EXTRACTING CNG TLS/SSL ARTIFACTS FROM LSASS MEMORY

Jake Kambic

2016

Defcon

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

LIST OF ACRONYMS & ABBREVIATIONS

AES	Advanced Encryption Standard
API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
CNG	CryptoAPI Next Generation
DER	Distinguished Encoding Rules
DLL	Dynamic Link Library
DPAPI	Data Protection API
DTB	Directory Table Base
DWORD	Double Word
IP	Internet Protocol
LSASS	Local Security Authority Sub-System
OS	Operating System
PDB	Program Database
PEB	Process Execution Block
PEM	Privacy-enhanced Electronic Mail
PFS	Perfect Forward Secrecy
PKCS	Public Key Cryptography Standards
PKI	Public Key Infrastructure
PMK	Pre-Master Key
PMS	Pre-Master Secret
PoC	Proof-of-Concept
PRF	Pseudo-Random Function

PRNG	Pseudo-Random Number Generator
PSK	Pre-Shared Key
QWORD	Quad Word
RDP	Remote Desktop Protocol
RFC	Request for Comment
SNI	Server Name Indicator
SSL	Secure Socket Layer
SSP	Security Support Provider
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TLV	Type-Length-Value
UDP	User Datagram Protocol
VAD	Virtual Address Descriptor

LIST OF TERMS

Bit	“A contraction of the term ‘binary digit’; the smallest unit of information storage, which has two possible states or values. The values usually are represented by the symbols ‘0’ (zero) and ‘1’ (one).” (Shirey, 2007, p. 36).
Bitstream	A contiguous linear serialization of bits.
Byte	“A fundamental unit of computer storage; the smallest addressable unit in a computer’s architecture. Usually holds one character of information and, today, usually means eight bits. (Compare: octet.)” (Shirey, 2007, p. 43).
Cipher	“A cryptographic algorithm for encryption and decryption” (Shirey, 2007, p. 61).
Ciphertext	“Data that has been transformed by encryption so that its semantic information content (i.e., its meaning) is no longer intelligible or directly available.” (Shirey, 2007, p. 62).

Common Criteria	An international “standard for evaluating information technology (IT) products and systems. It states requirements for security functions and for assurance measures. ” (Shirey, 2007, p. 69).
Double Word	“A DWORD is a 32-bit unsigned integer (range: 0 through 4294967295 decimal). Because a DWORD is unsigned, its first bit (Most Significant Bit (MSB)) is not reserved for signing.” (Microsoft, n.d.-c, s. 2.2.9).
Encode	“Use a system of symbols to represent information, which might originally have some other representation. Example: Morse code.” (Shirey, 2007, p. 119).
Encryption	“Cryptographic transformation of data (called ”plain text”) into a different form (called ”cipher text”) that conceals the data’s original meaning and prevents the original form from being used. The corresponding reverse process is ”decryption”, a transformation that restores encrypted data to its original form.” (Shirey, 2007, p. 119).

Ephemeral	"Refers to a cryptographic key or other cryptographic parameter or data object that is short-lived, temporary, or used one time." (Shirey, 2007, p. 122).
Hash Value	"The output of a hash function." (Shirey, 2007, p. 141).
Hash Function	"A (mathematical) function which maps values from a large (possibly very large) domain into a smaller range. A 'good' hash function is such that the results of applying the function to a (large) set of values in the domain will be evenly distributed (and apparently at random) over the range.[1]" (Shirey, 2007, p. 140).
Plaintext	"Data that is input to an encryption process." (Shirey, 2007, p. 225).
Pseudorandom	"A sequence of values that appears to be random (i.e., unpredictable) but is actually generated by a deterministic algorithm." (Shirey, 2007, p. 240).

[1]In the context of security, hash functions accept variable-length input and produce fixed-length output

Pseudorandom Function	A function that “uses a deterministic computational process (usually implemented by software) that has one or more inputs called ‘seeds’, and it outputs a sequence of values that appears to be random according to specified statistical tests.” (Shirey, 2007, p. 244).
Quad Word	“A QWORD is a 64-bit unsigned integer.” (Microsoft, n.d.-c, s. 2.2.40).
Request For Comment	“One of the documents in the archival series that is the official channel for IDOCs and other publications of the Internet Engineering Steering Group, the Internet Architecture Board, and the Internet community in general.” (Shirey, 2007, p. 250).

CHAPTER 1. INTRODUCTION

The forensic analysis of digital media has rapidly advanced over the course of the last decade, vaulting from the fringes of forensic science to prominence and acceptance within both the forensic community and the public eye. This explosion has been in part fueled by the rapid expansion of technology into nearly every facet of daily life. These digital systems, from traditional personal computers, to smart phones, vehicles, and intelligent appliances, are increasingly storing, processing, and transmitting data that is providing critical context in cases and investigations around the world.

More recently emerging as a sub-discipline of digital forensics is volatile memory forensics, focused on the incredible wealth of information that can be gleaned from capturing this ephemeral storage media. As the size of persistence storage both increases and becomes ever more economical, the smaller volatile memory that holds the salient points of recent activity is being turned to as a great triage mechanism. It stands on its own merits, however, providing exclusive access to items of evidentiary value that are simply unobtainable from any other source.

Driving the expansion of interest in volatile memory forensics has been the enumeration of artifacts belonging to the prolific Windows family of operating systems, and the development of tools that have abstracted the complexities of understanding Windows memory management, making the volatile medium far more accessible. It is upon this driver for development that the author builds, seeking to explore the extraction of Windows SSL/TLS secrets from within volatile memory.

1.1 Statement of Problem

Currently, there exists no reliable or automated way to forensically decrypt SSL/TLS connections that leverage ephemeral key negotiations as implemented by the modern Windows operating system. Since Secure Socket Layer was first put forth by Netscape and later galvanized in an RFC, the concept of Public Key Infrastructure (PKI) and the key exchange has been central to its security. The private key has historically been the penultimate secret, stored persistently with the server and used across all key exchanges to decrypt the pre-master secret. This has proved to be its Achilles' heel from a security standpoint, as the private key can be leveraged to retroactively decrypt the handshake of any session between the server and clients for which the key was used indefinitely. It has also been suggested that the key itself may often outlive a certificate renewal, being used across regenerated certificates (Taubert, 2014).

The advent of Perfect Forward Secrecy (PFS), a property that means past connections are secure from future decryption (accomplished in SSL and Transport Layer Security (TLS) through use of “throw-away” ephemeral keys), has proved a double-edged sword for network defenders, law enforcement activity, and attackers alike. On the one hand, it helps ensure the confidentiality of mission critical information against future theft; however, it similarly imposes a barrier to access when used nefariously to exfiltrate information from a network or shroud illicit activities.

Still, the implementations of PFS for TLS in practice have been mired by conflicting objectives. The desire for quick reconnection through the use of “Session Resumption” has meant caching of SSL/TLS secrets in main memory, or even to disk. This has been well-documented in instances of Web servers, like Apache or NGINX (Dreijer & Rijs, 2013) on Linux and Unix systems, but has not seen the same level of scrutiny in Windows, possibly owing to the closed-source nature of the material.

Beyond simply decryption of connections, the attribution or trace evidence of connections, while perhaps less interesting to an attacker, would still provide potentially invaluable context to an Incident Responder investigating a suspect system, or a forensic analyst working a case. Artifacts like public certificates or negotiated connection parameters could furnish clues about attack vectors or the intent of a given connection, but currently are not well documented artifacts on Windows-based systems.

1.2 Significance

As ephemeral cipher suites become increasingly popular and are embraced by the Windows operating system, it will become more important to identify a reliable and forensically sound mechanism to decrypt them. Windows components like Edge (and Internet Explorer), RDP, Outlook, Skype, Windows Update, SQL Server, IIS, Microsoft Exchange, and LDAPS (Active Directory mechanism) all leverage the Windows SSL/TLS implementation and will likely embrace PFS as a desired or default property in the future (Microsoft, 2015d). Anecdotal testing on Windows 10, suggests that it already does prefer ephemeral cipher suites. Other third-party applications (like the popular Citrix client) that leverage Windows libraries for SSL/TLS negotiation will also likely follow the system preference. This gives rise to several use cases for a legitimate capability to retroactively decrypt connections that were not otherwise captured through a pre-arranged mechanism (such as SSL Inspection), for example:

- Decryption of logged malicious RDP sessions after a breach for incident response purposes
- Decryption of illicit HTTPS traffic captured prior to the serving of a search warrant
- Automated, transparent decryption of malware connections (especially salient for instances where Powershell or other native elements are used).

- Decryption of Windows Update traffic

1.3 Research Question

Given the problem of decrypting connections made with the Windows implementation of ephemeral key exchanges, the research question stands thusly:

> Do the requisite connection parameters exist in the memory of modern Windows systems to retroactively decrypt sessions? If so, how long do these artifacts persist?

Complimenting this question is the ancillary pursuit of any other connection artifacts that may help provide context to the nature of the connection.

1.4 Hypothesis

It is posited that, to maintain the connection, at the very least the session keys must exist for the duration of the connection. It is further supposed that, outliving the connection itself, other secrets may exist to support the ability to perform session resumption or other implementation specific functionality.

1.5 Assumptions

The research question and subsequent hypothesis are predicated on several assumptions, which are as follows:

- The term modern with respect to operating systems refers to those currently actively supported by Microsoft at the time of writing
- It is assumed that the operating systems to be examined are standard in so much as the kernel and physical memory addressing schemes have not been altered by any party other than Microsoft and are in general distribution

- It is assumed that any operating system being examined may be either installed on physical or virtual hardware without consequence; that is to say that the use of virtualization will have no impact on the subject matter of the study
- It is assumed that any underlying analysis platform chosen is sound in its implementation and can be relied upon to perform its purported function

1.6 Limitations

In addition to the assumptions, it was important to define the scope of the research. The methodology appearing in Chapter Three of this thesis will discuss the instrumentation that inherently produces constraints, but the scope will be discussed here explicitly in terms of limitations and delimitations. Limitations refer specifically to those things that the research defines as within bounds of the scope. The delimitations, conversely, will seek to delineate those things which are reasonably outside the bounds of the research and were not explored at any depth.

The limitations for this study included:

- Windows 10 operating systems were considered on Intel x86 and x64 architectures
- All versions of SSL and TLS supported by Windows 10 were considered
- Either the Volatility or Rekall frameworks were used to demonstrate the findings of the thesis, or both as necessary

1.7 Delimitations

The delimitations for this study included:

- Linux, Unix, or other systems which have implementations or emulation of Microsoft cryptographic packages were not examined, unless explicitly identified in the limitations

- Third party cryptographic protocols or packages for the operating systems being considered were not examined
- This thesis did not examine methods of volatile memory acquisition outside of a cursory glance during the literature review, and the necessary semantic elements that may define the proprietary bitstream structures of different acquisition methods
- Paging constraints and operations were not considered as part of this work, though they may be in future work
- CPU cache size and levels were not controlled for due to the difficulty of manipulation in virtual environment that was used during testing

CHAPTER 2. RELEVANT LITERATURE

This section will highlight literature considered salient to the problem identified in Chapter One and the development of a solution. While this chapter will provide background information, it is assumed that the reader has an understanding of Transmission Control Protocol (TCP)/Internet Protocol (IP) networking, computer architectures, and digital forensics. Part of this literature review will not involve academic works, but also material from industry conferences, code from well-established open-source projects, and even blog posts from prominent members within the community. The review will progress through four larger stages:

- A brief discussion of modern memory analysis frameworks
- A cursory overview of SSL/TLS to provide framing for methods and results
- A review of Windows internal memory management and TLS implementation, the subject of the thesis
- A review of prior works as grounds for due diligence, in pursuit of showing good faith regarding the novelty and significance of the research

The most salient insights from the literature review across all of these sub-divisions are provided here for convenience:

- Volatility and Rekall are the memory analysis platforms that will be considered
- SSL/TLS require caching of secrets to support session resumption, even in cases where PFS is attempted
- LSASS is the Key Isolation component of the Microsoft's SSL/TLS implementation
- There is prior work on extracting persistent RSA private keys from Windows systems

- There is prior work on extracting OpenSSL secrets from UNIX-like systems
- There does not appear to be prior work on extracting ephemeral SSL/TLS secrets from the Windows SSL implementation

2.1 Volatile Memory Analysis Frameworks

Although many methods exist for memory collection, there are currently two primary frameworks that exist for analysis: the Volatility and Rekall Memory Forensics Frameworks. Both are considered here, as either one or both will be used to demonstrate findings of the thesis.

Perhaps the gold standard for volatile memory analysis has been the Volatility Framework originally developed by Alex Walters and Nick Petroni in the form of "FATKit" and "Volatools" (Petroni, Walters, Fraser, & Arbaugh, 2006; Walters & Petroni, 2007). Core developers of Volatility, led by Michael Hale Ligh, went on to develop the most extensive source material to date on memory analysis for forensics, entitled "The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac memory."

Volatility is extensible via plug-ins and has the capability to interpret the architecture and OS dependent address spaces, making it an ideal platform on which to develop a Proof-of-Concept (PoC) solution. It makes use of a structure referred to as a "V-Type" that allows objects to be created and extended based on a C-like strucure, without knowing all fields in the structure (Ligh et al., 2014). This is a powerful abstraction for reverse engineering and forensic artifact extraction.

Rekall originated as a branch of Volatility maintained by Micheal Cohen (also known by the alias "Scudette") and was called the "scudette" or "Technology Preview" branch. It seems that differences in philosophy and changes to the main code base meant that the scudette branch was never destined to be merged with the master branch of Volatility (Cohen, 2015a). The differences in approach are outlined principally in the Rekall blog and on the Rekall website (Cohen, 2015b). Succinctly, they are the way in which profiles for various operating systems are handled, the way operating systems are identified, and the preference in Rekall for the interactive command-line, as well as redesign of the plug-in structure. Rekall also (by extension of the way it manages profiles) has incorporated a dynamic symbol loading and parsing system that it can leverage automatically (when allowed Internet access).

Both Volatility and Rekall support much of the same functionality through plug-ins. Plug-ins implement an interface, and can be developed and incorporated by anyone, making both tools modular, hence they are described as a framework. The logical view of this architecture is depicted in Figure 2.1 The philosophy taken by Rekall, particularly the dynamic ability to look up symbols and localization of profiles, even within Windows versions, seems to be a desirable trait going forward. At the same time, many environments leveraged by the target audience require analysis systems that are completely disconnected from the Internet, meaning that any methods developed cannot solely rely on the capability to dynamically fetch symbols, if at all possible.

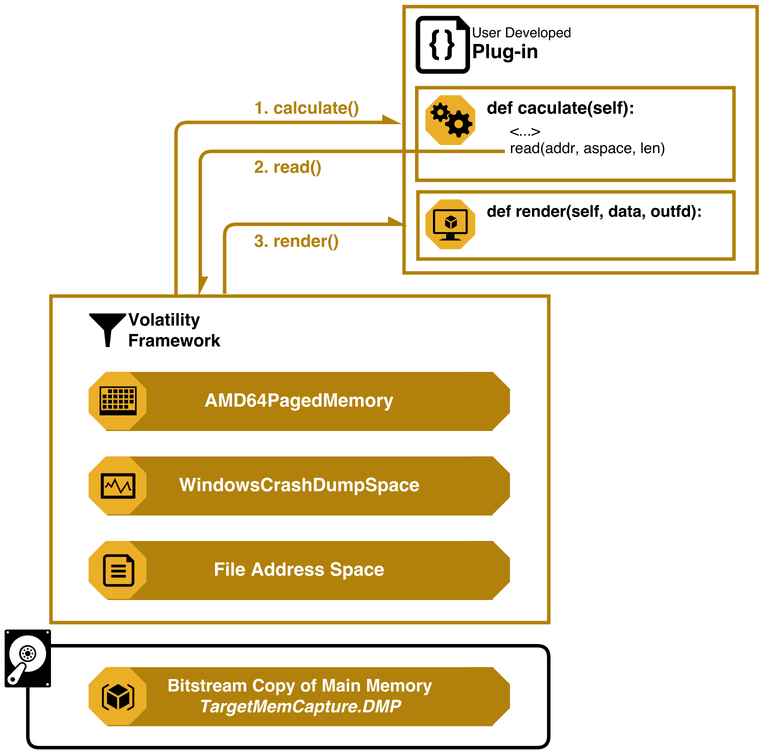


Figure 2.1. Volatility plug-in interface and address space abstraction (Ligh et al., 2014)

2.2 SSL & TLS

Understanding SSL/TLS internals is central to the thesis. The core purpose of SSL, and its successor TLS, is to provide a mechanism to securely establish communications between two parties by defining how both parties will decide upon and exchange the necessary information to authenticate each other and encrypt their communications. This section will overview the poignant components, namely: the handshake, key exchange, session resumption, session ticket extension, and the extended master secret extension. These topics will be reviewed per the Request for Comment (RFC) specifications here, and then particulars of the Windows implementation will appear in a later section.

A distinction that should be clarified is that, while it seems generally accepted that SSLv3 is equivalent to TLSv1, the two should not be conflated and are not interoperable. This information is relayed explicitly in the TLSv1 RFC:

> The differences between this protocol and SSL 3.0 are not dramatic, but they are significant enough that TLS 1.0 and SSL 3.0 do not interoperate (although TLS 1.0 does incorporate a mechanism by which a TLS implementation can downgrade to SSL 3.0).(Dierks & Allen, 1999, p.5)

The two are decidedly related though, closely enough in fact that the protocol identifier for TLSv1 is 0x0301, a reference to "SSLv3.1." Still, a few points worth noting (Freier et al., 2011; Goh & Boneh, 2001) are:

- TLS Extensions are not strictly supported by SSLv3
- The HMAC function differs between the two
- The Pseudo-Random Functions (PRFs) differ
- Total allowed padding lengths differ
- The last message of the handshake differs

These differences are touched upon because they could alter the artifacts seen in memory between the two (though in part this is implementation specific). There remains enough similarities that the overview will discuss them simultaneously, pointing out any differences as necessary. For readability and brevity, only TLS will be used for the remainder of the thesis.

2.2.1 Handshake & Key Exchange

The TLS key exchange is central to the security of the protocol. The purpose is to exchange enough information that both parties can derive the same (symmetric) secret key[1] to encrypt and decrypt messages. There are several standardized ways to achieve this objective, and so both client and server need to

[1]keys plural, actually, as will be discussed later

agree on which way the exchange will occur. There is also the question of what algorithm (cipher) will be used for encryption once a shared secret is known by both parties (the symmetric key size will be dependent upon this choice). A final concern is how to protect the integrity of the connection by verifying the parties are who they claim to be, and verifying that messages passed between them have not been surreptitiously altered. TLS addresses these problems by bundling the key exchange, symmetric cipher, and integrity components into a single parameter called the "Cipher Suite." An example of a cipher suite is "TLS_RSA_WITH_AES_128_CBC_SHA."

The cipher suite is negotiated through an TLS "handshake," in which these and other parameters for the connection are established. This process, as described in the SSLv3 and TLSv1 RFCs, is depicted in Figure 2.2, consisting of at minimum a "Client Hello" and "Server Hello" in which a cipher suite is determined, a client key exchange, and a change cipher spec message.

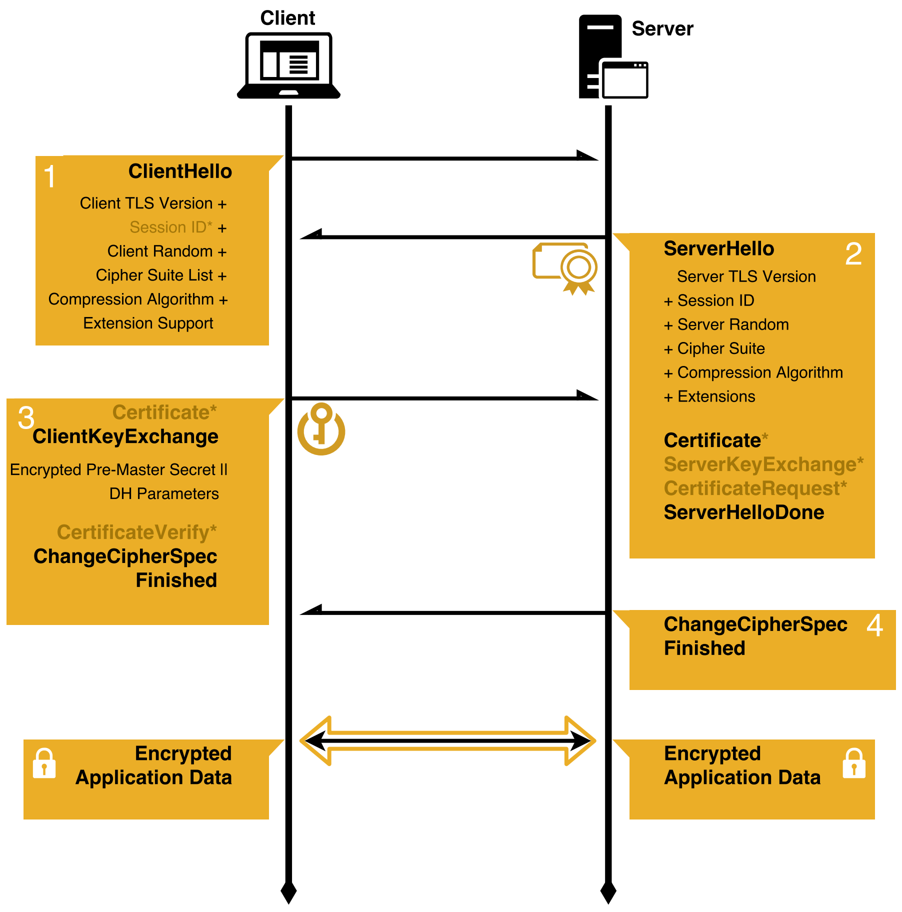


Figure 2.2. SSL/TLS Handshake (Dierks & Allen, 1999; Freier et al., 2011; Microsoft, 2003a)

A few key parameters set in this exchange are:

- The TLS version: The client and server agree on a TLS version for the connection, which is represented as a unique identifier (e.g. 0x0302 for TLS v1.1).
- The random values: The client and server random values are used to seed the PRF, and consist of a four byte timestamp followed by 28 bytes of random data.
- The session ID: Before TLS 1.2 (and in current practical implementation) the session ID is a 16 byte value often represented by 32 hexadecimal characters,

which is used to uniquely identify the session. The client should send a null session ID if it wants to start a new connection (or if tickets are being used to resume the session, discussed later).

