands. However, this requires that both – the client and the server – use the same modified protocol version.

TLSkex trusts the kernel structures of the guest to be uncorrupted and reliable and uses them for example to extract memory mappings of a process. Thus, a guest system might foil VMI based analysis by placing crafted data structures in memory (Bahram et al., 2010). This is a general problem of VMI based analysis and is out of the scope of this paper.

Furthermore, memory areas can be swapped out to disk. If the master secret is in a page that is swapped out, our current implementation would not be able find it. The time interval between generation of the master secret and the detection of the CSP message (and thus the creation of the snapshot) is very small, and thus it is highly unlikely that the memory page will be swapped out. We have not observed this problem during our experiments.

Finally, TLSkex can serve as a DoS vector, for example, when many TLS connections are spawned and many snaphshots must be taken. Thus, it is important to find ways to minimize the overhead of TLSkex in the future. Additionally, we have to investigate how this problem can be circumvented in practice.

Stealthiness

One of the use cases of TLSkex is to analyze the network traffic of malware. Thus, we have to discuss whether malware can detect that it is being analyzed and thus decides to behave differently (Balzarotti et al., 2010).

The first observation malware can make is that it runs inside of a virtual machine (Raffetseder et al., 2007). However, this does not state anything about whether TLSkex is monitoring the process.

The next thing a monitored process might detect is that some network packets have a higher latency than others. Especially packets with a CSP and a SH message are delayed until the corresponding action (setup memory events and take snapshot) has been executed. Depending on the size of the address space of a process this might be noticeable to the communicating process. We assume that the time required to take the snapshot can be decreased by future implementations which will make the delay not detectable anymore in common network scenarios.

Related work

Approaches for decrypting TLS traffic

Butler (2013) describes advantages of decrypting encrypted SSL network connections in company networks. He recommends doing that especially for intrusion detection, forensics and data loss prevention. Other authors have as well described this problem and proposed several solutions. In general we can divide these solutions into *active* and *passive* approaches. Passive approaches do not intercept the communication. They try to decrypt the network traffic by having knowledge about the key. Active approaches manipulate the network traffic, e.g., by installing man-in-the-middle proxies. In Taubmann et al. (2015) we already presented a use case where the decryption of TLS connections can be used to detect attacks and to perform malware analysis. Therefore, we described an attack that was executed via https and TLS encrypted IRC botnet communication.

Active approaches

The easiest way to decrypt network traffic is by forcing the client not to encrypt. This approach is taken by sslstrip (Marlinspike, 2009). It acts as a http to https proxy. Therefore, it replaces the https designator in the content for the client with http. Thus, the proxy can simply read the unencrypted data and forward it encrypted. To make it more stealthy to users, it also sets a fake encryption favicon (the icon next to the URL in address bar).

The tool sslsniff (Marlinspike, 2002) acts a man in the middle proxy for TLS based connections. In this case the client does not connect directly to the server. Instead, he connects to sslsniff and sslsniff connects to the server. Hence, the client does not see the certificate of the server but the certificate of sslsniff. This is the reason why the client is able to detect when someone intercepts the connection who does not have a valid certificate for the server. Thus, this approach is only feasible when the client program does not verify the server certificate. In order to solve this problem SSLsplit (Heckel, 2013) can be used. It generates on the fly certificates for each target. When an attacker is able to install the fake certificate authority (CA) certificate (e.g., in the browser) all SSL connections seem to be trustworthy to the user. This can be easily established in company networks where an administrator can easily install certificates on every client.

All active approaches work in most of the cases very well. This is caused by users that do not have the knowledge to detect intercepted connections or by programs that have flaws in the certificate verification process. However, all of them can be detected – at least in theory. For example it is possible to detect when a connection is not encrypted at all or if the certificate was signed by an untrusted CA or not the expected one. Active approaches are often inappropriate for malware analysis as the system under analysis might behave differently when it gets monitored (Balzarotti et al., 2010). Moreover, active approaches potentially lower the security of the monitored connections. Thus, they are not applicable in real world scenarios where keeping the security level is important.