- The cipher suite: A unique value used to represent the ciphers that will be used in the connection (e.g. 0xc028 for TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384).

The key exchange itself is the first cipher component dictated in the “Cipher Suite” chosen by the server, and is commonly one of several options specified in RFCs:

- Rivest-Shamir-Adelman (RSA)
- Diffie-Helman Ephmeral (DHE)
- Elliptic Curve DHE (ECDHE)
- Secure Remote Password (SRP)
- Pre-Shared Key (PSK)

The choice of cipher suite will also determine what messages follow. An RSA key exchange does not contain a “ServerKeyExchange” message, whereas DHE and ECDHE do. This is important, because the ServerKeyExchange is what allows the Pre-Master Secret (PMS) to be derived by both parties without the client sending it over the network, and as a result part of what enables the property of Perfect Forward Secrecy.

2.2.2 Perfect Forward Secrecy

Perfect Forward Secrecy (PFS) is a property of secure communications that employ key exchanges (which, by their nature, may be observable by third parties). Whitfield Diffie described PFS in his paper “Authentication and Authenticated Key Exchanges,” noting that “An authenticated key exchange protocol provides perfect forward secrecy if disclosure of long-term secret keying material does not compromise the secrecy of the exchanged keys from earlier runs” (Diffie, Van Oorschot, & Wiener, 1992, p. 7). In the context of TLS, this refers to the compromise of the private key.

The issue with non-PFS key exchanges, like RSA, is that a private key can be used to retroactively decrypt previously captured connections indefinitely. This is because the public key encrypts the Pre-Master Key (PMK) and sends it to the Server. The Pre-Master Key (PMK) is used to derive the master key, which in turn is used with public parameters to derive the session keys.

Ephemeral key exchanges address this in part by creating and rotating short-lived keys that are not persisted to disk (and, in theory, should not be persisted between connections). They also do not exchange a secret encrypted with this key over the network. Instead, Diffie-Helman relies on the communicative property of exponents to exchange a public value (g) raised to the power of a secret modulo a large prime number. If the client and server are Alice and Bob[2], then Alice would send $g^a \ mod \ p$ to bob, where g is a public base number, a is Alice's secret and p is a large prime number. Bob would do the same with his secret, and send it to Alice. Bob would then take the value that Alice sent and raise it to his secret value *mod p*. Alice does the same and the result is that both Alice and Bob have arrived at the same secret.

This helps achieve the property of PFS by ensuring that the values exchanged are not reliant on a persistently stored secret, and that a secret relying on that key is not exchanged across the network. For the sake of the thesis, any suite that is ephemeral will be said to be a TLS implementation of PFS. However, as will be discussed, another aspect of TLS negates some of the benefit provided by storing secrets in main memory for what might be termed an extended period of time with respect to the life of the connection. Another key consideration to remember is that the client may send support for an ephemeral cipher suite in its hello, but the suite chosen comes down to the Server's preference.

[2]A common method of representing a party "A" and "B"

2.2.3 Key Calculation

Having successfully exchanged or derived a secret, known as the "Pre-Master Secret (PMS)" both parties now generate the master secret or "master key." Whereas the PMK may vary in size, the master secret is always 48 bytes. The method used to generate the master secret for TLS is shown in Figure 2.3, as it appears in the RFCs.

```
master_secret = PRF(pre_master_secret, "master secret",
                    ClientHello.random + ServerHello.random)
                    [0..47];
```

Figure 2.3. TLS master secret generation pseudo-code (Dierks & Allen, 1999; Dierks & Rescorla, 2008)

The master secret is then, in turn, used to generate a series of session keys and unique values. This occurs by expanding the master secret (much like a key schedule), and then passing the master secret, the "key expansion," and the client and server random value into a PRF, yielding a "key block." The key block is then partitioned into keys in the following order[3] (Dierks & Rescorla, 2008):

- Client MAC key
- Server MAC key
- Client write key
- Server write key
- Client (write) IV
- Server (write) IV

The implications of this for the thesis are that there will be multiple matching symmetric keys on either side of the connection (at least for the duration of the connection), in addition to a fixed-length master key that may exist.

[3]Not all cipher suites will use the IVs, which is noted in the RFC

2.2.4 Session Resumption

TLS session resumption is a feature designed to enhance performance by enabling an abridged negotiation on subsequent reconnects between the same client and server. This is useful in situations when multiple connections may be made between the same hosts over a period of time (or even between a given client and cluster of servers). The general premise is that after an initial full key negotiation, some portion of the "state" of the connection, including at minimum a unique session identifier and the master key, is cached. When the client wishes to reconnect to the same server (or cluster of servers), it sends the unique identifier from its cache with the client hello message. If both parties have access to the master key tied uniquely to the session identified provided by the client, a willing server would send a "change cipher spec" message indicating that the symmetric cipher will be now be used, and the session resumes without another key exchange (Dierks & Allen, 1999; Freier et al., 2011; Salowey, Zhou, Eronen, & Tschofenig, 2008). The abridged handshake can be seen in Figure 2.4.

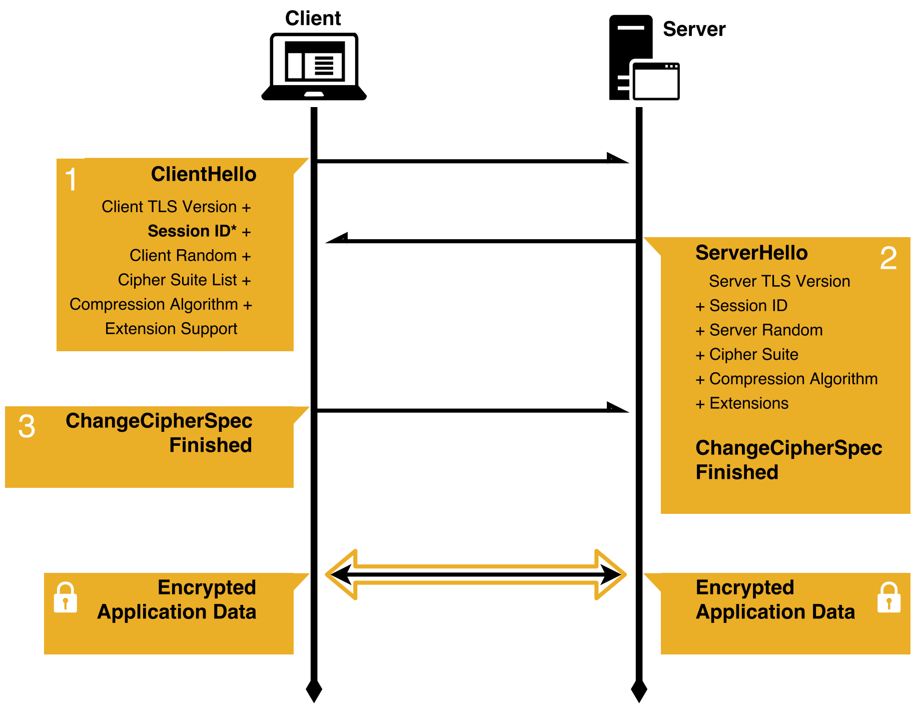


Figure 2.4. TLS session resumption abridged Handshake

There are two widely supported forms of unique identifier – the session ID, which is a component of the standard TLS implementations, and the session ticket, which is currently a TLS extension (Freier et al., 2011; Salowey et al., 2008). The session ID was described while discussing the TLS handshake, and is a fixed-length identifier of 32 bytes first introduced in SSL. The session ticket, conversely, is variable length because it is actually an encrypted composite structure of values representing the state.

The premise behind session tickets per RFC 5077, is that a server can create a “ticket” out of the state information typically saved in the cache, encrypt this state, and send it to the client for keeping. The advantage to the server is several fold: it no longer has to maintain the state in its cache, freeing up resources, and technically it does not have to be the server that the client resumes the connection with. Consider that a cluster of servers can share the key used to encrypt tickets, and thereby all resume sessions created by any of the servers, allowing for more effective load balancing.

session tickets are "opaque" to the client, meaning that the client does not know the contents of the tickets (nor does it need to know that information to resume the session). The client therefore caches the ticket with the master secret, as it would when using a session ID as a unique identifier. The exact elements that make up the ticket and "state" of which the opaque ticket is composed are not mandated by the RFC. A ticket construction put forward in the RFC is detailed in Figure 2.5.

```
struct {
         opaque key_name[16];
         opaque iv[16];
         opaque encrypted_state<0..2^16-1>;
         opaque mac[32];
     } ticket;
```

Figure 2.5. Session Ticket per RFC 5077

It's important to bear in mind that session tickets remain an extension of TLS, meaning that clients that are TLS compliant may elect not to use this extension and should still be able to inter-operate with Servers playing by the RFCs. It is at the client's option to initiate the use of session tickets by passing the extension in the client hello message. RFC 5077 also alters the standard handshake by introducing a "NewSessionTicket" message that the server sends after the ClientKeyExchange is finished.

Session tickets are intended to be mutually exclusive of a session ID. If the server intends to use session tickets, then, importantly, it should include an empty session ID in the server hello message. The client, likewise, should discard any session ID passed to it if the client receives a session ticket from the server (Salowey et al., 2008). Curiously, the RFC states that if a client sends a session ID during a session resumption, the server is supposed to respond with that same ID (presumably for consistent behaviour). This is purported to aide the client in differentiating when a server is resuming the session or performing a full handshake.

Wireshark currently supports decryption of sessions that employ session tickets based on the client random. That is to say that Wireshark requires the initial handshake and associated client random value to pair the master secret to a session using tickets. Wireshark is a well-known network analysis tool that will be leveraged in the thesis for PoC decryption of sessions. The implication of this is that the client random will need to be explicitly linked to the master secret in cases where the session ID is not present for ease of decryption (or, alternatively, the ticket will need to be decrypted).

Another point of interest is the Extended Master Secret SSL/TLS extension. This extension is proposed in RFC 7627 as a result of a weakness in the way session resumption works. The weakness is called a "triple handshake attack," essentially amounting to a man-in-the-middle attack that allows decryption of the PMK or session keys in a specific scenario (Bhargavan, Delignat-Lavaud, Pironti, Langley, & Ray, 2015). The fix, according to the RFC is to "contextually [bind] the master secret to a log of the full handshake that computes it, thus preventing such attacks" (Bhargavan et al., 2015, p. 1). Importantly for the thesis, this is implemented by changing the values seeded to the pseudo-random function to be more connection parameter specific; it does nothing to effect how the master key, once computed, is stored. The RFC does mention "shredding" of the PMK in memory as soon as possible.

Incidentally, RFC 7627 also mentions RFC 5077 (session tickets), indicating that the extended master secret does not necessarily cover session tickets. Specifically, "[...] if the client and server support the session ticket extension [RFC5077], the session hash does not cover the new session ticket sent by the server. Hence, a man-in-the-middle may be able to cause a client to store a session ticket that was not meant for the current session" (Bhargavan et al., 2015, p. 12).

As mentioned, Wireshark is a well-known and industry standard tool for network protocol analysis. It performs semi-automated analysis and protocol parsing through the use of "dissectors." This information is relevant because public and static values implemented in the RFCs can be seen in the source code of the SSL protocol dissector (Wireshark, 2015). These values, along with others discussed in the ensuing section on Windows implementation, will provide important context during searches for connection specific values and secrets.

2.3 Windows Internals

This section will review the components of the Windows operating system that may have a bearing on the thesis. Memory management is reviewed briefly to point out mechanisms that may effect design decisions that are incorporated into the methodology. The TLS implementation is also reviewed, looking at potential artifacts that should exist within the implementation, and where those artifacts may be found.

2.3.1 Memory Management

One of the most important aspects of memory management is how to efficiently virtualize the relatively constrained resource that is physical main memory and present the illusion to each process that it has it's own full, contiguous address space. This is generally accomplished through two mechanisms: Paging and Segmentation. These concepts are implemented through a combined effort of the hardware and operating system working in tandem.

The Windows operating systems expressly forgo segmentation per the Microsoft documentation. It is not actually possible to disable segmentation on 32-bit Intel hardware without disabling "protected-mode," the mode required to enable paging. Windows therefore takes advantage of the "flat model" of segmentation, in which a single contiguous segment is presented to the operating system and applications (in reality, there are at least two overlaid contiguous segments for code and data) (Intel Corporation, 2015). In contrast, Intel 64 architecture (ostensibly IA-32e) does not implement segmentation (Intel Corporation, 2015, p. 5-5).

Paging is supported from the Windows perspective through the storage of the Directory Table Base (DTB) and associated physical pages. This is a value stored within the _EPROCESS structure of each process and is the physical address of the process's page directory table. This value is what gets loaded into the CR3 hardware register during a context switch, and is an essential component of rebuilding a process's virtual address space forensically.

Windows extends paging further thorough a mechanism dubbed Virtual Address Descriptors (VADs). A VAD represents a given contiguous allocation of virtual memory pages (Ligh et al., 2014). This contiguous allocation can provide interesting insight, as it represents another layer of abstraction beyond the page table, functionally grouping address ranges into items like files and heaps.

The relevance of VADs to the thesis specifically is that they also provide access control information as to whether the VAD is readable, writable, and executable (amongst other states). When scanning for artifacts in a process address space, one could possibly narrow the scan to all read-writable pages. This would significantly improve performance by passing over regions that are not writable and would not contain a committed ephemeral structure.

2.3.1.1. Virtual Address Space

An important concept to grasp when considering any operating system forensics is the relationship between the underlying executive processes of the OS and the guest process that they service and supervise. The two roles of supervisor and guest are separated by privileges and typically denoted as "Kernel-mode" and

"User-mode" processes respectively. When a user-mode process wishes to perform a privileged action, such as accessing the hardware for I/O, it "traps" into the "Kernel-mode" OS process, which performs this action on its behalf.

In order to accomplish the task above, Windows splits the virtual address space of each process into two parts. The first part of the virtual address space is for the process itself as a user-mode address space, unique to each process and isolated. On 32-bit systems this is a value of 2GB by default, extensible to 3GB, and on 64-bit systems the user-mode space can be 8TB by default (Microsoft, n.d.-e). The second part, however, is the kernel address space, which contains both process specific structures and components that are mapped across all processes[4]. This is actually a relic of the design from VAX/VMS, which shared one of the main developers of Windows NT (one David N. Cutler) (Russinovich, Solomon, & Allchin, 2005).

The relevance this bears to the thesis is several fold. The first reason is that pointers are address, so scanning physical memory will only be so helpful before it becomes important to identify the virtual address space of the process. A physical page can also be mapped into multiple virtual address spaces, meaning that a single physical match could actually be relevant to multiple user-mode processes. The second is that processes share a writable portion of the kernel-mode address space in the form of "pools" which are essentially system heaps. The third is that user-mode process dumps do not include the kernel-mode portion of the virtual address space, so scanning the kernel-address space may become an important separate component of the methodology.

[4]there is also the concept of session space, which is mapped across all processes within the same session, but is functionally similar to "system space" and contained within the kernel-mode part of the virtual address space (Russinovich, Solomon, & Ionescu, 2012)

2.3.1.2. Virtual Memory Allocation

Data of arbitrary size is often allocated on the "heap" (or heaps), an area explicitly designated for such operations. In the Windows implementation, the kernel mode "_EPROCESS" structure contains a sub-component known as the Process Execution Block (PEB). The PEB contains two members of interest: a pointer "ProcessHeaps" and a numeric field "NumberOfHeaps." The ProcessHeaps pointer references an array of pointers to the heaps (the number of which is kept in the NumberOfHeaps member) (Microsoft, n.d.-h).

It may at first seem as though this would be the only information required, and knowledge of the pages or VADs would be superfluous to the task of scanning the address space efficiently. This assumption, however, is flawed because the heap is not the only place an application can store or load writable data. A specific example is that any memory allocated with "VirtualAlloc()" will not be part of the heap (Ligh et al., 2014, p. 192).

Windows also contains the concept of "Pools," which can essentially be thought of as Kernel-mode heaps. There are two types of pools: Paged and Non-Paged. As the name states, the paged pool contains elements that can be paged to the page file, whereas the non-paged pool contains elements which should and will not be paged (Microsoft, n.d.-f). Each pool contains sections identified by a "pool tag," which is composed of four characters. If the keys generated by the cache are in someway shared between processes, then they may exist in one of the pools, and may further be marked by a unique pool tag.

2.3.2 TLS Implementation

Microsoft implements TLS through the aptly named "Secure Channel" (Schannel) Security Support Provider (SSP). According to Microsoft:

> "The TLS protocols, SSL protocols, DTLS protocol, and the Private Communications Transport (PCT) protocol are based on public key cryptography. The Schannel authentication protocol suite provides these protocols." (Microsoft, 2015d)

Schannel then relies on several underlying cryptographic providers to handle key generation and management. The entire suite of cryptographic providers is part of the CryptoAPI Next Generation (CNG).

CNG provides a host of features, including Pseudo-Random Number Generators (PRNGs), NSA Suite B support, process isolation for "long-term keys" and kernel-mode cryptography that extend services like S/MIME, Key Storage, the Data Protection API (DPAPI) used for secure secret storage like Certificates, and of course the cipher suites leveraged by TLS. CNG was introduced in Windows Vista and will eventually completely replace the CryptoAPI which has provided such services previously, though the documentation claims that CNG will continue to provide legacy support (Microsoft, 2014a).

2.3.2.1. DPAPI

The DPAPI is a central component to the cryptographic services provided by Windows to developers, allowing encryption of arbitrary secrets as "blobs" through a very simple interface. The entire scheme leveraged by DPAPI is outside of the scope of this literature review, but it bears relevance in that Private Certificates that are stored persistently on disk are protected via this mechanism (Microsoft, 2014b).

Private Keys are stored embedded within the certificate in a DPAPI blob, though the rest of the certificate (including the public key) remains unencrypted (Picod, 2016). A lot of great work has been done on reversing DPAPI, which is discussed in the "prior work" section of this literature review.

2.3.2.2. Key Isolation

The CNG documentation explicitly devotes a section to key isolation, perhaps the most important detail of the implementation. Microsoft explicitly states the following requirement: “To comply with common criteria (CC) requirements, the long-lived keys must be isolated so that they are never present in the application process” (Microsoft, 2014b). To meet this criteria, keys are isolated into a single process; fittingly, the LSASS process was chosen.

Furthermore, the documentation states that access to all private keys is handled by the “key storage router,” the functionality of which is placed exclusively within Ncrypt.dll (Microsoft, 2014b). This revelation is critical as it suggests that any key negotiations for SSL/TLS will route through Ncrypt. Key isolation is enabled by default beginning with Windows Vista/Server 2008. The model for this isolation from Microsoft can be seen in Figure 2.6

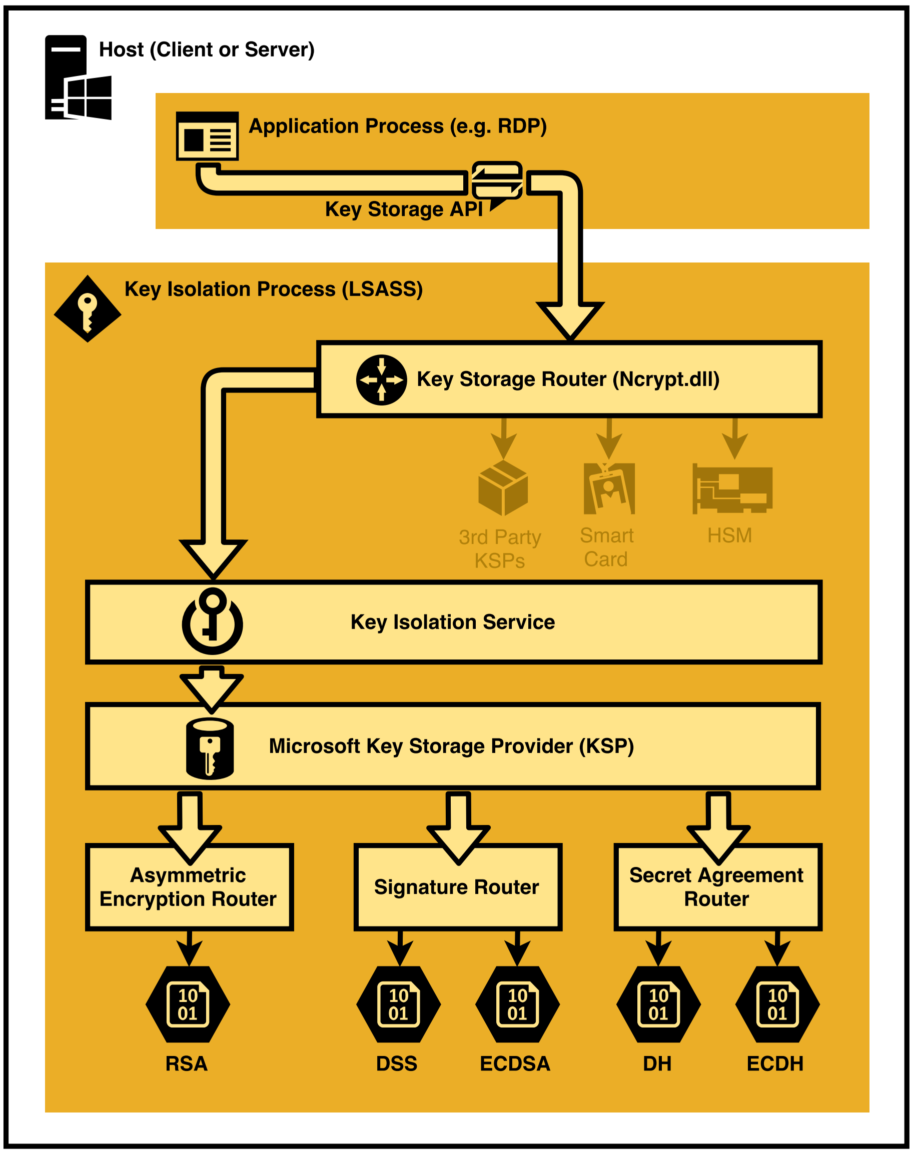

Host (Client or Server)
Application Process (e.g. RDP)
Key Storage API
Key Isolation Process (LSASS)
Key Storage Router (Ncrypt.dll)
3rd Party KSPs
Smart Card
HSM
Key Isolation Service
Microsoft Key Storage Provider (KSP)
Asymmetric Encryption Router
Signature Router
Secret Agreement Router
RSA
DSS
ECDSA
DH
ECDH

Figure 2.6. Microsoft Key Isolation paradigm (Microsoft, 2014b)

2.3.2.3. Schannel

Schannel is loaded into LSASS, along with the process that is leveraging TLS to encrypt and decrypt data. The request for key material is passed to the LSASS process through another user-mode DLL within the requesting process. This flow is depicted in figure 2.7 from Microsoft.

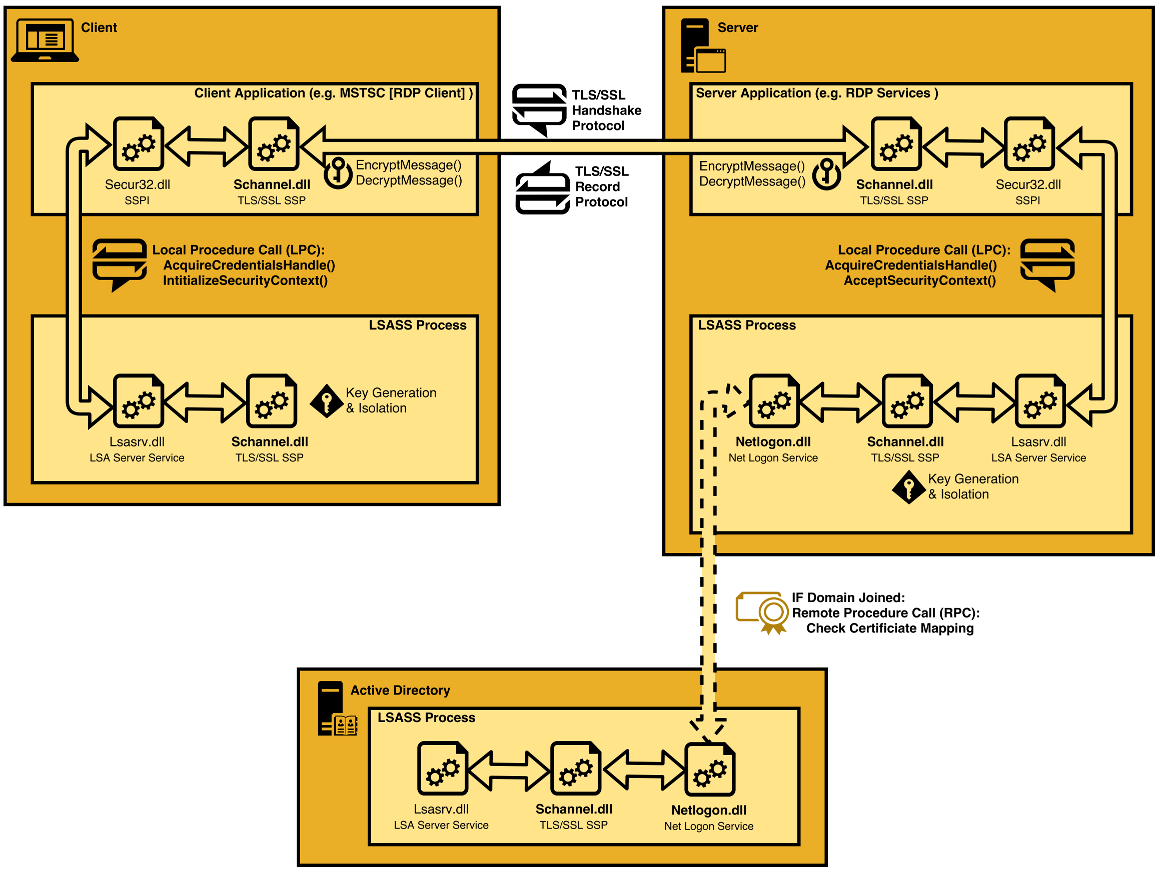


Figure 2.7. Schannel SSP architecture (Microsoft, 2015d)

A few points of interest pertaining to the Schannel implementation were scattered throughout the documentation. The pertinent questions about capability of Schannel for the thesis surrounds three main topics:

1. Support for session resumption
2. Order of preference for cipher suites
3. Schannel-specific constants

Microsoft states that they added support for RFC 5077 session resumption in Windows 8 and Server 2012 R2. This extends to clients like Windows phone and

Windows RT. The article also makes an interesting statement that RFC 5077 support would allow a cluster of servers to resume each others tickets (Microsoft, 2014d), indicating that somehow the key to decrypt these session tickets must be shared between servers.

The order for preference of cipher suites is listed in a section of the Schannel SSP documentation, and is shown as consistent from Vista forward (the point at which CNG, and thus AES cipher suites, were introduced) (Microsoft, 2014c). This document shows a preference for RSA key exchanges. Separate documentation indicates that the cipher suite preference can be changed through group policy or programmatically and can be queried using "BCryptEnumContextFunctions," which returns them in the order of priority. Incidentally, this function denotes bcrypt as a library of interest. It is described by Microsoft at the "Windows Cryptographic Primitives Library" (Microsoft, n.d.-d). The author noted a discrepancy with the schannel SSP documentation and anecdotal observations of the cipher suite preferences in practice. The author then compiled the Microsoft provided "BCryptEnumContextFunctions" sample code on the same documentation page (Microsoft, 2014c, p. 1) to list cipher suites in order of preference. Running this code across several test systems revealed a different order than was listed in the Schannel documentation, notably with Ephemeral suites preferred.

The CNG function and structure documentation contain some constants that may appear in Schannel, but surprisingly fewer than was hoped. Because of this, the available header files were examined. Relevant header files were identified through the CNG documentation, and were noted to be "bcrypt.h," "ncrypt.h," and "sslprovider.h." Unique constants were identified that could be used to locate structures, like keys, in memory. Examples include the magic values for RSA and ECC private keys, cipher suite identifiers, TLS version identifiers, and flag values.

2.4 Prior Work

This final section of the literature review focuses on techniques that exist for identification and extraction of keys or other SSL secrets from memory.

2.4.1 Key Identification

This paper would likely be remiss to review key identification with respect to SSL/TLS without mentioning the paper "Playing hide and seek with stored keys" by Shamir and van Someren (1998). The paper discusses mechanisms for identifying RSA keys in "Gigabytes of data," which at the time referred to secondary storage, but holds true today, particularly "efficient algebraic attacks which can locate secret RSA keys in long bit strings" (Shamir & van Someren, 1998, p.1).

Many of the techniques described are still valid and apply to ephemeral keys as well as persisted keys, but some are impractical considering the size of the data set and the evolution of stored data over time. For instance, entropy based attacks against a process that routinely generates random bits and handles encrypted blobs, as LSASS does, will likely not yield efficient and meaningful results. Consider 32-byte P-256 private keys for instance and compare that to any other pseudo-random value like a GUID of some kind (even within SSL to values such as the client/server random or session ID). Additionally, the private keys may be encrypted in memory, which would provide a match when employing such a method, but not in a form that is directly usable. The paper addresses sound generalizable approaches, however implementation specific details will likely prove more efficient in the context of the thesis, as will be examined later in this section.

In addition to RSA Keys, AES keys may also exist in memory. AES is the de-facto standard for symmetric cipher amongst the TLS cipher suites. In fact, TLS_RSA_WITH_AES_128_GCM_SHA256 is the only cipher suite required for TLS 1.2 in the NIST TLS Implementation guidelines (Polk, McKay, & Chokhani, 2014). It plays the role of the session key that is derived after the PMK is exchanged or derived. Because the symmetric key must exist while the connection is active in order to encrypt and decrypt data, this is an artifact that will definitely be sought in memory. At issue, however, is that the AES key is intended to be pseudo-randomly generated, so it should be indistinguishable unless stored in a specific context.

One paper entitled "Lest we remember: cold-boot attacks on encryption keys" discusses such context that typically appears across implementations (Halderman et al., 2008). Specifically, a novel technique was developed based on the AES key schedule, which is not random. The key schedule is described described in FIPS-197, and is a mechanism to take the relatively short key material, and "expand" it into a number of round keys for use in different rounds of the algorithm. The product of this operation is of a fixed size as there are a fixed number of rounds per keys-size, and is also referred to as the key schedule. FIPS-197 notes that "The Key Expansion generates a total of $N_b(N_r + 1)$ words[5]" (NIST, 2001, p. 19), where N_b is a fixed value of 4 representing the number of columns in the state and N_r is the number of rounds, which is 10 for 128-bit keys, 12 for 192-bit keys, and 14 for 256-bit keys (NIST, 2001, p. 7). The schedule is then 176 bytes for 128-bit keys, 208 bytes for 192-bit keys, and 224 bytes for 256-bit keys. Because the schedule is deterministically based on the key and will be used every time a specific key is used, it is seen as computationally expensive to recalculate and destroy the schedule repeatedly, which is ostensibly why it is often stored.

Not only do the authors discuss identification of AES keys, but RSA keys as well, going so far as to release PoC source code to detect both. Since this seminal work, tools such as FindAES by Jesse Kornblum and the subsequent functionality ported to the scanning tool bulk_extractor, curated by Simson Garfinkel, also implement key schedule based scanning (Garfinkel, 2013). Clearly this method relies on the implementer storing the schedule, but this appears by the accounts of the paper to be common practice (Halderman et al., 2008).

[5]A word in this context is 4 bytes

An even more direct revelation concerning key identification was discovered in a paper about identifying weaknesses in Dual EC entitled “On the Practical Exploitability of Dual EC in TLS Implementations” (Checkoway et al., 2014). The examination of specific implementations meant that Schannel was considered, and in the case of Schannel the authors “focus on ECDHE/ECDSA handshakes that use P-256” (Checkoway et al., 2014, p. 8). They note that all information they acquire and disseminate about Schannel in the paper is the fruit of their reverse engineering efforts. The authors also note the fact that LSASS handles the TLS handshake and key derivation. The other findings relevant to this thesis are noted here:

- bcryptprimatives.dll implements the user-mode CNG API
- Requested random bytes for key material are not cached
- Schannel requests 40 bytes for an ephemeral P-256 private key (even though only 32 are used)
- Schannel “caches ephemeral keys for two hours (this timeout is hard-coded in the configurations we examined)” (Checkoway et al., 2014, p. 8)
- Schannel generates a session ID that is implementation unique and can be roughly fingerprinted
- Schannel has a hard-coded cache size of 20,000 entries (per the authors’ examination) (Checkoway et al., 2014, p. 9)

These findings provide clues about key caching, hard-coded values, and temporal context about operations occurring around key generation. The bcryptprimatives module can now be targeted as part of the analysis, adding to the other modules identified while reviewing the Microsoft documentation. Additionally, the fact that Schannel generates 40 random bytes means that, in the event that these random bytes are stored together with the ephemeral key, they may help identify it as such when looking the key on the heap. Hard-coded values like the 2 hour timeout and the 20,000 cache are also somewhat unique values that may be used to locate and target specific functions for reverse engineering efforts.

The session ID generation mechanism is of particular instance, as a non-random component to the session ID may make it more distinguishable when searching. This is incredibly desirable as all of the public Secret values are randomly generated or based on something mutable like time that does not make for flexible scanning features. The session ID, according to the paper, gets it's fingerprint from the "hard-coded" cache length value of 20,000. The paper states that the first four bytes of the 32 random bytes requested from "BCryptGenRandom" are taken as an integer and replaced with the modulus of that number against the cache length value, producing two bytes of zeros in the third and fourth position (Checkoway et al., 2014, p. 9).

2.4.2 SSL/TLS Decryption

Perhaps the most common and well documented example of SSL/TLS decryption is the implementation in Wireshark. Wireshark has both a graphical interface and scripting engine that make it a powerful analysis tool, particularly for small data sets like the ones that will be examined in the thesis experimentation. As far as the thesis is concerned, the actual internals of the decryption process are irrelevant excepting that the overall process is sane and can be relied upon. More interesting in this context are what parameters and preconditions Wireshark requires to perform decryption of an arbitrary connection, and how parameters may be passed.

Wireshark's source code (Wireshark, 2015) shows that it accepts an "SSL keylog file" as a text file with secrets in the following formats[6]:

- RSA Session-ID:$< session\_id >$ Master-Key:$< master\_secret >$
- CLIENT_RANDOM $< session\_id >$ $< master\_secret >$
- RSA $< encrypted\_pre\_master\_secret >< pre\_master\_secret >$

[6]Any value in brackets denotes a value to be replaced with the hexadecimal representation of the binary value

- PMS_CLIENT_RANDOM $< client\_random >< pre\_master\_secret >$

The source code also shows that Wireshark accepts a Pre-Shared Key (PSK), which is given as a hexadecimal string representing between 0 and 16 bytes (Wireshark, 2015). Finally, one can provide the provide an RSA private key in X.509, Privacy-enhanced Electronic Mail (PEM), or Public Key Cryptography Standards (PKCS) #12 format (Wireshark, 2015).

One of the inspirational sources for the topic of this thesis was a series of blog posts about decrypting RDP traffic using the private key stored persistently on Windows systems. Both posts noted that:

1. The full key exchange needed to be captured
2. Sessions which used ephemeral cipher suites (and by extension, the TLS implementation of PFS) were incompatible with this approach

The blog post by Steve Elliot of Contexis went on to leverage FreeRDP to rebuild video and keyboard data from the session, replaying it in real-time or at extended speed (Elliot, 2014). Contexis, however, did not make that tool available to the general public, and it is hoped that a method developed here could be leveraged to do so. Further, if successful, both barriers posed would be negated within the time frame of the session cache life.

2.4.3 Key Extraction

An encoding-specific detection mechanism for RSA keys was discussed in 2006 by Tobias Klein, in a paper entitled “All your private keys are belong to us (Extracting RSA private keys and certificates from process memory).” The primary premise of the paper is that all common binary Private Key formats leverage some form of Abstract Syntax Notation One (ASN.1) encoding (The author specifically mentions the PKCS #8 and x.509 standards). ASN.1 generally follows a Type-Length-Value (TLV) structure, and the author correctly notes that all certificates begin with the tag “SEQUENCE” which is represented by the ID 0x30, followed by the length field, which is a single byte unless the value is greater than 127 bytes long (the highest order bit is set). If this is the case, then the length bytes notes how many following bytes are used to express the total length. So, in the case of certificates with RSA public keys, this value is often “0x82” or “1000 0010” in binary, meaning that the two following bytes express the length of the certificate, giving them a common signature of “0x3082.” The author also notes that Private Keys are then followed by a TLV of “0x020100” representing the version (Klein, 2006, p. 2). Windows leverages these encoding schemes (Microsoft, 2015a), making it likely that this technique could apply to the thesis when observing ephemeral private keys.

In 2010, Jean-Michel Picod and Elie Bursztein presented work at BlackHat about reverse engineering Windows Data Protection API (DPAPI), and provided a PoC tool known as DPAPIck. The particular relevance that the paper has to this thesis is that, while not explicitly discussed, the private keys are protected with DPAPI. In 2014, Francisco Picasso updated DPAPIck (now at version 0.3), and added support for Windows 8 and Windows 10 systems, though DPAPI-NG (the component of CNG that replaces DPAPI) has not been “fully reversed” (Picasso, 2015).

Amongst the ensuing updates to the work was the addition of “Probes” for parsing out specific artifacts protected by DPAPI, one of which was generically the RSA Private Key file. The catch is that most DPAPI secrets are indirectly protected by a user password, where as System DPAPI secrets require a system “password” that is protect by LSA Secrets. The author of this thesis leveraged the RSA “Probe” by writing a quick interface for it based on an example provided by Fransico Picasso for decrypting a different system secret (Wi-Fi credentials). This allowed the author to later decrypt the RDP private key to search for it in memory (still used for signing, even when ephemeral cipher suites were leveraged).

Another closely related work was presented by Jason Geffner at BlackHat 2011, entitled “Exporting Non-Exportable RSA Keys.” In this work, the author looks for ways to export private keys in both CryptoAPI and CNG, successfully completing both tasks by flipping the exportable flag accordingly. Of particular interest is his work with CNG, in which he reverse engineers some important functions of CNG through debugging and dis-assembly (Geffner, 2011).

The venerable Mimikatz tool developed by Benjamin Delphy is also capable of decrypting and exporting Certificates stored with CNG, as can be seen in its source code on Github (Delphy, 2016a). Mimikatz can operate on a live machine or against a user-mode dump file, something which Volatility and Rekall currently do not do. Mimikatz also provides a Dynamic Link Library (DLL) which can be loaded into WinDbg to leverage the tool in concert with Microsoft's own debugging and analysis platform (Delphy, 2013).

Perhaps most directly in line with the goal of the thesis are several community plug-ins that have been developed for Volatility and target TLS related artifacts. A brief list of those available on the Volatility Github repository, including the community repository, appears below alongside the docstring provided by the plug-in and the author:

- dumpcerts (vol): Dump RSA private and public SSL keys
- haystack (Loïc Jaqueme): Search for a record in all the memory space.
- linux_ssh_agent_key (Ying Li): Get SSH keys from ssh-agent process heaps
- rsakey (Philip Huppert): Extract Base64/PPEM encoded private RSA keys from physical memory.

Importantly, none of these plug-ins address decryption or identification of connections using ephemeral cipher suites. The ssh agent key plug-in specifically targets Linux systems, but was reviewed to identify any reusable logic that could apply to finding ephemeral private keys. The "RSAKey" plug-in is particularly simple, scanning physical memory in fixed chunk sizes for the string "—–BEGIN RSA PRIVATE KEY—–" as an identifier for keys in the PEM format. The comments in the plug-in note that, for simplicity, the plug-in does not use overlap between chunks, meaning that cross-boundary misses may occur (Huppert, 2015).

While documentation suggests that Windows supports importing the PEM format (Microsoft, n.d.-a), the storage is unified into a different format and the private keys encrypted via DPAPI (Microsoft, 2014b). It is therefore unlikely that this would reveal keys in LSASS memory. Finally, the dumpcerts plug-ins is an implementation of the “trapkit” method discussed earlier, per the comments in the plug-in itself (Volatility, 2015).

The closest functionality to the task at hand and most versatile of these plug-ins is “haystack,” which is not SSL specific. It is designed to identify heaps within processes not necessarily listed in the PEB, and identify the specific allocations on the heap. In the README documentation, an example is given whereby the author notes that one can extract the OpenSSL session records to obtain the session keys using this plug-in (and the provided structure) (Jaqueme, 2015).

2.4.4 Perfect Forward Secrecy

Up until this point in the research, all efforts have been focused on obtaining the persistent private key for decryption. None have focus on cipher suites embracing PFS, which render the persistent key useless for decryption (though not impersonation, as it is still leveraged for identity). This section reviews research which instead focuses on ephemeral cipher suites and methods for subverting them.

One of the most informative and thorough sources on subverting PFS, though it provides no implementation, is a blog post by Tim Taubert from November of 2014. The post explains how TLS session tickets, as implemented, break the property of Perfect Forward Secrecy, by allowing the state (including the master key) to be transmitted, and thus captured by an adversary, which could be potentially decrypted at any point in the future, leading to decryption of the connection (Taubert, 2014). Tim goes on to give examples of configuring the cache settings in different Web Server implementations to mitigate this as best as possible.

The post itself cites several other works about issues with the TLS handling of PFS, including the work described below.

The most recent and directly relative work was presented at BlackHat USA in 2013, entitled “TLS ‘Secrets.”’ This presentation discusses implementing a mechanism for extracting OpenSSL cache entries, specifically looking at Unix-like systems. The focus of the presentation was actually heavily oriented around RFC 5077 as implemented by OpenSSL. A major finding was the fact that the Specification, in an attempt to remain implementation agnostic, does not provide any particulars or guidance on key usage or storage. OpenSSL was found to use 128 bit AES keys to encrypt tickets, which did not get rotated according to the author. The author sums up the impact of this finding by stating that “128 bit of security is all you get (at best), regardless of the cipher which has been negociated [sic]” (Daigniere, 2013, p. 8). This work cannot directly translate to the Windows implementation, sadly, but shows that the concept upon which the thesis is based is sound.

Finally, a paper entitled “Perfect forward not so secrecy” seemingly completed in December 2013 also looks at associating session keys with captures TLS sessions and, interestingly, how to further limit access to these keys (Dreijer & Rijs, 2013). The paper focuses on OpenSSL server-side implementations and, like the BlackHat USA presentation, on RFC 5077. The authors develop a python PoC to extract the master key with session ID from OpenSSL implementations (Dreijer & Rijs, 2013, p. 8).