Passive approaches

They only monitor encrypted connection and do not modify it. They try to decrypt it by getting the key from other sources. For example, the tool ssldump (Iveson, 2014) is able to decrypt RSA-based TLS connections on-the-fly when the private key of the server is available. This can be achieved by copying it from the hard disk or by extracting it from main memory. However, this approach is not feasible when a client connects to a server (e.g, a command & control server) where its private key is not available or when symmetric session keys are negotiated with PFS key-agreement protocols like DH or ECDH. Additionally, this approach works only with TLS session tickets (Salowey et al., 2008) when the key exchange of the session ticket was captured.

The tool Wireshark (Wireshark contributors, 2015) is also able to decrypt TLS connections if the TLS session keys are made available to it. Some webservers like Apache or browsers such as Firefox and Chrome support to log session keys into a file which can be interpreted by Wireshark.

Another approach to extract the master secret of a TLS session is to monitor the function calls of a process. This can be achieved by setting a breakpoint on the key generation function (e.g., the PRNG function) of the TLS implementation in use and to extract the results of the function. This approach is feasible only when a known library is used in order to set the breakpoint to the right address and to parse the parameters correctly (Bremer, 2015). However, if malware is compiled statically and an unknown TLS implementation is used or the function symbols are removed it is not easy to dynamically find the point in code where the master secret is stored and accessed.

Key extraction

The extraction of cryptographic keys from main memory is an important task in computer forensics in order to decrypt encrypted information (Maartmann-Moe et al., 2009). If the position of a key in a memory dump is not known, heuristics help to identify potential keys. Shamir et al. (Shamir and Someren, 1999) described theoretical approaches to find RSA cryptographic keys efficiently in gigabytes of data by using stochastic information. Klein (2006) used a different approach. He is searching for

Table 3
Comparison of different approaches to decrypt TLS communication.

Solution	Approach	Stealthiness	PFS	Key renegotiation	Server & client	Session resumption
TLSkex	VMI based	✓	✓	✓	✓	✓
ssldump	Requires private RSA key	✓	✗	(✓)	(✓)	(✓)
sslstrip	Removes encryption partially	✗	N/A	✗	✗	N/A
sslsniff	SSL proxy with fake certificate	✗	✓	✓	✓	✓

sequences of the ASN.1 encoding which is commonly used to store the keys. Unfortunately, this approach is only feasible for finding RSA keys. It does not work for finding DH or ECDH keys as there is no standardized way to store them.

An approach to find symmetric AES and DES keys in main memory was described by Halderman et al. (2008). They do not search for an implementation specific structure but they are searching for byte sequences with a potential key and its corresponding key schedule. As they are using this approach on corrupted main memory snapshots gained with cold boot attacks, they also implemented a heuristic that is able to identify a key schedule with flipped bits.

To our best knowledge, there has been no work that extracts the master secret of a specific TLS connection in main memory. By using the master secret, the key extraction routine of TLSkex is independent of the utilized cipher suite and cryptographic parameters.

Virtual machine introspection

In order to extract the key of a TLS encrypted connection from a virtual machine we need to access and interpret the memory of it. Therefore, we have to bridge the *semantic gap problem*: reconstructing high level state information from low level data-sources (Garfinkel and Rosenblum, 2003). This problem can be divided into the weak and the strong semantic gap problem (Jain et al., 2014). There has been much research in the past years and the weak semantic gap problem can be considered “a solved engineering problem” (Jain et al., 2014; Dolan-Gavitt et al., 2011).

Additionally, we can divide between active and passive VMI methods. For example parsing the process structure of an operating system is a passive method (Schuster, 2006). In contrast, the continuous tracing of system calls is an active method (Pfoh et al., 2011).

Volatility (Volatility Foundation, 2015) and Rekall (2015) are the most known frameworks for doing VMI and interpreting the memory of operating systems such as Windows, Linux and MacOS X. LibVMI (2015) is a library that abstracts the interface for accessing the memory of virtual machines of different hypervisors like XEN (Xen, 2015) and KVM (2015). It also supports events (e.g., memory access or interrupts) in order to dynamically trace memory access or the execution of instructions at runtime. However, it does not come with functions that allow taking differential snapshots of a process or to parse the kernel task structure of Linux processes in order to extract the memory mappings and open network connections.