CHAPTER 3. RESEARCH METHODOLOGY

This chapter focuses on developing a logical, methodical, and practical way to go about identifying whether secrets and other TLS artifacts exist in LSASS memory, and, if so, whether they maintain a discernible relationship in memory. The exploratory nature of the research means that such artifacts or relationships may not exist; however, like other examples where encryption is employed, a key generated and used ephemerally may reside in volatile memory for some duration. Other artifacts from the transaction should also exist, particularly because connections are stateful and the operating system manages the creation, maintenance, and termination of these connections. There are also reasons why artifacts may necessarily outlive the connection itself for some time (consider the session resumption feature of TLS).

3.0.1 Design Decisions

The foremost design goal was accessibility and reproducibility by the community. This goal translated to the selection of tools and techniques that were exclusively either free and publicly available or open source. A clear exception to the rule was the use of Windows operating systems as these are the subject of the research; even in this case, freely available development virtual machines from Microsoft were used as much as possible towards the ease of reproducibility. A final nuance of the stated goal is that preference was given to public tools released by Microsoft specifically, in an effort to remain as “native” as possible.

Another contributor to the decision to remain native as often as possible was the second design goal: rapidity and effectiveness of analysis. By leveraging Microsoft tools, such as WinDbg, and built-in capabilities, like on-demand user-mode process dump creation, a variety of inherent benefits became available to speed up analysis. For instance:

- User-mode process dumps can be created without additional tools beginning with Windows Vista (Microsoft, 2010).
- Native creation of these dumps does not require or induce any special state for the operating system (no need to pause or restart the system), allowing rapid sampling repetition during any variable manipulation.
- The dumps are "sparse" by default, meaning that unused ranges in the virtual address space are not included. This creates a more efficient search scope that avoids unnecessarily parsing empty ranges of memory.
- The user-mode dumps can be read natively by WinDbg, which in turn can automatically leverage Microsoft's public symbols (Program Databases or PDBs) to make sense of structures and memory pointers therein. Although public, the information provided by Program Databases (PDBs) is not necessarily documented via any other source, and they have historically played an integral role in Windows memory forensics (Dolan-Gavitt, 2007).

In order to generate TLS sessions in a way that is native, RDP was used. The RDP client and server functionality is built-in to all modern releases of Windows Microsoft (2015c), and so requires very little alteration to enable testing of a given operating system. The use of RDP simultaneously minimizes external variables that would be introduced by third-party software and has the ancillary benefit of exploring one of the use-cases presented previously (Incident Response).

Finally, the desire for flexibility influenced the experimental design towards virtualization. The use of virtual machine abstraction allows for portability and repeatability (through snapshots for example). Particularly though, it enabled inspection of RAM without reading through the target operating system, thus leaving it mostly insulated to analysis. Using abstract virtualization mechanisms like pausing and snapshots, the guest physical memory could be saved to a file (VMEM) on the host, which could then be analyzed and manipulated as needed.

The VMEM format is supported by powerful open source analysis tools (e.g., Volatility and Rekall), which are able to trivially convert it to other formats. These tools were leveraged to convert the VMEM to a raw copy, in which physical offsets into the file were synonymous with offsets into physical memory. Physical offsets produced by scanning tools like bulk_extractor could then be compared with the physical offsets rendered by virtual to physical memory translations of analysis tools like Rekall and Volatility.

3.0.2 Overview

The methodology that emerged from the interaction of these design goals and the technology available to meet them is briefly highlighted in Figure 3.1 and then expanded upon in Section 3.0.3. Philosophically, there were at least two approaches to the problem of identifying the key – observing the process (live debugging and static analysis) or observing the data (post-mortem analysis). While both are valid and they are not mutually exclusive (in fact, they are combined in the articulated methodology), post-mortem analysis was given precedence for three reasons. Nominally, these are scope, simplicity, and fidelity to the problem.

The exact scope of debugging wasn't quite known – the executable(s) and functions that interact with the master key were unknown, though presumed to originate with, or orientate around, Schannel and the cryptographic libraries. While parsing PDBs ahead of time would have provided insight to develop scope, in isolation it could have introduced biases and pivoted the focus of analysis with misleading or enticing symbol names. It could have also falsely restricted the scope through exclusion of symbols (and many sensitive symbols are excluded in the public PDBs (Microsoft, 2015b)) for functions that are highly relevant. It would be possible to simply trace through a TLS transaction to generate scope, but this still requires enabling kernel debugging if attaching to the LSASS process. Manually tracing would also likely run into issues due to timing requirements when dealing with live network connections. This too is solvable through adjustments, but each adjustment to the base system moves the environment further and further from the actual parameters being emulated, thus allowing lurking variables to creep in. Any adjustment would also need to be made against every test system.

The problem itself is one that would be post-mortem – whether or not the artifacts exists after a connection has been established, possibly terminated. If the key exists temporarily during execution and is freed or destroyed very quickly, this is still interesting, but less so in the forensic context of the problem. Analyzing in a post-mortem way allowed for closer alignment with the use cases and minimized temporally-based variables to some degree. It also provided information that could be used to make more informed debugging and tracing decisions, enhancing the potency of the debugging effort.

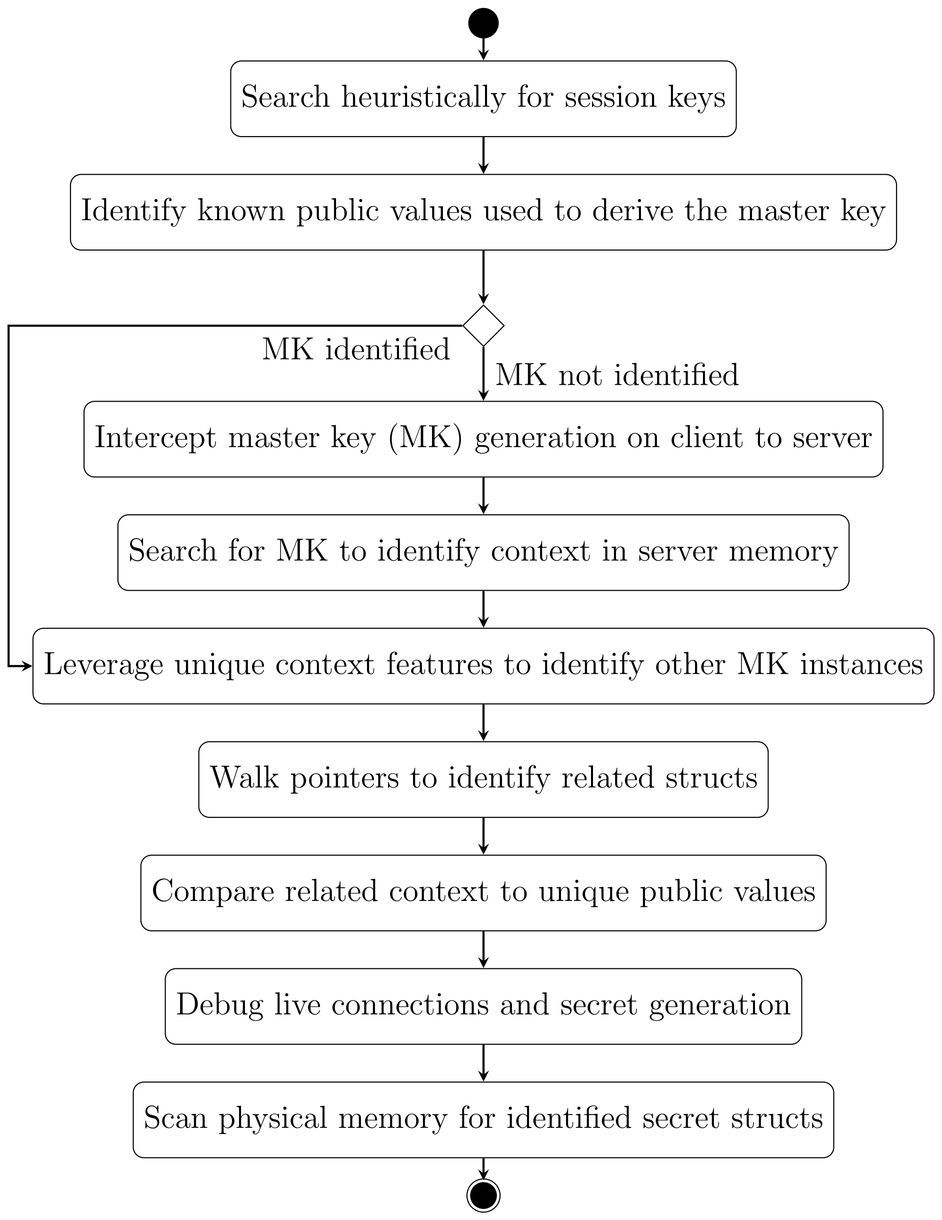

Search heuristically for session keys
Identify known public values used to derive the master key
MK identified
MK not identified
Intercept master key (MK) generation on client to server
Search for MK to identify context in server memory
Leverage unique context features to identify other MK instances
Walk pointers to identify related structs
Compare related context to unique public values
Debug live connections and secret generation
Scan physical memory for identified secret structs

Figure 3.1. TLS artifact identification methodology

This implementation, when distilled, resembles the Inman-Rudin paradigm (Identification → Classification → Individualization → Association → Reconstruction (Inman & Rudin, 2002)):

- The first several steps are used to "identify" structures that could be relevant to the event of interest (a TLS connection)

- The master key structs are "classified" and enumerated (by virtue of being pseudo-random, they are inherently "individualized")
- After identifying and classifying master keys and other structures, a given MK is then "associated" with a given event via unique features of the connection
- Finally, the associated MK and connection parameters are used to decrypt the TLS session and reconstruct the event of interest

3.0.3 Discussion of Methods

This section augments the overview of the methodology by activity depicted in Figure 3.1.

3.0.3.1. Searching for session keys

During previous research and analysis of LSASS, the author used AES key schedule heuristic scanning (via the "FindAES" tool produced by Jesse Kornblum (2011) to identify the AES keys used to encrypt user secrets in LSASS. This same technique was applied to session keys when AES was used in the cipher suite of a TLS connection. The theory was that, as symmetric keys, the AES keys used for encryption of the session would exist in both the client and the server memory for the duration of the connection (generated in the LSASS process of both if they are Windows machines). Scanning both hosts for AES key schedules may then yield two or more pairs of matching keys (the client and server write keys), which would be those used for the connection (Dierks & Allen, 1999; Freier et al., 2011).

3.0.3.2. Identifying known values

One method used to identify possible structures that could either contain or lead to secrets of a connection was to start with known-knowns. In the context of a TLS connection, that term applies to public values – either those unique to the connection, such as the client/server random values, or constant values like those defined in the RFCs (Dierks & Rescorla, 2008, p. 68). Unique values present the opportunity to enumerate and dissect artifacts of a specific connection. Conversely, constants as defined in the RFCs, or perhaps particular to the implementation, act as generic identifiers that can be used to laterally identify similar artifacts across arbitrary connections. This was useful when no particular connection was targeted or known, and instead the desire was to exhaust all connection artifacts in memory.

The reasoning behind searching specifically for public values was several-fold. The first reason was that, while secret values (e.g., keys) may be sequestered in an encrypted "vault" in memory, public values likely remain unencrypted in memory, as they are unprotected in other contexts. The second was that, intuitively, secrets are often compartmentalized, and so unique to a specific connection [1]. Further, key material is often pseudo-randomly generated in such a way that detection is theoretically impossible or impractically costly. This makes secrets less flexible as search features than looking for public values. Finally, the public values were readily accessible from a network capture, which matched a use-case identified and aided in the ease of testing.

3.0.3.3. Intercepting master key generation

The pre-master key (and subsequently master key) are the two most useful secrets in decrypting a given connection. This is because, as discussed in Chapter Two, the master key is used to derive the sessions keys, and always exists, regardless of cipher suite chosen. This very fact is the reason Perfect Forward Secrecy (PFS) is important – the private key in non-ephemeral key exchanges is used to decrypt the pre-master secret that is shared during an RSA TLS handshake. So the usefulness of the persistent private key in "ex post facto" decryption is predicated on the usefulness of the pre-master key.

[1]An exception to this in TLS is the persistent private key

Knowing the pre-master key or master key would allow for scanning of memory to identify the location and number of instances of the key in memory for a given connection, which could then be compared over several connections to identify consistent structures (particularly if combined with input fuzzing like altering the TLS version or cipher suite used in some connections).

Several popular modern browsers (e.g., Firefox and Chrome) allow for logging of master keys to a file Mozilla (2015). However, these browsers leverage a different cryptographic library, and, even so, the goal is to minimize external variables on the target host, making RDP or Internet Explorer (IE) preferred options regardless. Fortunately, several open source RDP clients exist, which rely on OpenSSL, a well documented open source SSL library. A small "shim" library can be preloaded to intercept the master key generation calls to OpenSSL and transparently dump the returned key to a file. This key can then be compared with what is resident in memory on the target (Windows) host acting as the RDP server.

3.0.3.4. Leveraging unique structure identifiers

The master key itself is pseudo-random. It is therefore desirable to locate some contextual feature that could be used as a "class" marker for identifying unknown master keys. Specifically, there likely exists some structure around, or reference to, the master key if it is both managed and memory resident. This could be co-located known values, some form of "magic number," or even pointers to the master key from other structures if it has no direct context in memory. Identifying these features will form the link between master keys and unique connection identifiers that is required for decryption. Any master key "class" markers could be portable or reusable across other instances of the same implementation (i.e, other hosts that implement the same Schannel paradigm).

Furthermore, unique "magic" values can provide especially valuable insights. Values that are magic are hard-coded or the generation thereof is hard-coded. This means that identified magic values (or the way they are generated) must exist in the functions of one or more modules loaded into memory. This fact links an artifact containing the magic value to the modules and functions that created it or use it, which is useful intelligence for live debugging.

Symbols can be leveraged to extrapolate from magic values. If the memory address of the magic value (or generator for the magic value) resolves symbolically to a function named, for example, “GenerateSessionKeys” or “CreateSSLSessionStruct,” then there is intuitive meaning and informed speculations can be made at the purpose of the structure.

A caveat hinted at earlier is that the public Windows symbol files contain only the symbols that Microsoft deems necessary for reasonable levels of debugging, so not all functions or global variables are exposed. This has a potentially insidious side-effect when performing analysis, as the “ln” function of WinDbg that performs symbol resolution lists the nearest symbols before and after a given address appended with an offset to the provided address. The implication of this is that if an address exists within a function for which symbols are not present, it may appear that the address is inside of the next-nearest symbol. For this reason, any time that function symbols are used for analysis, the function will be disassembled to ensure that the value of interest is actually contained within.

3.0.3.5. Walking pointers

Related structures are often chained together by pointers in memory. Starting with a public value, identifying its structure and pointers, and then dereferencing (or “walking”) the pointers inside of that structure will provide a series of chained structures. One issue with this approach is that it can quickly become overwhelming. Pointers can lead to exponential branching (or infinite looping in the case of lists), so it can be important to have some semblance of the purpose for the structure being observed before (potentially) needlessly walking a pointer to a value of little or no use. Pointers also exist to functions or static variables embedded in binary files, which, if these have symbols, is helpful in identifying the structure as well.

Another important aspect is walking pointers in reverse to identify what is pointing to a structure. There are many situations where the artifact of interest does not point back to structures referencing it. In this case, the simplest solution is to scan for the memory address of the structure, to see if a pointer to that memory address exists anywhere else in memory.

3.0.3.6. Comparing related structs to unique public values

This is nearly the inverse of the initial step. Once a series of structures related to the master key are identified in memory, they need to be linked to a given connection to be valuable.

Two examples of ostensibly unique public values are the session ID and the client/server random values. The random values are particularly interesting, as the first four bytes of the random are very often a timestamp (Freier et al., 2011, p. 25), which has forensic value. The client random is also the value that is included in the NSS log format used by browsers and Wireshark (although Wireshark also accepts the session ID, as mentioned in Chapter Two). As was also mentioned in the literature review, the session ID can have an interesting quality in the case of Schannel, because a hard-coded value in the session ID generation process causes them to typically have zeros in the third and fourth byte positions, acting as a rough fingerprint (Checkoway et al., 2014). This means that, to some extent, the server that was connected to can be identified as a Windows server in the case of a client. It also provides a visually recognizable pattern when performing analysis.

3.0.3.7. Debugging Local Security Authority Sub-System

Up until this stage, every part has been piecing together the reconstruction of an event (the TLS connection) and the context around that event. Debugging will then play events forward, and possibly fill in missing information within the scope of the artifacts identified, or lead to new artifacts. It also has the potential to validate or challenges assumptions made about what was seen in memory. An example is that something which is seen as managed and permanent may be less so than initially anticipated, or contrastingly, something seen as ephemeral or non-existent may actually be transitioned to a different, more permanent (even encrypted) structure. Static analysis will also likely be performed in advance of this task where applicable, based upon the complexity of the functions that are identified and examined, as that will also narrow the scope for debugging.

3.0.3.8. Scanning Physical Memory

The focal point has been LSASS memory space based largely on documentation provided by Microsoft; however, artifacts that are generated within that process do not necessarily remain within its user-mode virtual address space. This is especially salient when considering how Windows handles kernel memory (i.e., mapping it into the latter end of the process address space). Scanning physical memory may help illuminate relationships or external artifacts that were not scoped into the preceeding methodology (user-mode dumps for example do not include the kernel address space).

3.0.4 Infrastructure

This section discusses the technical instrumentation of the methodology (including tools, tool versions, and setup of the live debugging environment), so that the methods are faithfully reproducible.

The design guidelines highlighted previously are manifest in the tool selection in several demonstrable ways. At a high level, reproducibility and simplicity led to an attempt to minimize the breadth of tool use and narrow to a core of analysis suites. The linchpin of these tools is WinDbg for user-mode process dump analysis and kernel debugging. The choice to use WinDbg also meant that Windows was the choice for analysis platform. Of the versions available, Windows 10 was used as it is the latest available version from Microsoft at the time of writing. Microsoft also released a tool for basic tasks related to PDB files and executables called "cvdump" that could be leveraged for symbol inspection outside of WinDbg. To augment WinDbg as a hex viewer, a Windows-based hex editor was required, and HxD by Maël Hörz was selected. At the time of writing, HxD is freely available, comes as a single executable, and supports large file sizes with quick searching capability, a key requirement.

What HxD lacks, however, is a tunable scanning engine that can output matches quickly and easily. The "lightgrep" scanner from the bulk_extractor utility will be leveraged to this end. Bulk_extractor also supports an implementation of the "FindAES" tool mentioned previously in the form of an "aes" scanner, and was used for session key scanning[2].

Windows 10 was also selected as the initial target system, in part for the same reason that it was selected as the analysis platform. In addition to being the latest release, and Microsoft intends for it to be a "universal" operating system across devices (Warren, 2015). This is evident not only from statements by the company, but actions like offering free upgrades from previous versions of Windows (Microsoft, 2015f) and announcing plans to upgrade one billion devices to Windows 10 (Myerson, 2015). In brief testing, Windows 10 also prefers ephemeral cipher suites when using RDP. The free upgrades, universal deployment strategy, and

[2]although Volatility may also be used for AES key scanning, employing the same technique as bulk_extractor, bulk_extractor was selected as it does not require interpretation and loading of any address space, but treats all inputs as bitstreams

preference for PFS connections make it an ideal candidate system to maximize the impact of the research going forward.

The systems were virtualized atop the open-source VirtualBox platform. VirtualBox supports the required capabilities for testing, namely snapshots and the use of pipes which can be connected between hosts to simulate a serial connection. This support will significantly ease the burden of Kernel debugging in Windows, which requires an external OS attached to the debuggee through a serial connection.

A logical diagram of the analysis infrastructure and list of exact tools with versions can be found in Figure 3.2 and Table 3.1 respectively.

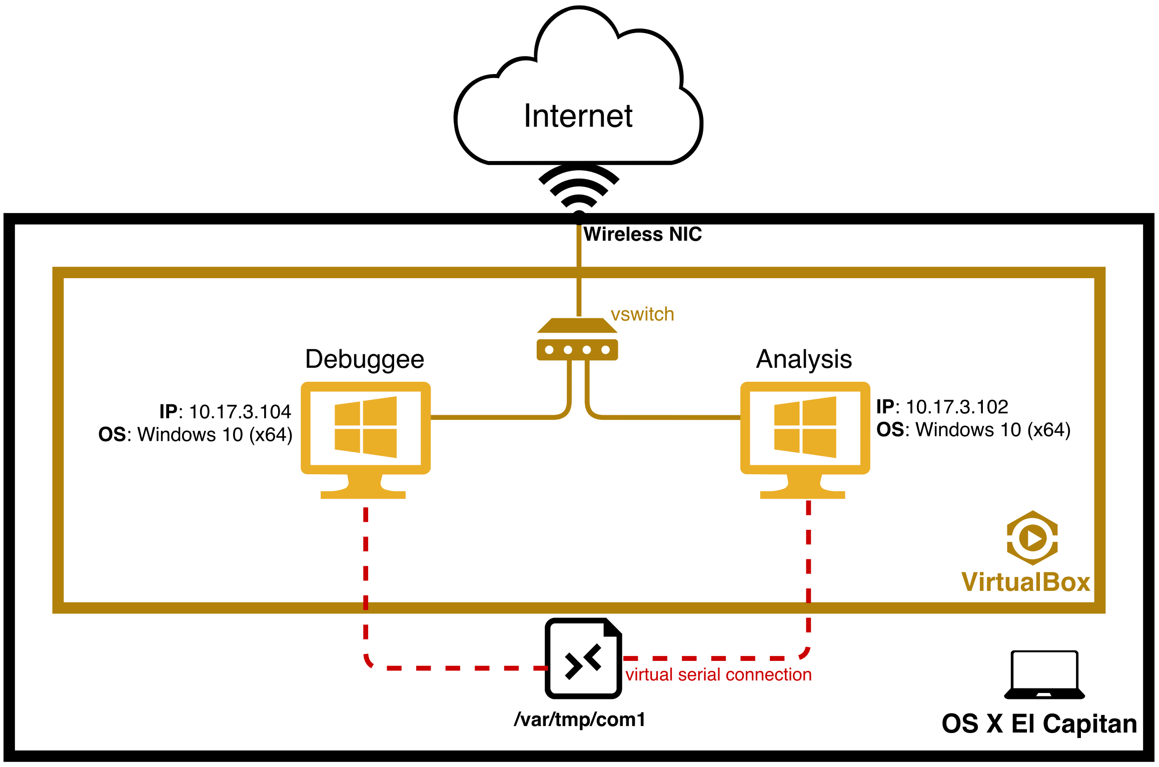


Figure 3.2. Analysis infrastructure logical diagram

Table 3.1. Analysis infrastructure

Tool	Version	Author
VirtualBox	5.0.14	Oracle
Windows 10	10.0.10586	Microsoft
Python	2.7	Python Software Foundation
WinDbg	10.0.10586.567	Microsoft
radare2	w32-0.10.1	pancake@nopcode.org
Volatility	2.5	Volatility Foundation
Rekall	1.4.1	Google
HxD	1.7.7.0	Maël Hörz
cvdump	14.00.23611	Microsoft
bulk_extractor	1.5.2	Simon Garfunkel
Kali Linux	2016.1	Offensive Security
Wireshark	2.0.2	Wireshark Foundation
FreeRDP	2.0.0-dev (git 2a3e999)	The FreeRDP Team
sslkeylog	N/A	Peter Wu

3.0.5 Measure of Success

The defining criterion for success was the implementation of a repeatable, reliable, and possibly automated method for identification and extraction of artifacts that enabled the decryption of arbitrary connections. For connections employing PFS, this would involve extraction of either the pre-master key, master key, or ephemeral private key and association with a unique identifier for a given connection. Even if this goal is not achieved, there are still valuable forensic artifacts that may be extracted, which could constitute contribution to the community. Extraction of public keys, timestamps, and connection identifiers that may outlive or compliment other sources used for enumerating prior connections from memory could be forensically significant.

3.1 Summary

This chapter provided the methodology and technical implementation that will be employed in the research. It discussed the design decisions that led to the choice of methodology and instrumentation, and infused the information gleaned in the literature review to do so. Chapter Four will describe the results achieved through application of this methodology, and any anecdotal findings or deviations from what has been described.

CHAPTER 4. RESULTS & DISCUSSION

A variety of outcomes were obtained through application of the steps discussed in Chapter Three and through adaptations that deviated from the anticipated methodology. This section discusses those outcomes, roughly segmented in alignment with the steps that appeared in the methodology. A brief summary of the most salient findings is as follows:

- Master key structures were identified and mapped to the session ID (but not the client/server random), allowing successful decryption of TLS connections
- Other opaquely documented TLS structures were identified and documented
- Time constants alluded to in the Checkoway paper were identified and explained
- Methodical evolutions that are applicable to future endeavors were developed and employed

For clarity, the remainder of the results will be explained and illustrated in the context of a single client/server memory pair and complimentary packet capture. This facilitates understanding the chain of discovery events and flows more naturally. Similarly, most examples/figures, when applicable, are related through the lens of WinDbg, allowing more consistent and lucid explanations.

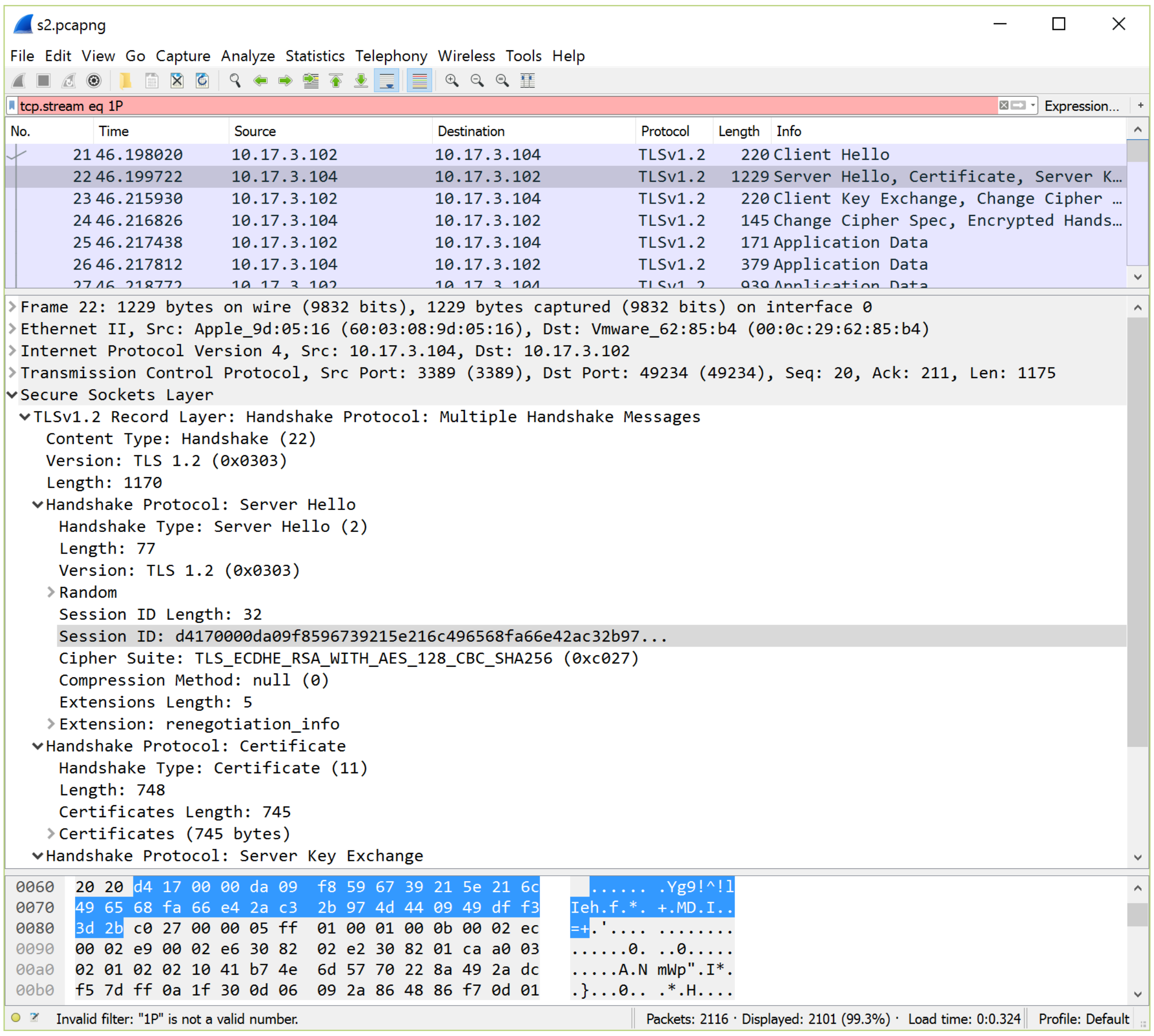
4.1 Staging and Execution

The testing environment was established as shown in Figure 3.2 from Chapter Three. Various RDP connections were made between hosts during the analysis, and the hosts were rebooted periodically. The hosts were allowed access to the Internet, for the sake of broadening the number of possible artifacts that could be found. These additional artifacts result from connections instigated by autonomous tasks Windows engages in like checking for updates.

The focus of the research was on the exploration and development of a technique to extract the artifacts. Future work will focus on rigorous scientific testing of the methods to further validate the provenance and accuracy of the findings. Even so, as will be illustrated, most conclusions are derived not only from direct observation, but from the mechanics that are used to implement TLS in Windows (the DLLs, Functions, and default variables) and the Microsoft's documentation thereof, adding weight beyond simply experimental observation.

Below are a list of sources and connection parameters from one experimental run that will be used to frame the discussion of the rest of the results. These typify the evidentiary items and artifacts used for other experimental runs. The hope is that discussion of these parameters will portray the relationships of artifacts and logic employed to arrive at conclusions more clearly. The data sources analyzed in a given run include those below, and the public values of a single RDP connection established appear in Figure 4.1

- Network capture from client of bidirectional RDP session
- LSASS user-mode process dump of the server
- LSASS user-mode process dump of the client
- VMEM of server directly after LSASS dump creation
- VMEM of client directly after LSASS dump creation


s2.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Expression…

```
tcp.stream eq 1P

No.  Time       Source       Destination  Protocol  Length Info
  21 46.198020  10.17.3.102  10.17.3.104  TLSv1.2    220 Client Hello
  22 46.199722  10.17.3.104  10.17.3.102  TLSv1.2   1229 Server Hello, Certificate, Server K…
  23 46.215930  10.17.3.102  10.17.3.104  TLSv1.2    220 Client Key Exchange, Change Cipher …
  24 46.216826  10.17.3.104  10.17.3.102  TLSv1.2    145 Change Cipher Spec, Encrypted Hands…
  25 46.217438  10.17.3.102  10.17.3.104  TLSv1.2    171 Application Data
  26 46.217812  10.17.3.104  10.17.3.102  TLSv1.2    379 Application Data
  27 46.218772  10.17.3.102  10.17.3.104  TLSv1.2    939 Application Data

> Frame 22: 1229 bytes on wire (9832 bits), 1229 bytes captured (9832 bits) on interface 0
> Ethernet II, Src: Apple_9d:05:16 (60:03:08:9d:05:16), Dst: Vmware_62:85:b4 (00:0c:29:62:85:b4)
> Internet Protocol Version 4, Src: 10.17.3.104, Dst: 10.17.3.102
> Transmission Control Protocol, Src Port: 3389 (3389), Dst Port: 49234 (49234), Seq: 20, Ack: 211, Len: 1175
v Secure Sockets Layer
  v TLSv1.2 Record Layer: Handshake Protocol: Multiple Handshake Messages
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 1170
    v Handshake Protocol: Server Hello
        Handshake Type: Server Hello (2)
        Length: 77
        Version: TLS 1.2 (0x0303)
      > Random
        Session ID Length: 32
        Session ID: d4170000da09f8596739215e216c496568fa66e42ac32b97...
        Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 (0xc027)
        Compression Method: null (0)
        Extensions Length: 5
      > Extension: renegotiation_info
    v Handshake Protocol: Certificate
        Handshake Type: Certificate (11)
        Length: 748
        Certificates Length: 745
      > Certificates (745 bytes)
    v Handshake Protocol: Server Key Exchange

0060  20 20 d4 17 00 00 da 09  f8 59 67 39 21 5e 21 6c   ...... .Yg9!^!l
0070  49 65 68 fa 66 e4 2a c3  2b 97 4d 44 09 49 df f3   Ieh.f.*. +.MD.I..
0080  3d 2b c0 27 00 00 05 ff  01 00 01 00 0b 00 02 ec   =+.'.... ........
0090  00 02 e9 00 02 e6 30 82  02 e2 30 82 01 ca a0 03   ......0. ..0.....
00a0  02 01 02 02 10 41 b7 4e  6d 57 70 22 8a 49 2a dc   .....A.N mWp".I*.
00b0  f5 7d ff 0a 1f 30 0d 06  09 2a 86 48 86 f7 0d 01   .}...0.. .*.H....

Invalid filter: "1P" is not a valid number.   Packets: 2116 · Displayed: 2101 (99.3%) · Load time: 0:0.324   Profile: Default
```

Figure 4.1. Example connection parameters in Wireshark

4.2 Heuristic Scanning for AES keys

Bulk_extractor scans yielded AES keys on both client and server, but not common keys between the two. It was expected that AES keys would be returned due to the aforementioned role that LSASS plays in encrypting secrets for locally authenticated users. Despite no cross-system matches, there were keys on each individual host that appeared multiple times.

The lack of an overt match does not preclude the possibility that the matching AES session keys do reside in memory. It simply means that a valid key schedule was not identified between the two using the algorithm employed by bulk_extractor. It is conceivable that the session key was transferred to the requesting process and then destroyed within the LSASS process. Considering the presence of other key schedules, the lack of a valid schedule for a session keys that does exist seems somewhat unlikely as it would point to an inconsistent implementation. The absence of a match led to the identified keys schedules being inspected manually.

Manual inspection involved viewing context surrounding the keys in a hex editor for possible clues as to whether or not they may relate to an TLS connection. If the keys were allocated in close temporal proximity to other connection parameters, there may be adjacency to those parameters on the heap for instance.

The keys were examined sequentially from the lowest match address to the highest. In the sample above, both AES-128 and AES-256 keys were returned. The scope of the search could have been restricted to only AES-128 keys (based on the cipher suite) for efficiency; however, the limited number of keys meant that examining the additional keys was not burdensome and actually provided a valuable anti-pattern (e.g. what values exist around all AES keys found and are therefore dis-interesting in the context of a search for TLS-specific artifacts).

The examination led to several relevant discoveries. The majority of the first keys were prefixed with the magic dword values "RUUU" and "KSSM" if rendered in ASCII. These numeric values are stored Little Endian, and so when considered as strings they would be "UUUR" and "MSSK" respectively. The two strings appear in the source code for credential extraction tools like Mimikatz in relation to Microsoft's bcrypt module (Delphy, 2016b).

The interesting point, however, was when keys appeared that did not have "RUUU" prefixing them, but maintained a similar structure (including "MSSK"). These keys appeared to have a magic dword with the value of "3lss" or "ssl3" in Big

Endian. This was an immediate indicator that they could be related to SSL. As anticipated earlier, the first key preceded by the “ssl3” tag was located near connection artifacts such as the unicode text “Microsoft SSL Protocol Provider,” what appears to be a session ID, connection parameters, and URLs. Some of this context can be viewed in Figure 4.2.

```
48 80 31 e7 5e 00 00 00-98 5b 24 e7 5e 00 00 00  H.1.^....[$.^...
04 00 00 00 00 c2 02 0c-00 08 00 00 28 c0 00 00  ............(...
18 00 00 00 80 01 00 00-10 00 00 00 10 00 00 00  ................
00 00 00 00 00 00 00 00-01 00 00 00 00 00 00 00  ................
01 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00-b0 e5 2d e7 5e 00 00 00  .........-.^...
20 00 00 00 6f 23 00 00-a0 21 aa c4 8d 15 54 45   ...o#...!....TE
24 c1 45 4e 4e c0 1d 5a-db 30 5d 8d 9d 57 ab 2b  $.ENN..Z.0]..W.+
99 1d d5 97 00 00 00 00-00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00-00 00 00 00 05 00 00 00  ................
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
ff ff ff ff 00 00 00 00-00 00 00 00 00 00 00 00  ................
09 00 00 00 40 00 00 00-40 00 00 00 00 00 00 00  ....@...@.......
4d 00 69 00 63 00 72 00-6f 00 73 00 6f 00 66 00  M.i.c.r.o.s.o.f.
74 00 20 00 53 00 53 00-4c 00 20 00 50 00 72 00  t. .S.S.L. .P.r.
6f 00 74 00 6f 00 63 00-6f 00 6c 00 20 00 50 00  o.t.o.c.o.l. .P.
72 00 6f 00 76 00 69 00-64 00 65 00 72 00 00 00  r.o.v.i.d.e.r...
02 00 00 00 80 02 00 00-7c 02 00 00 00 00 00 00  ........|.......
7c 02 00 00 33 6c 73 73-03 03 00 00 28 c0 00 00  |...3lss....(...
01 00 00 00 30 02 00 00-30 00 00 00 31 de 5f 96  ....0...0...1._.
80 e3 a4 10 fc 78 f1 cf-c5 de e6 dd 7d 63 7b ab  .....x......}c{.
c0 68 c5 b3 cf 1d c4 7b-17 1b 01 ad 60 00 c8 f2  .h.....{....`...
8c d5 3b be 6a 42 49 d0-a7 9f 2c bc 30 02 00 00  ..;.jBI...,.0...
4b 53 53 4d 02 00 01 00-01 00 00 00 10 00 00 00  KSSM............
00 01 00 00 20 00 00 00-4c 2a 57 5a 12 c3 f4 e2  .... ...L*WZ....
15 31 48 4d ff 9f ab 34-aa 54 91 28 cc 9e 6f 5f  .1HM...4.T.(..o_
4e a9 b3 d1 f9 29 8c 8d-00 00 00 00 4c 2a 57 5a  N....)......L*WZ
12 c3 f4 e2 15 31 48 4d-ff 9f ab 34 aa 54 91 28  .....1HM...4.T.(
cc 9e 6f 5f 4e a9 b3 d1-f9 29 8c 8d e8 4e 0a c3  ..o_N....)...N..
fa 8d fe 21 ef bc b6 6c-10 23 1d 58 60 72 35 42  ...!...l.#.X`r5B
ac ec 5a 1d e2 45 e9 cc-1b 6c 65 41 ba 03 89 6c  ..Z..E...leA...l
```

Figure 4.2. ssl3 tag to AES artifact adjacency

Searching for the dword “MSSK” with WinDbg led to hits both in writable space and in a loaded module (as expected for a static magic value). Resolving the nearest symbols to the magic values within the module led to both insight into the meaning of MSSK and a tactic that will be employed repeatedly later. All of the symbols were found in the “bcryptprimitives” module and were related to “MSCrypt” keys. The most interesting symbol name resolved was “bcryptprimitives!validateMSCryptSymmKey.” This was a pivotal revelation for the analysis: assumption about a structure can be verified to some degree by looking for a “validate” functions related to that structure. The validate functions encountered were fairly simple, as can be seen by the disassembly of the “bcryptprimitives!validateMSCryptSymmKey” in Figure 4.3.

```
0:000> uf  bcryptprimitives!validateMSCryptSymmKey
bcryptprimitives!validateMSCryptSymmKey:
00007ffa`436a79ac 33c0            xor     eax,eax
00007ffa`436a79ae 4885c9          test    rcx,rcx
00007ffa`436a79b1 740b            je      bcryptprimitives!validateMSCryptSymmKey+0x12 (00007ffa`436
    a79be)  Branch

bcryptprimitives!validateMSCryptSymmKey+0x7:
00007ffa`436a79b3 8179044b53534d  cmp     dword ptr [rcx+4],4D53534Bh // ascii 'MSSK'
00007ffa`436a79ba 480f45c8        cmovne  rcx,rax

bcryptprimitives!validateMSCryptSymmKey+0x12:
00007ffa`436a79be 488bc1          mov     rax,rcx
00007ffa`436a79c1 c3              ret
```

Figure 4.3. Disassembled validateMSCryptSymmKey function (annotated)

It was inferred from this and other symbol names that MSSK likely stands for “MicroSoft Symmetric Key” or “Microsoft Software Symmetric Key.” This finding is in-line with discovery of the AES keys within the MSSK structure. The AES key is actually repeated twice in close proximity, but bulk_extractor (and, by proxy, findaes) only identified one of the two keys. The reason is that the first value is the actual key, whereas the second “key” exists as part of the (much larger) identified schedule. As mentioned, the “cold-boot attacks” paper discussed the observation that schedule pre-calculation produces a large deterministic block as a time-memory trade-off which, while efficient, also constitutes a security risk by enabling the type of scanning employed here (Halderman et al., 2008).

Performing the same analysis on the ssl3 tag resolved the symbols listed in Figure 4.4, all of which were inside of the “ncryptsslp” (Ncrypt SSL Provider) module. There was no “validate” function for ssl3, but the combination of the symbol names and close proxity to the AES keys provide corroborating circumstantial evidence that 3lss could be a session key structure, or at the very least another structure that requires a key (for instance the HMAC or the key used for session ticket encryption / decryption). Note that the trailing symbols from the “ln” command and miscellaneous hyperlinks have been removed from Figure 4.4 for brevity.

```
0:000> .foreach(match {s -[1]a 00007ffa‘00000000 L?800000000000 3lss}){ln match}
(00007ffa‘3f172400)   ncryptsslp!TlsGenerateSessionKeys+0x251
(00007ffa‘3f172400)   ncryptsslp!TlsGenerateSessionKeys+0x2aa
(00007ffa‘3f173000)   ncryptsslp!SPSslDecryptPacket+0x43
(00007ffa‘3f1735c0)   ncryptsslp!SPSslEncryptPacket+0x43
(00007ffa‘3f173f20)   ncryptsslp!SPSslImportKey+0x19a
(00007ffa‘3f173f20)   ncryptsslp!SPSslImportKey+0x22b
(00007ffa‘3f1743b0)   ncryptsslp!SPSslExportKey+0x76
(00007ffa‘3f1743b0)   ncryptsslp!SPSslExportKey+0x103
(00007ffa‘3f175440)   ncryptsslp!SPSslFreeObject+0x1b
(00007ffa‘3f176b20)   ncryptsslp!guard_dispatch_icall_nop+0x1e07**
(00007ffa‘3f17c8d4)   ncryptsslp!Ssl2GenerateSessionKeys+0x22c
(00007ffa‘3f17c8d4)   ncryptsslp!Ssl2GenerateSessionKeys+0x294

**Missing Symbols for function, nop was closest symbol
```

Figure 4.4. ssl3 resolved symbols

The composition of the ssl3 structure and other SSL structures will be discussed in Section 4.4 of this chapter. Briefly, notice from Figure 4.2 that directly after the ssl3 structure, the values “0x0303” and “0x0c28” appear, which match the public connection parameters listed earlier. The value directly preceding the “ssl3” magic is a dword indicating the size of the structure, which encompasses the MSSK structure and adds weight to the supposition that the keys may in fact be session keys when combined with the symbols, despite the lack of cross-host matches.