Conclusion

TLSkex is a VMI based solution that extracts the master secret of TLS connections of virtual machines. The master secret can be used to derive the symmetric keys of TLS session in order to decrypt them. Thereby, TLSkex is independent of the cryptographic algorithms or the key exchange algorithm. Thus, this approach even works when a PFS key agreement algorithm such as DH or ECDH is used. As this approach is so general, it can be extended to other cryptographic protocols such as the key exchange mechanism of SSH (Ylonen and Lonvick, 2006).

We have also shown, how differential snapshots and heuristics can be used to enhance the process of extracting cryptographic keys from main memory.

In contrast to other solutions, TLSkex does not modify or weaken the security of TLS connections (see Table 3). Thus, TLSkex can be considered as a stealthy approach for finding session keys and decrypting TLS connections. Only the delay of some network packets – those that contain a CSP TLS record – may be noticeable to monitored applications. This can be improved by better key extraction heuristics and a faster implementation. Moreover, TLSkex can also be used for TLS sessions that use a session state ticket without capturing the key negotiation process of previous sessions. Finally, TLSkex can be used to extract the key material either from a server or a client application.

TLSKex is above all a tool that allows its user to extract/access the information from a communication secured by the TLS protocol. The described traffic decryption method has the only objective to provide a mean that eases the investigation of eventual security incidents within a target virtualized environment. However, we cannot guaranty that all TLSKex users take profit from using the tool only for ethical purposes, since acceding encrypted information may also hide a malicious attempt to break the confidentiality of data. In all cases we believe that, in order to comply to privacy rights, all cloud customers should always acknowledge that their communications might be the object of analyze and decryption for security purposes and further investigations. But it remains the sole responsibility of the cloud provider to prove or justify the legal relevance and legitimacy of such use and provide official documentation of used methods.

We are planning to release a revised version of TLSkex under an open source license on our website.[1]

[1] http://www.fim.uni-passau.de/sis/.

Acknowledgments

The research leading to these results was supported by the “Bavarian State Ministry of Education, Science and the Arts” as part of the FORSEC research association.

References

Bahram S, Jiang X, Wang Z, Grace M, Li J, Srinivasan D, et al. DKSM: subverting virtual machine introspection for fun and profit. In: Reliable distributed systems, 2010 29th IEEE symposium; 2010. p. 82–91. http://dx.doi.org/10.1109/SRDS.2010.39.

Balzarotti D, Cova M, Karlberger C, Kirda E, Kruegel C, Vigna G. Efficient detection of split personalities in malware. In: Proc. of the network and distributed system security symposium, NDSS; 2010.

Bremer Jurriaan. Transparent MITM with cuckoo sandbox. 2015. http://jbremer.org/mitm/ [accessed 02.10.15].

Butler M. Finding hidden threats by decrypting SSL, A SANS analyst whitepaper. SANS Institute; 2013. http://www.sans.org/reading-room/whitepapers/analyst/finding-hidden-threats-decrypting-ssl-34840 [accessed 19.08.15].

Dolan-Gavitt B, Leek T, Zhivich M, Giffin J, Lee W. Virtuoso: narrowing the semantic gap in virtual machine introspection. In: Proceedings of the 2011 IEEE symposium on security and privacy, SP'11. Washington, DC, USA: IEEE Computer Society; 2011. p. 297–312.

Garfinkel T, Rosenblum M. A virtual machine introspection based architecture for intrusion detection. In: Proc. network and distributed systems security symposium; 2003. p. 191–206.

Heckel PC. Use SSLsplit to transparently sniff TLS/SSL connections – Including non-HTTP(S) protocols. 2013. http://blog.philippheckel.com/2013/08/04/use-sslsplit-to-transparently-sniff-tls-ssl-connections/ [accessed 19.08.15].

Halderman J, Schoen SD, Heninger N, Clarkson W, Paul W, Calandrino JA, et al. Lest we remember: cold boot attacks on encryption keys. In: Proc. of the 17th USENIX Security Symposium; 2008.