Returning to the literature review, it was conceived that, as Schannel is also loaded into the process requesting the SSL/TLS connection, the 3lss structure may still be the session key structure, and may be passed to the application after the key exchange, even though LSASS performs the key isolation for longer-term keys. For this reason, dumps were made of the RDP processes on either side of the connection[1]. This yielded matching 3lss structures, solidifying the circumstantial evidence that 3lss is the session key structure. The actual structure with the ’3lss’ tag was different in these processes than in LSASS, however. As anticipated, more than one matching key was found between hosts, presumably for a client write and server write key. Supporting this is a value which appears to be a flag for read or write that is set oppositely for matching keys on each side of the connection.

4.3 Public Connection Values

Scanning with the unique public connection values did return hits, but did not immediately lead to a direct correlation with a key. The session ID and the client/server random values both appeared in multiple locations in memory across samples, but did not appear reliably linked to a key in the immediate context. As will be discussed later, the session ID was eventually linked to the master key.

The client and server random values were seen to be stored concatenated together, with the client random preceding the server random value. A loose context for the random values was developed based on the hash suite list, possibly

[1]mstsc.exe on the client-side and the svchost process that contains the RDP Services on the server-side of the connection

stored in the same structure or allocated at the same time. An example of this relationship can be seen in Figure 4.5.

The hash suite list appears to be stored in multiple structures, so use of this list to find the random automatically would be more complex. Misses can be seen in the output depicted in the figure. In x64 Windows 10 systems, the random values appear to begin approximately 0xD2 bytes above the first entry in some of the hash suite lists.

There were also non-aligned instances of individual random values, where it was stored in close proximity to the hash suite list, appearing alongside what could be buffered application data. No references to or from either nebulous hash suite list "structures" were identified, which meant that this was a dead end for further analysis during the time-frame available.

```
0:000> .foreach(hashSuite {s -[1w]b 0x0 L?800000000000 04 01 05 01 02 01 04 03 05 03 02 03 02 02}){db
    ${hashSuite}-D2 L40;.echo **}
0000005e`e67623c0  56 ba 52 74 9b 13 80 53-02 dc f5 71 ca a5 0d 64  V.Rt...S...q...d
0000005e`e67623d0  6a de d1 e5 d4 c7 30 46-98 88 83 90 b9 1b 50 b3  j.....0F......P.
0000005e`e67623e0  56 ba 7c a4 b9 e0 c4 87-76 10 ee 27 be b3 33 11  V.|.....v..'..3.
0000005e`e67623f0  cd 78 23 0d 4a 84 f8 51-d4 a8 28 1e 69 07 32 a6  .x#.J..Q..(.i.2.
**
0000005e`e6767790  56 ba 52 74 41 ca a9 aa-d0 11 e0 7e e0 b7 67 02  V.RtA......~..g.
0000005e`e67677a0  81 24 26 c0 c9 db 2d 2b-9d 0b 0d 34 04 bd ea 92  .$&...-+...4....
0000005e`e67677b0  56 ba 7c a4 83 81 c0 60-a3 5f 3a 6d b3 4d c8 c8  V.|....`._:m.M..
0000005e`e67677c0  10 07 41 ff e5 ce ce 17-a0 25 76 4f 9f 20 e1 20  ..A......%vO. .
**
0000005e`e678a248  56 ba 7d 0c 3b 9a 75 54-02 e7 c1 5a c7 d6 d5 59  V.}.;.uT...Z...Y
0000005e`e678a258  72 5d f4 12 98 b5 c5 5f-8c e1 6b 74 a5 6a 25 44  r]......_..kt.j%D
0000005e`e678a268  56 ba 52 dc ce 0f fb 20-61 5d 40 cc 27 2d 33 8e  V.R.... a]@.'-3.
0000005e`e678a278  1d c3 63 4d b7 ba 60 3b-89 ba 69 d3 2f d8 89 52  ..cM..`;..i./..R
**
0000005e`e678aed8  56 ba 7c a7 f0 84 1d 9f-49 51 4a 61 41 58 97 bc  V.|.....IQJaAX..
0000005e`e678aee8  40 a3 53 79 91 4d 8a cb-d1 c7 00 8a 77 f6 f5 c6  @.Sy.M......w...
0000005e`e678aef8  56 ba 52 77 81 5f df 8d-95 4c 9f 6a d0 47 62 f7  V.Rw._...L.j.Gb.
0000005e`e678af08  5a 2e 91 4f 5a 87 04 a0-6a 55 0a 01 20 a3 ba 11  Z..OZ...jU.. ...
**
0000005e`e7245ca8  56 ba 7c ad 4b fb f3 07-29 00 30 b4 da 42 37 c5  V.|.K...).0..B7.
0000005e`e7245cb8  ba 8b 58 40 e0 9b d9 e7-93 ae 92 2c b2 49 0d 8d  ..X@.......,.I..
0000005e`e7245cc8  56 ba 52 7d d6 76 9f 9b-d2 52 ce 18 64 e1 47 be  V.R}.v...R..d.G.
0000005e`e7245cd8  9a 85 c4 cf 38 35 06 68-8f fb 89 58 f6 e6 ce 35  ....85.h...X...5
**
0000005e`e72483f0  eb 00 00 00 02 00 00 00-00 00 02 00 00 00 00 00  ................
0000005e`e7248400  eb 00 00 00 00 00 00 00-16 03 03 00 e6 01 00 00  ................
0000005e`e7248410  e2 03 03 56 ba 7d 0c 3b-9a 75 54 02 e7 c1 5a c7  ...V.}.;.uT...Z.
0000005e`e7248420  d6 d5 59 72 5d f4 12 98-b5 c5 5f 8c e1 6b 74 a5  ..Yr]......_..kt.
**
0000005e`e7258838  00 00 00 00 00 00 08 00-00 00 63 e6 5e 00 00 00  ..........c.^...
0000005e`e7258848  66 77 04 80 a1 ff ff ff-16 03 03 00 e6 01 00 00  fw..............
0000005e`e7258858  e2 03 03 56 ba 7d 0c 3b-9a 75 54 02 e7 c1 5a c7  ...V.}.;.uT...Z.
0000005e`e7258868  d6 d5 59 72 5d f4 12 98-b5 c5 5f 8c e1 6b 74 a5  ..Yr]......_..kt.
```

Figure 4.5. Client/Server random values in memory

Scanning with the common public values like TLS version and cipher suite identifier led to the discovery of another structure, which was later confirmed to be the master key structure. When the ssl3 tag was initially recognized, it seemed as though it might be a reference to SSL v3, which is sometimes conflated with TLS. This theory, however, changed rapidly upon scanning for other instances of the TLS version number, which led to the discovery of the magic value “ssl5” (the master key).

4.4 Identified SSL Structures

After discovery of the ssl3 and ssl5 tags, all loaded modules were scanned for other appearances of the ASCII “ssl” appended with a number. This led to the identification of “ssl1” through “ssl7” inclusive, all of which were found in the ncryptsslp module. These values were then used to scan writable memory for instantiations, which yielded results for the majority of the magic values. Before going on to explore those instances, the magic values in the module were symbolically resolved to gain better insight into the meaning and utility of the identified instances.

It was at this point that the “validate” functions came back into play; the ncryptsslp symbols were filtered to display those symbols that contain the phrase “validate” and the results can be seen in Figure 4.6. All of these functions shared a common flow when disassembled:

1. accept a handle (always a pointer in this case)
2. check that the first dword is a particular size value
3. check that the second dword is a specific magic value

This explicitly ties a given magic to a particular purpose/concept in the TLS paradigm; further, it dictates the exact size of each structure and that the first two fields of each are a size followed by a magic. The disassembled “ncryptsslp!SslpValidateMasterKeyHandle” function that illustrates this flow can be seen in Figure 4.7 and the magic values are mapped to the Validate function in Table 4.1.

```
0:000> x /2 ncryptsslp!*validate*
00007ffa‘3f17b558 ncryptsslp!SslpValidateEphemeralHandle
00007ffa‘3f17b5b8 ncryptsslp!SslpValidateMasterKeyHandle
00007ffa‘3f1717ec ncryptsslp!SslpValidateProvHandle
00007ffa‘3f17b578 ncryptsslp!SslpValidateHashHandle
00007ffa‘3f17b598 ncryptsslp!SslpValidateKeyPairHandle
```

Figure 4.6. Ncryptsslp “Validate” function symbols

```
0:000> uf ncryptsslp!SslpValidateMasterKeyHandle
ncryptsslp!SslpValidateMasterKeyHandle:
00007ffa‘3f17b5b8 4885c9          test    rcx,rcx
00007ffa‘3f17b5bb 7412            je      ncryptsslp!SslpValidateMasterKeyHandle+0x17 (00007ffa‘3
    f17b5cf)  Branch

ncryptsslp!SslpValidateMasterKeyHandle+0x5:
00007ffa‘3f17b5bd 833950          cmp     dword ptr [rcx],50h
00007ffa‘3f17b5c0 720d            jb      ncryptsslp!SslpValidateMasterKeyHandle+0x17 (00007ffa‘3
    f17b5cf)  Branch

ncryptsslp!SslpValidateMasterKeyHandle+0xa:
00007ffa‘3f17b5c2 817904356c7373  cmp     dword ptr [rcx+4],73736C35h
00007ffa‘3f17b5c9 7504            jne     ncryptsslp!SslpValidateMasterKeyHandle+0x17 (00007ffa‘3
    f17b5cf)  Branch

ncryptsslp!SslpValidateMasterKeyHandle+0x13:
00007ffa‘3f17b5cb 488bc1          mov     rax,rcx
00007ffa‘3f17b5ce c3              ret

ncryptsslp!SslpValidateMasterKeyHandle+0x17:
00007ffa‘3f17b5cf 33c0            xor     eax,eax
00007ffa‘3f17b5d1 c3              ret
```

Figure 4.7. Disassembled ValidateMasterKey function (x64)

The ssl3 and ssl7 magic values did not map to a “validate” function. The supposed purpose of ssl3 was discussed previously in Section 4.2, and is likely the session key structure. The ssl7 magic value symbolically resolved in three functions:

Table 4.1. Ncryptsslp magic values to function mapping

SSL Magic	Size (x86)	Size (x64)	Function
ssl1	0xE4	0x130	SslpValidateProvHandle
ssl2	0x24	0x30	SslpValidateHashHandle
ssl3	?	?	< *none* >
ssl4	0x18	0x20	SslpValidateKeyPairHandle
ssl5	0x48	0x50	SslpValidateMasterKeyHandle
ssl6	0x18	0x20	SslpValidateEphemeralHandle
ssl7	?	?	< *none* >

"ncryptsslp!SPSslGeneratePreMasterKey," "ncryptsslp!SPSslGenerateMasterKey," and "ncryptsslp!TlsDecryptMasterKey." It also appeared in a fourth function for which no symbols existed. Because it was not identified as regularly instantiated during the various runs of the development phase, and because the master key structure did regularly appear, this was not pursued within the scope of the analysis.

After enumerating the numbered SSL magics, scanning was performed on the "C:\Windows\System32\" directory where the majority of operating system DLLs reside. This was done with a rudimentary "findstr" command, simply to see if other (non-loaded) libraries referenced these magic values. The only match returned on the Windows 10 analysis machine was the ncryptsslp.dll file.

With the knowledge gained from the validate functions about the intended size and purpose of the structures, the actual instantiations in memory were analyzed to enumerate the members of those structures. The focus was on the "keys," beginning with the master key structure (ssl5) and including the presumed session key structure (ssl3), the "KeyPair" structure (ssl4), and the "Ephemeral" structure (ssl6).

The most salient, and therefore most explored, of these was the master key. This structure was eventually leveraged to decrypt connections. Analysis performed on the ssl3, ssl4, and ssl6 structures, was not as thorough or conclusive, but the fruits of that analysis are also provided here as a basis for future work.

The master key instantiations were identified and dumped via the WinDbg command in Figure 4.8. Using this method, every identified structure was displayed separated by a series of asterisks, with the length determined by the size field (allowing the same command to be portable between x86 and x64 systems). Only one example instantiation was included for demonstration of the output. All of the resulting instances were then compared for similarities and known values.

```
0:000> .foreach(sslMK {s -[1w]d 0x0 L?800000000000 'ssl5'}){db ${sslMK}-4 Ldwo(${sslMK}-4);.echo
    ************}
0000005e`e72d2500  50 00 00 00 35 6c 73 73-03 03 00 00 00 00 00 00  P...5lss........
0000005e`e72d2510  10 18 18 3f fa 7f 00 00-01 00 00 00 35 41 87 dd  ...?........5A..
0000005e`e72d2520  50 81 b2 18 5d b7 ff 7e-c9 db be a8 55 15 99 24  P...]..~....U..$
0000005e`e72d2530  a7 cb 8a 3d bf 33 6c 0a-a9 57 19 bb 2b 51 3a 36  ...=.3l..W..+Q:6
0000005e`e72d2540  c7 7a 3a 9e e5 04 00 39-cd 05 a0 90 00 00 00 00  .z:....9........
************
<...>
```

Figure 4.8. SSL master key instance in memory (x64)

The TLS version was found stored as a dword next to the magic values. This was verified through experimentation with the TLS version number during connections. Specifically, by changing the supported server TLS version, one could articulate the negotiated version for a connection. The version is changed by modifying registry values located within "HKLM SYSTEM\CurrentControlSet\Control\SecurityProviders\SCHANNEL" under an eponymous subkey, which must be manually created for each SSL or TLS version desired (Microsoft, 2015e).

If the master key structure were to be represented as a C struct using Hungarian Notation, as is used in much of the native code accessible on MSDN, it would appear similar to Figure 4.9.

```
typedef struct _SSL5_Struct {
    ULONG cbLength,          // The count in bytes (cb), of the structure
    ULONG dwMagic,           // a dword (dw) of the ASCII value 'ssl5' [stored as '5lss']
    ULONG dwProtocol,        // One of the CNG SSL Provider Protocol Identifier values (TLS Version)
    ULONG dwUnknown1,        // non-existent in x86 -> padding?
    PVOID pvCipherSuite,     // a pointer to an ncryptsslp!CipherSuiteList entry
    ULONG bIsClient,       // boolean value - 0 for server, 1 for client
    UCHAR[48] MasterKey,     // the 48-byte master key
    ULONG dwUnknown2         // always 0 -> reserved?
} SSL5_Struct, *PSSL5_Struct;
```

Figure 4.9. SSL master key (ssl5) C data structure

Viewable in the comments within Figure 4.9, there are two unknown members. Speculatively, the first unknown may have any number of explainations, including:

- This could be padding for alignment, though the structure appears to be a packed structure.
- The protocol field may actually be a quadword (8 bytes) in x64 instead of a dword. This seems unnecessary considering the type of value being stored
- It could be a dword only existing in x64, but this seems to be the least likely of the three and the value remains zero across the limited samples observed.

The second unknown appears at the very end and was always observed to be zero.

There was one pointer within the structure, which pointed to a list of cipher suites inside of the ncryptsslp binary. This list contained entries composed of the numeric cipher suite identifier (e.g. 0xc028) and a series of pointers to Unicode strings the described the elements of the cipher suite. The first pointer in the structure is to the full cipher suite name (e.g. TLS_ECDHE_RSA_AES_256_CBC_SHA384). This can be seen in Figure 4.10

```
0:000> dc 0000005e‘e73544d0 L2
0000005e‘e73544d0  00000050 73736c35                   P...5lss
0:000> dps 0000005e‘e73544d0 Ldwo(0000005e‘e73544d0)/8
0000005e‘e73544d0  73736c35‘00000050
0000005e‘e73544d8  00000000‘00000303
0000005e‘e73544e0  00007ffa‘3f181810 ncryptsslp!CipherSuiteList+0x1300
0000005e‘e73544e8  2993e86c‘00000001
0000005e‘e73544f0  5d4a9b02‘4e8401f2
0000005e‘e73544f8  6bddd58d‘eb766f24
0000005e‘e7354500  ba53633b‘d4cdb1b9
0000005e‘e7354508  2d3b1142‘255cd666
0000005e‘e7354510  01a6a81c‘9af5559e
0000005e‘e7354518  00000000‘73f84d9e
0:000> dpu poi(0000005e‘e73544d0+10) L2
00007ffa‘3f181810  0000c028‘00000c00
00007ffa‘3f181818  00007ffa‘3f181fc0 "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384"
```

Figure 4.10. Dereference of CipherSuiteList entry from ssl5

The ssl3 structure maintained certain similarities with the master key structure. There are substantially more unknown variables and assumptions, but the first three members are the same, and the fourth known member serves the same purpose (identifying the cipher suite). Figure 4.11 provides the annotated structure as observed, but the variables were not directly manipulated to assist in identification, so in most cases speculations are noted.

Interestingly, only AES keys were identified in this structure when valid, likely because it is the preferred symmetric algorithm broadly in use. If more time was available, attempts would be made to generate ssl3 structures with 3DES or RC4 to observe differences.

```
typedef struct _SSL3_Struct {
    ULONG cbLength,         // the count in bytes (cb), of the structure (usually 0x027C on x64)
    ULONG dwMagic,          // a dword (dw) of the ASCII value 'ssl3' [stored as '3lss']
    ULONG dwProtocol,       // One of the CNG SSL Provider Protocol Identifier values (TLS Version)
    ULONG dwCipherSuite,    // numeric cipher suite identifier
    ULONG dwUnknown1        // boolean value -- read or write key?
    ULONG cbSymmKey         // this value observed to match the size value for MSSK
    ULONG cbHashLength,     // the size of the ensuing hash, based on MAC algo
    UCHAR[48] HashData,     // fixed field - if preceding length is not 48 bytes, then 0 padded
    MSSK_Struct SymmKey     // the associated MSSK Structure
} SSL3_Struct, *PSSL3_Struct;

typedef struct _MSSK_Struct {
    ULONG cbLength,         // the count in bytes (cb), of the structure (usually 0x0230 on x64)
    ULONG dwKeyMagic,       // 'KSSM' -> MS SK -> MicroSoft Symmetric Key
    ULONG dwUnknown2,       // usually 0x02000100 -> 0x00010002 -> NCRYPT_SCHANNEL_INTERFACE?
    ULONG dwUnknown3,       // Typically observed as 1
    ULONG dwKeyBitLen,      // the length in bits of the AES key, usually 0x100 or 0x80
    ULONG cbKeyLength,      // the count in bytes of the AES key (compliments below field)
    UCHAR[32] AesKey,       // the AES key
    ULONG dwUnknown4,       // always 0 -> padding?
    UCHAR[448] KeySchedule, // fixed length - the AES Key Schedule or state for each round, 0 padded
    ULONG dwUnknown5,       // Half of the length, equivalent to N_state * N_rounds
    ULONG cbScheduleLen,    // Overall size of KeySchedule member
    UCHAR[16] Unknown6      // Possibly an IV? this seems likely
} MSSK_Struct, *PMSSK_Struct;
```

Figure 4.11. LSASS SSL session key (ssl3) data structure

As discussed in the literature review, the key schedule size is fixed for each AES key size (176, 208, or 240 bytes). So, for an AES 256-bit key, the schedule is 240 bytes long; however, the length of the expanded key sequence that bulk_extractor identified is seemingly 448 bytes long for AES 256 and 320 for AES 128 keys, zero padded in the case of the latter. Incidentally, this was observed to be 32 bytes multiplied by the number of rounds, which would equate to twice the size of the state for each round. The first $N_b(N_r + 1) * 4$ bytes were determined to match the key schedule by passing the identified key to an AES implementation and printing the key schedule. This intuitively is why bulk_extractor identified the keys in the first place. Still, this leaves 208 bytes remaining in the case of AES 256. Of note is that the last 16 bytes (N_{state} is always 128 bits) are always the first 16 bytes of the key (and by extension the first 16 bytes of the key schedule).

It was also observed that two dwords typically followed the key schedule, both of which were related to size. The first dword containing the size of the $N_{state}(N_{rounds})$ and the second was double that value, coinciding with the size of the full "KeySchedule" blob.

As noted earlier, the session key structure (3lss) was also found in the requesting process memory, RDP in this case. This structure was similar to the one found in LSASS, but varied in several obvious ways.

- The first is that the cipher suite ID was replaced with a pointer to the ncryptsslp!CipherSuiteList entry that the cipher suite ID appears in.
- The second was that the structure appeared far less sparse, with pointers that referenced addresses within its own size value.
- The third was the the "RUUU" magic value appeared inside ssl3 and pointed to the MSSK structure directly below it, also inside ssl3

An example of the first 0x100 bytes of one of these entries can be seen in Figure 4.12. Following that, the perceived structure, based on limited testing, is depicted in the Volatility V-Type format in Figure 4.13.

```
0:000> .foreach(sessK {s -[1w]a 0 L?800000000000 3lss}){.echo **;db ${sessK}-4 L100}
**
000001ef‘b9d1a1e0  2e 0d 00 00 33 6c 73 73-03 03 00 00 00 00 00 00  ....3lss........
000001ef‘b9d1a1f0  90 19 57 d8 fc 7f 00 00-01 00 00 00 00 00 00 00  ..W.............
000001ef‘b9d1a200  50 a2 d1 b9 ef 01 00 00-00 00 00 00 00 00 00 00  P...............
000001ef‘b9d1a210  00 00 00 00 00 00 00 00-ff 1c 62 2b 00 00 00 00  ..........b+....
000001ef‘b9d1a220  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
000001ef‘b9d1a230  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
000001ef‘b9d1a240  00 00 00 00 00 00 00 00-04 00 00 00 00 00 00 00  ................
000001ef‘b9d1a250  be 0c 00 00 52 55 55 55-20 f4 c8 bb ef 01 00 00  ....RUUU .......
000001ef‘b9d1a260  70 a2 d1 b9 ef 01 00 00-00 00 00 00 00 00 00 00  p...............
000001ef‘b9d1a270  80 0c 00 00 4b 53 53 4d-02 00 01 00 05 00 00 00  ....KSSM........
000001ef‘b9d1a280  10 00 00 00 01 00 00 00-00 01 00 00 01 00 00 00  ................
000001ef‘b9d1a290  30 b1 e5 bb ef 01 00 00-00 00 00 00 00 00 00 00  0...............
000001ef‘b9d1a2a0  00 00 00 00 00 00 00 00-20 00 00 00 b9 f0 65 ef  ........ .....e.
000001ef‘b9d1a2b0  0a 27 33 62 0d 92 3d 2a-1e ba 24 3b 9a 1d 94 a8  .’3b..=*..$;....
000001ef‘b9d1a2c0  70 d4 b5 ab 08 18 d6 f8-d8 04 1d 07 00 00 00 00  p...............
000001ef‘b9d1a2d0  b9 f0 65 ef 0a 27 33 62-0d 92 3d 2a 1e ba 24 3b  ..e..’3b..=*..$;
**
<...>
0:000> dps 000001ef‘b9d1a1e0 L5
000001ef‘b9d1a1e0  73736c33‘00000d2e
000001ef‘b9d1a1e8  00000000‘00000303
000001ef‘b9d1a1f0  00007ffc‘d8571990 ncryptsslp!CipherSuiteList+0x1400
000001ef‘b9d1a1f8  00000000‘00000001
000001ef‘b9d1a200  000001ef‘b9d1a250
```

Figure 4.12. Non-LSASS SSL session key (ssl3) data structure

```
'_SSL_SESSION_KEY': [ 0x0d2e, {
    'Length': [0x0, ['unsigned long']],
    'Magic': [0x4, ['unsigned long']],
    'Protocol': [0x8, ['unsigned char']],
    'CipherSuite': [0x10, ['pointer64', ['_SSL_CIPHER_SUITE_LIST_ENTRY']]],
    'IsWriteKey': [0x18, ['Enumeration', dict(target = 'unsigned long', choices={0: False, 1: True})
        ]],
    'BcryptKey': [0x20, ['pointer64', ['void']]],
    // RUUU Bcrypt Key struct pointed to by pointer
    // MSSK struct pointed to by RUUU
}]
```

Figure 4.13. Non-LSASS SSL session key (ssl3) V-Type data structure (x64)

The final key structures to be discussed in this section are ssl4 and ssl6. These two structures contain identical members, which is logical when considering that ssl6 appears to be the ephemeral version of ssl4 (the Public/Private key pair). In the case of a cipher suite like TLS_ECDHE_RSA_AES_256_CBC_SHA384, both the ssl4 and ssl6 structures should exist, as an ephemeral key pair is created for the key exchange and the persistent key is used for signing. Because the two are so similar, they will be discussed simultaneously with deviations contrasted as appropriate. The manifested C structures can be seen in Figure 4.14.

```
typedef struct _SSL4_Struct {
    ULONG cbLength,         // The count in bytes (cb), of the structure
    ULONG dwMagic,          // a dword (dw) of the ascii value 'ssl6' [stored as '6lss']
    ULONG dwUnknown1,       // usually 0x01000300 -> 0x00030001
    ULONG dwUnknown2,       // always 0 -> padding?
    PVOID pvKspProvider,     //
    PVOID pvKspKey          //
} SSL4_Struct, *PSSL4_Struct;

typedef struct _SSL6_Struct {
    ULONG cbLength,         // The count in bytes (cb), of the structure
    ULONG dwMagic,          // a dword (dw) of the ascii value 'ssl6' [stored as '6lss']
    ULONG dwUnknown1,       // usually 0x0003000a or  0x00030007
    ULONG dwUnknown2,       // always 0 -> padding?
    PVOID pvKspProvider,     //
    PVOID pvKspKey          //
} SSL6_Struct, *PSSL6_Struct;
```

Figure 4.14. SSL ephemeral key (ssl6) and key pair (ssl4) C data structures

The two pointers in the structure both reference an interstitial structure, containing a header and series of pointers. The author named “KspProvider” pointer eventually points to the Unicode string “ncryptsslp.dll” and the binary of that module.

The second “KspKey” pointer is so named because the interstitial structure it references then itself points to another data structure with the magic value “KPSK,” or “KSPK” in Big Endian. Following the same procedures employed previously, it was found that all KSPK references occur inside of “ncryptprov.dll,” and a validate function exists named “KspValidateKeyHandle.” Microsoft’s CNG documentation suggests that “KSP” refers to “Key Storage Provider” (Microsoft, 2014b), which seems to be logical in this context.

```
0:000> .foreach(srvK {s -[w1]a 0 L?80000000000000 4lss}){.echo **ServKey**;dpp ${srvK}-4 L4}
**ServKey**
0000005e‘e725e840  73736c34‘00000020
0000005e‘e725e848  00000000‘00030001
0000005e‘e725e850  0000005e‘e72141a0 00000003‘44440001
0000005e‘e725e858  0000005e‘e725d710 00000000‘44440002
**ServKey**
0000005e‘e72cf9c0  73736c34‘00000020
0000005e‘e72cf9c8  00000000‘00030001
0000005e‘e72cf9d0  0000005e‘e72141a0 00000003‘44440001
0000005e‘e72cf9d8  0000005e‘e72f8e50 00000000‘44440002
0:000> dpp 0000005e‘e72f8e50 L4
0000005e‘e72f8e50  00000000‘44440002
0000005e‘e72f8e58  0000005e‘e72141a0 00000003‘44440001
0000005e‘e72f8e60  0000005e‘e720bd20 4b53504b‘00000208
0000005e‘e72f8e68  0000005e‘e72f8e70 00650053‘00530054
0:000> dc 0000005e‘e720bd20 L2
0000005e‘e720bd20  00000208 4b53504b                      ....KPSK
0:000> dpu 0000005e‘e720bd20 Ldwo(0000005e‘e720bd20)/$ptrsize
0000005e‘e720bd20  4b53504b‘00000208
0000005e‘e720bd28  0000005e‘e72cfaa0 "TSSecKeySet1"
0000005e‘e720bd30  0000005e‘e72f4cb0 "f686aace6942fb7f7ceb231212eef4a4_f928a10b-2557-4456-b0e"
0000005e‘e720bd38  00000800‘00000001
0000005e‘e720bd40  00000000‘00000001
0000005e‘e720bd48  00000001‘00ffffff
0000005e‘e720bd50  0000005e‘e72fb920 "Microsoft Strong Cryptographic Provider"
0000005e‘e720bd58  00007ffa‘366e6a90 ".      "
0000005e‘e720bd60  0000005e‘e72cfbc0 "C:\ProgramData"
<...>
```

Figure 4.15. SSL key pair (ssl4) provider Dereference

The author was able to further identify the encrypted private key blob from this structure by taking the known key file from disk for RDP and comparing it to a DPAPI blob that one of the pointers referenced (at offset 0xd0 on 64-bit systems and 0x7c on 32-bit systems in KPSK structures that were pointed to by ssl4 structures. This is only mentioned anecdotally, as only RDP was tested, so its possible that other private keys may appear elsewhere in the structure. An example of the first part of the DPAPI blob (matching what was on disk) can be seen in

Figure 4.16. The equivalent was not true of KPSK structures pointed to by ssl6 (the ephemeral key pair).

```
0:000> .foreach(ephemK {s -[1w]a 0 L?80000000000000 4lss}){.echo **;db poi(poi(poi(${ephemK}-4+8+(2*
    $ptrsize))+(2*$ptrsize))+d0) L100}
**
0000005e`e678ec70  01 00 00 00 d0 8c 9d df-01 15 d1 11 8c 7a 00 c0  .............z..
0000005e`e678ec80  4f c2 97 eb 01 00 00 00-81 7e 1f 3d 16 c8 58 46  O........~.=..XF
0000005e`e678ec90  83 4b 08 20 12 5a 67 94-04 00 00 00 2c 00 00 00  .K. .Zg.....,...
0000005e`e678eca0  43 00 72 00 79 00 70 00-74 00 6f 00 41 00 50 00  C.r.y.p.t.o.A.P.
0000005e`e678ecb0  49 00 20 00 50 00 72 00-69 00 76 00 61 00 74 00  I. .P.r.i.v.a.t.
0000005e`e678ecc0  65 00 20 00 4b 00 65 00-79 00 00 00 10 66 00 00  e. .K.e.y....f..
0000005e`e678ecd0  00 01 00 00 20 00 00 00-ac 89 d9 c9 56 ce e4 d8  .... .......V...
0000005e`e678ece0  41 fd 32 4a 1d 29 61 77-64 d4 51 10 e5 e1 23 fe  A.2J.)awd.Q...#.
0000005e`e678ecf0  38 78 30 c3 d8 95 7e 2a-00 00 00 0e 80 00 00  8x0...~*........
0000005e`e678ed00  00 02 00 00 20 00 00 00-b3 8e 99 58 cd 4c 1e 55  .... ......X.L.U
0000005e`e678ed10  d8 aa 2c b9 24 6b e8 c1-19 a5 ab f4 33 49 be 69  ..,.$k......3I.i
0000005e`e678ed20  20 11 09 98 9d 37 68 67-50 05 00 00 c5 df e0 93   ....7hgP.......
0000005e`e678ed30  9d 28 65 31 ee 8d 18 eb-d4 da f0 04 13 ed 81 c5  .(e1............
0000005e`e678ed40  c1 bb d2 ca d8 1b 39 35-4c 4c 0d 20 df d3 f6 a0  ......95LL. ....
0000005e`e678ed50  48 54 2a 67 fc a5 11 66-c5 18 f5 96 b9 c3 b5 0a  HT*g...f........
0000005e`e678ed60  88 7b e2 de ca 83 32 36-78 09 e4 29 61 43 07 de  .{....26x..)aC..
**
0000005e`e678e600  01 00 00 00 d0 8c 9d df-01 15 d1 11 8c 7a 00 c0  .............z..
0000005e`e678e610  4f c2 97 eb 01 00 00 00-81 7e 1f 3d 16 c8 58 46  O........~.=..XF
0000005e`e678e620  83 4b 08 20 12 5a 67 94-04 00 00 00 2c 00 00 00  .K. .Zg.....,...
0000005e`e678e630  43 00 72 00 79 00 70 00-74 00 6f 00 41 00 50 00  C.r.y.p.t.o.A.P.
0000005e`e678e640  49 00 20 00 50 00 72 00-69 00 76 00 61 00 74 00  I. .P.r.i.v.a.t.
0000005e`e678e650  65 00 20 00 4b 00 65 00-79 00 00 00 10 66 00 00  e. .K.e.y....f..
0000005e`e678e660  00 01 00 00 20 00 00 00-ac 89 d9 c9 56 ce e4 d8  .... .......V...
0000005e`e678e670  41 fd 32 4a 1d 29 61 77-64 d4 51 10 e5 e1 23 fe  A.2J.)awd.Q...#.
0000005e`e678e680  38 78 30 c3 d8 95 7e 2a-00 00 00 0e 80 00 00  8x0...~*........
0000005e`e678e690  00 02 00 00 20 00 00 00-b3 8e 99 58 cd 4c 1e 55  .... ......X.L.U
0000005e`e678e6a0  d8 aa 2c b9 24 6b e8 c1-19 a5 ab f4 33 49 be 69  ..,.$k......3I.i
```

Figure 4.16. Dereferenced KSPK private key DPAPI Blob

4.5 Confirming the master key

In order to verify that the 5lss structure was indeed the master key, a quick test was performed. Briefly, the process was as follows:

1. Wireshark was started on Kali Linux
2. xfreerdp was launched to connect to the target through Peter Wu's sslkeylog bash wrapper
3. the library pre-loaded by the script transparently dumped the pre-master key to a file
4. a process dump of LSASS was created on the target
5. the process dump was searched with HxD for the pre-master key from the file

This did confirm that the master key was in memory, and that it was stored in the ssl5 structure. This step was only performed once, and is not necessary to validate findings. It is included here solely as a demonstration of another approach for verification.

4.6 Mapping master keys to session IDs

The analysis up until this point had not yielded a master key paired to a unique public value for a given session, which is required to decrypt the session. The master key structure only had a single pointer, which did not provide an avenue to one of these unique values. For this reason, the approach was taken to inversely walk the pointers by scanning for references to instantiated master key memory addresses.

Scanning for pointers to master key structures was successful and led to identification of a new interstitial structure in a different module. Specifically, every master key was pointed to by a single structure with the Magic value "BDDD."

Leveraging the methods employed previously, BDDD was noted to appear in the Ncrypt module, and a "validate" function for this structure was identified: the "ncrypt!ValidateSslKeyHandle" function. Two other validate functions existed, one for the Hash (CDDD) and the other for the Provider (ADDD).

The BDDD structure will be referred to as the “NcryptSslKey” structure for distinction. The NcryptSslKey structure bears resemblance to the ssl4 and ssl6 structures, constituted mainly by the size, magic, and two pointers. The first pointer is to the key structure, and the second is to an NcryptSslProvider (ADDD).

Armed with this information, the BDDD magic was used to scan memory for additional key instances. This endeavor yielded several new NcryptSslKey instances in addition to those found pointing to the master key structures. Dereferencing the key pointer in all of the identified NcryptSslKeys led to the output depicted in Figure 4.17, which clearly shows ssl4, ssl5, and ssl6 key structures being referenced.

```
0:000> .foreach(nKey {s -[1w]a 0 L?800000000000 BDDD}){db poi(${nKey}-4+10) L10;}
0000005e‘e72d2500  50 00 00 00 35 6c 73 73-03 03 00 00 00 00 00 00  P...5lss........
0000005e‘e73544d0  50 00 00 00 35 6c 73 73-03 03 00 00 00 00 00 00  P...5lss........
0000005e‘e725e840  20 00 00 00 34 6c 73 73-01 00 03 00 00 00 00 00   ...4lss........
0000005e‘e72fa520  50 00 00 00 35 6c 73 73-03 03 00 00 00 00 00 00  P...5lss........
0000005e‘e72cf580  20 00 00 00 36 6c 73 73-0a 00 03 00 00 00 00 00   ...6lss........
0000005e‘e72fbba0  50 00 00 00 35 6c 73 73-03 03 00 00 00 00 00 00  P...5lss........
0000005e‘e72fb790  50 00 00 00 35 6c 73 73-03 03 00 00 00 00 00 00  P...5lss........
0000005e‘e72cfd60  20 00 00 00 36 6c 73 73-0a 00 03 00 00 00 00 00   ...6lss........
0000005e‘e72cf9c0  20 00 00 00 34 6c 73 73-01 00 03 00 00 00 00 00   ...4lss........
0000005e‘e72fd770  50 00 00 00 35 6c 73 73-03 03 00 00 00 00 00 00  P...5lss........
0000005e‘e73551f0  50 00 00 00 35 6c 73 73-03 03 00 00 00 00 00 00  P...5lss........
```

Figure 4.17. NcryptSslKey instances Dereferenced (x64)

After exhausting the two pointers in the NcryptSslKey structure, another pointer scan was performed with the addresses of the NcryptSslKeys associated with master keys. This consistently led to at least one, but sometimes more than one, pointer to those NcryptSslKeys instances. The context around these pointers was inspected, which did not yield any magic values but did lead to consistency denoting a managed structure and an important finding – a unique public value.

Values heuristically matching the session ID pattern were spotted reliable at 0x70 bytes below one pointer to every NcryptSslKey that pointed to a master key. As discussed in the Checkoway paper, each of these values had two sets of zeros in the third and fourth position due to a peculiarity in the session ID generation mechanism employed by Windows (Checkoway et al., 2014). One of these suspected IDs matched the session ID from the test connection, confirming this speculation. Figure 4.18 shows an example of mapping master keys to session IDs by pointer scanning, and then dumping the session ID. Extraneous pointers are filtered on the loose fingerprint of two non-zero bytes followed by two zero bytes in the first dword of the possible session ID.

```
0:000> .foreach(sslSess {.foreach (BDDDPoi {.foreach(ssl5Key {s -[1w]d 0x0 L?800000000000 'ssl5'}){s
    -[1]q 0x0 L?800000000000 ${ssl5Key}-4;}}){s -[1]q 0x0 L?80000000000 ${BDDDPoi}-10}}){.if (dwo(${
    sslSess}+78) < 0x00010000 & dwo(${sslSess}+78) >= 0x00000101){db ${sslSess}+78 L20;.echo ***}}
0000005e`e72e5c08  78 1f 00 00 f3 50 8a 72-39 87 ba 3b cb 15 fb 48  x....P.r9..;...H
0000005e`e72e5c18  a8 1d 89 61 d1 94 b0 15-eb 64 b5 4f 05 a0 ad 3d  ...a.....d.O...=
***
0000005e`e72e5ac8  29 2b 00 00 c6 6f 97 9b-a0 f8 eb 44 26 01 79 f3  )+...o.....D&.y.
0000005e`e72e5ad8  82 f4 66 cc ae ac 81 d5-d6 e2 97 e5 11 90 bf 27  ..f...........'
***
0000005e`e72ffd78  d4 17 00 00 da 09 f8 59-67 39 21 5e 21 6c 49 65  .......Yg9!^!lIe
0000005e`e72ffd88  68 fa 66 e4 2a c3 2b 97-4d 44 09 49 df f3 3d 2b  h.f.*.+.MD.I..=+
***
0000005e`e67fe1d8  b9 3c 00 00 a1 10 69 0b-4a e9 11 1b ce 57 25 c6  .<....i.J....W%.
0000005e`e67fe1e8  c4 7a 03 7b 3c 39 c4 9c-75 ce 51 e1 c2 eb 79 ee  .z.{<9..u.Q...y.
***
0000005e`e72fe508  43 2a 00 00 bf 4f 62 2f-0f c1 19 97 4a 0e f3 0c  C*...Ob/....J...
0000005e`e72fe518  d8 38 c3 a0 25 b8 3a bb-dc db ce 7b 23 25 d2 d9  .8..%.:....{#%..
***
0000005e`e72e5fc8  b0 17 00 00 9c c4 95 44-a9 0d 12 97 a9 e3 c8 10  .......D........
0000005e`e72e5fd8  7c d4 2f 3a 82 0b 6f f7-6d 62 25 45 26 93 df f1  |./:..o.mb%E&...
***
0000005e`e72e5988  6f 23 00 00 a0 21 aa c4-8d 15 54 45 24 c1 45 4e  o#...!....TE$.EN
0000005e`e72e5998  4e c0 1d 5a db 30 5d 8d-9d 57 ab 2b 99 1d d5 97  N..Z.0]..W.+....
***
```

Figure 4.18. Master key to session ID mapping

The pointer addresses that did have seemingly valid session IDs below them were then explored and compared to identify a possible structure. This showed that, consistently, a pointer appeared directly before the NcrypSslKey pointer that referenced an offset into the Schannel module. The remainder of the analysis regarding this new structure is discussed in the ensuing section.

4.7 SSL Session Cache Items

Symbolically resolving the memory address of the pointer above the reference to the master key revealed that it was the "vftable" for a class inside of the schannel module. In fact, it was vftable symbols of two different classes: "CSslCacheClientItem" and "CSslCacheServerItem." These names follow the internal Microsoft use of Hungarian Notation by prefixing classes with a capital "C" (Microsoft, n.d.-b, n.d.-g). The class names themselves are descriptive, revealing both whether the host was the client or server of a given connection, and that the structures identified are in fact part of a cache, likely maintained to enable session resumption.

"Vftable" is a reference to the Virtual Function Table, sometimes also abbreviated "vtable." It appears that in C++ binaries compiled with the MSVC compiler, virtual function tables are often the first item stored for a class instantiation in memory (Microsoft, n.d.-i). This means that one could parse PDBs for vftable entries, find classes of interest, and use the vftable symbol for the class to scan for instantiations of the class as a method for reversing.