Homan J. How to decrypt OpenSSL sessions using wireshark and SSL session identifiers. 2013. http://www.cloudshield.com/blog/advanced-malware/how-to-decrypt-openssl-sessions-using-wireshark-and-ssl-session-identifiers/ [accessed 02.10.15].

Iveson S. Using ssldump to decode/decrypt SSL/TLS packets. 2014. http://packetpushers.net/using-ssldump-decode-ssltls-packets/ [accessed 19.08.15].

Jain B, Baig M, Zhang D, Porter D, Sion R. Sok: introspections on trust and the semantic gap. In: Security and privacy (SP), 2014 IEEE symposium; 2014. p. 605–20.

Klein T. All your private keys are belong to us – Extracting RSA private keys and certificates from process memory. 2006. http://www.trapkit.de/research/sslkeyfinder/keyfinder_v1.0_20060205.pdf [accessed 19.08.15].

Krawczyk H, Bellare M, Canetti R. HMAC: Keyed-hashing for message authentication. IETF, RFC 2104, 1997. https://tools.ietf.org/html/rfc2104.

KVM, http://www.linux-kvm.org/ (July 30 2015).

LibVMI, http://libvmi.com/ (July 30 2015).

Maartmann-Moe C, Thorkildsen SE, Årnes A. The persistence of memory: forensic identification and extraction of cryptographic keys. Digit Investig 2009;6:S132–40.

M. Marlinspike, sslsniff, http://www.thoughtcrime.org/software/sslsniff/, [accessed 19.08.15] (2002).

M. Marlinspike, sslstrip, http://www.thoughtcrime.org/software/sslstrip/, [accessed 19.08.15] (2009).

Pfoh J, Schneider C, Eckert C. Nitro: hardware-based system call tracing for virtual machines. In: Advances in information and computer security, Vol. 7038 of lecture notes in computer science. Springer; 2011. p. 96–112.

Pric JW. Significant SSL performance loss leaves much room for improvement. NSS Labs Report. 2013. https://www.nsslabs.com/reports/ssl-performance-problems [accessed 11.08.15].

Raffetseder T, Kruegel C, Kirda E. Detecting system emulators. In: Information security, Vol. 4779 of LNCS. Springer; 2007. p. 1–18.

Rekall, Memory forensics analysis framework, http://www.rekall-forensic.com (January 15 2015).

Salowey J, Zhou H, Eronen P, Tschofenig H. Transport Layer Security (TLS) Session Resumption Without Server-Side State. IETF, RFC 5077, 2008. https://tools.ietf.org/html/rfc5077.

Schuster A. Searching for processes and threads in microsoft windows memory dumps. Digit Investig 2006;3:10–6. URL, http://dx.doi.org/10.1016/j.diin.2006.06.010.

Shamir A, Someren NV. Playing “hide and seek” with stored keys. In: Proc. of the 3rd Int. conf. on financial cryptography, FC'99. London, UK, UK: Springer-Verlag; 1999. p. 118–24. URLhttp://dl.acm.org/citation.cfm?id=647503.728464.

Taubmann B, Dusold D, Frädrich C, Reiser HP. Analysing malware attacks in the cloud: a use case for the tlsinspector toolkit. In: 2nd Workshop on security in highly connected IT systems (SHCIS); 2015.

Tcpdump contributors, TCPDUMP/LIBPCAP public repository, http://www.tcpdump.org/, [accessed 19.08.15].

Volatility Foundation, Volatility command reference, https://github.com/volatilityfoundation/volatility/wiki/Command-Reference [accessed 19.08.15].

Wireshark contributors. Wireshark wiki about secure socket layer (SSL). 2015. https://wiki.wireshark.org/SSL [accessed 19.08.15].

Xen, http://www.xenproject.org (July 30 2015).

Guanglin Xu, Peter Klemperer, James Hoe. Improving LibVMI introspection performance with shared memory snapshots. 2013. http://www.andrew.cmu.edu/user/guanglin/PDL-presentation.pdf.

Ylonen T, Lonvick C. The secure shell (SSH) transport layer protocol. IETF, RFC 4253, 2006.