Following the process employed when discovering the NcryptSslKey, scanning was performed for the SslCacheServerItem and SslCacheClient Item to enumerate any instantiations not yet identified. This did not return additional results, giving a strong indicator that a master key may exist for every SslCache item. This creates an opportunity to scan for the class, dump the session ID and simply deference the pointers to the master key, a process depicted in Figure 4.19. The added context of "Client" or "Server" is echoed in the output as well, and only two entries for each are shown for brevity, with missing entries being replaced by an ellipses.

```
0:000> .foreach(cacheSess {s -[w1]q 0x0 L?800000000000 schannel!CSessionCacheServerItem::‘vftable’}){.
    echo **SERVER*********;.echo **SessID**;db ${cacheSess}+88 L20;.echo **MasterKey**;db poi(poi(${
    cacheSess}+10)+10)+1C L30};.foreach(cacheSess {s -[w1]q 0x0 L?800000000000 schannel!
    CSessionCacheClientItem::‘vftable’}){.echo **CLIENT*********;.echo **SessID**;db ${cacheSess}+88
    L20;.echo **MasterKey**;db poi(poi(${cacheSess}+10)+10)+1C L30}
**SERVER*********
**SessID**
0000005e‘e67fe1d8  b9 3c 00 00 a1 10 69 0b-4a e9 11 1b ce 57 25 c6  .<....i.J....W%.
0000005e‘e67fe1e8  c4 7a 03 7b 3c 39 c4 9c-75 ce 51 e1 c2 eb 79 ee  .z.{<9..u.Q...y.
**MasterKey**
0000005e‘e72fbbbc  bc 28 46 79 99 b9 9f d3-fd f3 a2 46 42 c5 d9 3b  .(Fy.......FB..;
0000005e‘e72fbbcc  9a b4 3e 51 62 7f 6e 01-45 ef 12 0b a9 8a 1c 32  ..>Qb.n.E......2
0000005e‘e72fbbdc  23 f3 db e0 15 4e 30 d7-86 9b db 7a b6 6f 53 18  #....N0....z.oS.
<...>
**SERVER*********
**SessID**
0000005e‘e72ffd78  d4 17 00 00 da 09 f8 59-67 39 21 5e 21 6c 49 65  .......Yg9!^!lIe
0000005e‘e72ffd88  68 fa 66 e4 2a c3 2b 97-4d 44 09 49 df f3 3d 2b  h.f.*.+.MD.I..=+
**MasterKey**
0000005e‘e72fb7ac  44 b5 03 be f7 84 2e a9-a4 16 fb f8 b6 3b 93 2b  D............;.+
0000005e‘e72fb7bc  23 b7 b6 87 fb f5 29 7b-25 3e ac 42 78 77 c8 e1  #.....){%>.Bxw..
0000005e‘e72fb7cc  15 95 e1 4c 3f 00 c4 0b-f2 a0 f4 68 8d e0 b7 aa  ...L?......h....
**CLIENT*********
**SessID**
0000005e‘e72e5988  6f 23 00 00 a0 21 aa c4-8d 15 54 45 24 c1 45 4e  o#...!....TE$.EN
0000005e‘e72e5998  4e c0 1d 5a db 30 5d 8d-9d 57 ab 2b 99 1d d5 97  N..Z.0]..W.+....
**MasterKey**
0000005e‘e735520c  8b c9 e9 df 65 3e 3c bf-53 3b e8 4c 68 97 78 7b  ....e><.S;.Lh.x{
0000005e‘e735521c  d4 53 b8 ce e9 d5 38 9e-9c 36 59 eb f9 97 d9 c8  .S....8..6Y.....
0000005e‘e735522c  d0 66 6a ad ca 5b e2 25-8f 30 b9 25 12 15 a7 17  .fj..[.%.0.%....
**CLIENT*********
**SessID**
0000005e‘e72e5ac8  29 2b 00 00 c6 6f 97 9b-a0 f8 eb 44 26 01 79 f3  )+...o.....D&.y.
0000005e‘e72e5ad8  82 f4 66 cc ae ac 81 d5-d6 e2 97 e5 11 90 bf 27  ..f...........’
**MasterKey**
0000005e‘e72fa53c  8a 74 2d 06 53 29 d5 c6-e1 92 c8 52 db d9 d1 4c  .t-.S).....R...L
0000005e‘e72fa54c  83 fe 6d 87 1d c1 9d 34-a4 e6 3a ea ea db 4b c8  ..m....4..:...K.
0000005e‘e72fa55c  9f ec c9 72 60 d4 48 55-0c de 15 43 4a d8 90 9d  ...r‘.HU...CJ...
<...>
```

Figure 4.19. Schannel CSslCacheItem parsing (x64)

Being able to scan once and then reference the pointers forward is far more efficient than scanning multiple times to walk the pointers inversely, however, it relies on symbols. The symbols, in this case, simply provide an offset from the base of the image to the address where the vftable can be found (in the .rdata section of Schannel). The difficulty is that this offset cannot be statically referenced due to the mutability of the vftable location, which can change with every version of Schannel. The Schannel dll may be arbitrarily updated for any number of reasons, causing variations even within a given Windows release. The implications of this for the main memory analysis suites will be discussed in a later section.

The SslCacheServerItem and the SslCacheClientItem differ in length and composition, as expected. One interesting artifact of the SslCacheClientItem is a pointer to the public certificate for the connection that appears directly after the pointer to the master key, providing additional context about the other end of the connection. There is also a pointer to the Unicode server name, though this value can also be empty. The total size of each class, while technically unknown, has been roughly estimated through comparison of many instantiations across samples to be approximately 0x140 bytes.

Time constraints and the the ability to map the master key to the session ID meant that full enumeration of the members of these classes was left for future work. The members that were identified are shown in Figure 4.20. Because there are a large number of unknowns, Figure 4.20 depicts the known values in the Volatility VType format, which provides a structure length, then members with an offset into the structure and a data type.

```
'_SSL_SESSION_CACHE_CLIENT_ITEM': [ 0x140, {
    'ClassVftable': [0x0, ['pointer64', ['void']]],
    'MasterKey': [0x10, ['pointer64', ['void']]],
    'PublicCertificate': [0x18, ['pointer64', ['void']]],
    'PublicKey': [0x28, ['pointer64', ['void']]],
    'NcryptSslProv': [0x60, ['pointer64', ['void']]],
    'SessionID': [0x88, ['array', 0x20, ['unsigned char']]],
    'ServerName': [0xf8, ['pointer64', ['void']]],
    'SessionTicket': [0x128, ['pointer64', ['void']]]
    'CSessCacheManager': [0x110, ['pointer64', ['void']]]
}]

'_SSL_SESSION_CACHE_SERVER_ITEM': [ 0x140, {
    'ClassVftable': [0x0, ['pointer64', ['void']]],
    'MasterKey': [0x10, ['pointer64', ['void']]],
    'NcryptSslProv': [0x60, ['pointer64', ['void']]],
    'SessionID': [0x88, ['array', 0x20, ['unsigned char']]],
    'CSslCredential': [0xf0, ['pointer64', ['void']]]
}]
```

Figure 4.20. Schannel CSslCacheItem V-Types (x64)

One point of note is that the cache may not always contain a session ID, but may instead reference a session ticket if it is a client cache. This was first noticed during a brief test that was not originally going to be included in the thesis. Skype was installed in a Windows 10 target to identify whether or not it leveraged the SSL Cache. This may sound odd considering Skype is a Microsoft product, but Skype was an acquisition, not developed internally, and so logically may not have used Schannel.

This led to very interesting results: Skype, when launched, did leverage the cache, but entries appeared for the CSslCacheClientItem which had a null session ID. This is because those sessions used Ticket-based session resumption. A pointer to the ticket seems to be located at 0x128 into the structure on x64 systems examined. This was then verified through examination of the "schannel!CSessionCacheClientItem::SetSessionTicket" function, which uses this address.

An unexpected discrepancy was noted amongst the ticket-based cache items: multiples of the same ticket existed for different cache entries, but mapped to different master key values. Intuitively, this is not possible for successful cache operation; the ticket is meant to be unique, and contains the master key as part of the state within the ticket. By virtue of a sound encryption scheme, different masker keys would necessitate a different ticket, as a single bit changed in the plaintext state would cause a massive change in the ciphertext.

Wireshark can decrypt sessions that employ session tickets, but it does so by mapping them to the client random. Because the session cache does not appear to maintain a relationship with the client random (as, after all, it used the ticket for a unique identifier and it caches the master key that was derived with the client random), this means that manual decryption of items using session tickets is currently required. It would be feasible, though outside the scope of this thesis, to programmatically enable this by extending Wireshark through its scripting engine.

4.8 Schannel Classes

Extrapolating from the use of the vftable symbol as a class marker, the Schannel symbols were searched for other vftable symbols to enumerate classes, which yielded several classes of interest. There were two entries containing the word "Key": "CSslServerKey" and "CSslEphemKeyData." Noting the similarity to 4lss and 6lss respectively, this was investigated.

The first entry in CSslServerKey after the vftable pointer was a pointer to an NcryptSslKey that in turn pointed to a 4lss struct. Similarly, the first entry in CSslEphemData after the vftable pointed to an NcryptSslKey that pointed to a 6lss struct.

Another set that were interesting were those that used the term "Credential." Two classes with this in the title had interesting names: "CSslCredential" and "CCredentialGroup," both of which had instantiations that were memory resident. A pointer to CSslCredential was found within the CSslCacheServerItem structure. A pointer to CCredentialGroup was found inside of another identified class: "CSchannelTelemetryContext." This telemetry class was found to also contain the session ID (when session tickets were not employed), but it did not reliably exist across samples. In addition to the session ID, this class was observed to contain the TLS version, cipher suite, and the Server Name Indicator, amongst other values. It seems, purely speculatively, that it may exist for active connections, though this is truly anecdotal.

The "CSchannelTelemetryContext" class, when present, also pointed to either "CSchannelClientTelemetryContext" or "CSchannelServerTelemetryContext."

4.9 Schannel Parameters

While looking at the Schannel binary, several strings of interest were found in the .rdata section. Nominally, the following three string contained the text "cache," bearing possible relation to the classes discovered previously: "ClientCacheTime," "ServerCacheTime," and "MaximumCacheSize."

As the strings are resident in the so-named "read-only data" section of the binary, it seemed that they may not be variables to be modified, but possibly used as a comparator against something like a registry key (e.g. as an existence check of a regisrty value that overrides the default value). The strings were also reminiscent of the "hard-coded" values alluded to by Checkoway and others in their Dual EC paper. They specifically note that "SChannel caches ephemeral keys for two hours (this timeout is hard-coded in the configurations we examined)" (Checkoway et al., 2014, p. 8) and referenced a value of 20,000 when discussing the generation of session IDs (Checkoway et al., 2014, p. 9).

Searching MSDN revealed that these values are indeed registry settings, with default values that can be overridden (Microsoft, 2015e). A section defining the ClientCacheTime setting contains the following excerpt:

> The first time a client connects to a server through the Schannel SSP, a full TLS handshake is performed. When this is complete, the master secret, cipher suite, and certificates are stored in the session cache on the respective client and server. (Microsoft, 2015e, p. 1)

This statement corroborates the independent findings of the thesis.

Table 4.2 contains the default values for the Client/Server cache time, per Microsoft's documentation (Microsoft, 2003b, 2015e):

Table 4.2. Client and server SSL cache time

OS Version	ClientCacheTime	ServerCacheTime
Windows 8.1	10 Hours	10 Hours
Windows Server 2012 R2	10 Hours	10 Hours
Windows 8	10 Hours	10 Hours
Windows Server 2012	10 Hours	10 Hours
Windows 7	10 Hours	10 Hours
Windows Server 2008 R2	10 Hours	10 Hours
Windows Vista	10 Hours	10 Hours
Windows XP	10 Hours	10 Hours
Windows 2000 SP2	10 Hours	10 Hours
Windows 2000 SP1	2 Minutes	2 Minutes
Windows NT 4.0 SP6a + Q265369	1 Hour	5 Minutes
Windows NT 4.0 SP6a	2 Minutes	2 Minutes

These default values can also be found in the .rdata section of the binary. Searching for the dword 0x00004e20 (20,000) in Schannel matched exactly on the symbol “schannel!CSslGlobals::m_dwMaximumEntries.” Likewise, searching for the dword 0x02255100 (10 hours in milliseconds) returned exact matches on the symbols “schannel!CSslGlobals::m_dwSessionTicketLifespan,” “schannel!CSslGlobals::m_dwServerLifespan,” and “schannel!CSslGlobals::m_dwClientLifespan.”

The outcome is that a captured connection created using Schannel, even if it uses an ephemeral key exchange, can be decrypted by plundering the cache entry of either party up to 10 hours after the connection was initiated, for up to 20,000 entries. It also means that the loose fingerprint present for Schannel-generated session IDs can be altered by creating the registry key that overrides the cache size and setting it to something other than 20,000.

4.10 Scanning Physical Memory

After exhausting those master key entries within the LSASS process space, scanning was then performed on the physical address space with bulk_extractor’s lightgrep scanner. Interestingly, about twice as many seemingly valid SSL structures appeared. About half of these were duplicates of known master keys, but some were unique to those found within LSASS. Those not resident in the LSASS process space (i.e. the other structures) were not aligned on 32-bit or 64-bit boundaries (respective of the system), whereas those that were identified in the LSASS process space were byte aligned. Volatility was leveraged to scan all other processes in the same memory image for instances of the ssl5 struct, but did not return the results that were identified with bulk_extractor from physical memory.

4.11 Automating Extraction

While the WinDbg one-liner is a semi-automation solution, it isn’t can’t be applied across other memory samples without significant alteration to the sample. To that end, the Volatility and Rekall frameworks were leveraged to make the PoC flexible. Rekall was initialy favoured for development because of its philosophical approach and embrace of symbols; however, during some preliminary examinations, Rekall would not parse the test Windows 10 VMEM files correctly, whereas Volatiliy did. This could be due to some peculiarity of the testers environment, however, the latest release version and master branch from the projects git repository were tested on Windows, Linux, and Mac environments to the same effect.

Volatility was extended through a plug-in that dumps the session IDs and master keys into the format that Wireshark accepts, allowing a single file to be imported to decrypt packet captures containing sessions that were in the cache. Because Volatility does not currently support the dynamic downloading and extraction of symbols in the either the release version or master branch from its git repository, a different algorithm was leveraged to extract cache items. Specifically, because each cache entry maintains a one-to-one relationship with an NcryptSslKey structure, which maintains a one-to-one relationship with a master key structure, one can walk those structures backwards. This is inefficient, however, because it involves scanning the entire address space three times instead of one time, followed by two dereferences.

For efficiency, the plugins scans for NcryptSslKey and checks if they point to master keys, and then scans for pointers to the NcryptSslKey that do. This saves an extra iteration of scanning the entire address space. The address space itself is limited by only scanning VADs that have read-write permissions. This was chosen as a precautionary middle ground between only scanning the heaps and scanning the full address space, effectively checking the writable address space within the process.

The real issue is detecting the vftable class identifier without symbols, which was addressed successfully but imperfectly. In order to identify a valid cache entry, the identified pointer to the NcryptSslKey is instantiated as a cache item, and then the pointer that should reference the vftable is tested. It is tested to see if it points to Schannel's .rdata section, which would rule out any invalid address, but does not rule out. Currently, this has been completely Using rekall's method of scanning for the RSDS signature[2], downloading the appropriate symbol file, and locating the exact address would alleviate this concern.

The next issue is determining whether or not a cache entry is a server cache item or a client cache item. This is done currently by checking a flag present within the master key structure that, in testing, was always zero for server cache entries and one for client cache entries. This too is imperfect, but has proven functional in all tests. As the client and server caches that employ session IDs store them at the same offset, this only becomes problematic when looking for session tickets or other structure specific information.

The final issue is how to handle cache items with session tickets instead of session IDs. This was accomplished by adding the capability to recognize valid cache items that don't have a session ID but do have a session ticket pointer and enable printing them as debugging information, but not printing them by default. If it is found that a relationship does persist for the client random and the session tickets, then this could be extended to print in the format that Wireshark expects, as the session ID entries do. One could also extend Wireshark to handle this through its scripting engine, as has been pointed out earlier.

The result is a plug-in that successfully identifies and extracts the required information from the cache to decrypt a session with Wireshark when employing session IDs. Items that use session tickets can be recognized and decrypted with Wireshark more manually, but require a packet capture containing the client random value. The plug-in is named "LSASSLKey," and the result of running the plug-in can be seen in Figure 4.21.

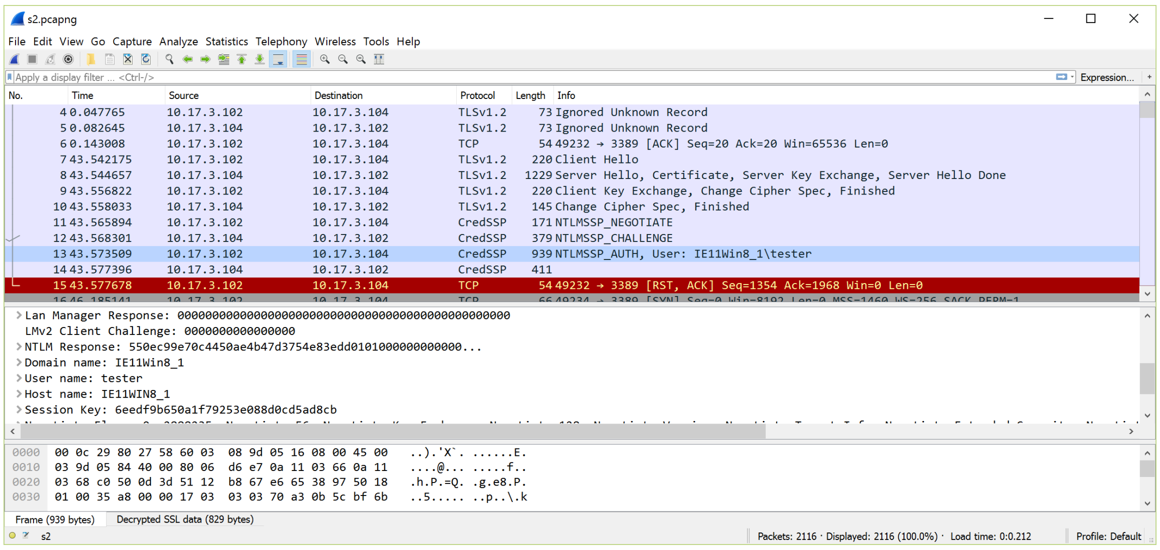
[2]The RSDS is a unique GUID that changes every time the binary is compiled, when this feature is enabled, as it is by default on all Microsoft binaries

```
> vol.exe --plugins=./plugins --profile=Win10x64 -f Win10-Test-c2a4a77d.vmem lsasslkey
Volatility Foundation Volatility Framework 2.5
RSA Session-ID:b93c0000a110690b4ae9111bce5725c6c47a037b3c39c49c75ce51e1c2eb79ee Master-Key:
    bc28467999b99fd3fdf3a24642c5d93b9ab43e51627f6e0145ef120ba98a1c3223f3dbe0154e30d7869bdb7ab66f5318
RSA Session-ID:173300000f84a86aebb2c5de0af20e6d5c2cab95ab65043e14c6e19cee54ee17 Master-Key:9
    dd750e12e6e4439b08326d4a1f9eba2d2fe65c2a26c2088e7cec22ce1d91e9f219b704547a2b2eccb9a81d557d5ae1a
RSA Session-ID:3c2c000024b8f70dd2613d8b13d0c4ac4daaefbe53ab4b7cb9763e80feccb4f1 Master-Key:2
    d119c64695ffc9c143c136471f5625d8cde92d35721f5f2849b92639603799a45e1e601786cbf89b00c186969d44983
RSA Session-ID:d4170000da09f8596739215e216c496568fa66e42ac32b974d440949dff33d2b Master-Key:44
    b503bef7842ea9a416fbf8b63b932b23b7b687fbf5297b253eac427877c8e11595e14c3f00c40bf2a0f4688de0b7aa
RSA Session-ID:432a0000bf4f622f0fc119974a0ef30cd838c3a025b83abbdcdbce7b2325d2d9 Master-Key:552699
    d61e21d1b871af4b05a54003bf03eade60666dd1e54b94c3b5ec98f296db4ae99baed4e23882175e5ffd88be31
RSA Session-ID:6f230000a021aac48d15544524c1454e4ec01d5adb305d8d9d57ab2b991dd597 Master-Key:8
    bc9e9df653e3cbf533be84c6897787bd453b8cee9d5389e9c3659ebf997d9c8d0666aadca5be2258f30b9251215a717
```

Figure 4.21. Volatility LSASSLKey plug-in output

4.12 Decrypting a TLS session

Decrypting a TLS connection was now trivial, and simply involved directing Wireshark to use the output from the Volatility plugin, and opening the associated packet capture. The decrypted RDP session can be seen in Figure 4.22


s2.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...

```
No.  Time          Source        Destination   Protocol  Length Info
   4 0.047765      10.17.3.102   10.17.3.104   TLSv1.2       73 Ignored Unknown Record
   5 0.082645      10.17.3.104   10.17.3.102   TLSv1.2       73 Ignored Unknown Record
   6 0.143008      10.17.3.102   10.17.3.104   TCP           54 49232 → 3389 [ACK] Seq=20 Ack=20 Win=65536 Len=0
   7 43.542175     10.17.3.102   10.17.3.104   TLSv1.2      220 Client Hello
   8 43.544657     10.17.3.104   10.17.3.102   TLSv1.2     1229 Server Hello, Certificate, Server Key Exchange, Server Hello Done
   9 43.556822     10.17.3.102   10.17.3.104   TLSv1.2      220 Client Key Exchange, Change Cipher Spec, Finished
  10 43.558033     10.17.3.104   10.17.3.102   TLSv1.2      145 Change Cipher Spec, Finished
  11 43.565894     10.17.3.102   10.17.3.104   CredSSP      171 NTLMSSP_NEGOTIATE
  12 43.568301     10.17.3.104   10.17.3.102   CredSSP      379 NTLMSSP_CHALLENGE
  13 43.573509     10.17.3.102   10.17.3.104   CredSSP      939 NTLMSSP_AUTH, User: IE11Win8_1\tester
  14 43.577396     10.17.3.104   10.17.3.102   CredSSP      411
  15 43.577678     10.17.3.102   10.17.3.104   TCP           54 49232 → 3389 [RST, ACK] Seq=1354 Ack=1968 Win=0 Len=0
  16 46.185141     10.17.3.102   10.17.3.104   TCP           66 49234 → 3389 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
```

```
> Lan Manager Response: 000000000000000000000000000000000000000000000000
  LMv2 Client Challenge: 0000000000000000
> NTLM Response: 550ec99e70c4450ae4b47d3754e83edd0101000000000000...
> Domain name: IE11Win8_1
> User name: tester
> Host name: IE11WIN8_1
> Session Key: 6eedf9b650a1f79253e088d0cd5ad8cb
```

```
0000  00 0c 29 80 27 58 60 03  08 9d 05 16 08 00 45 00   ..).'X`. ......E.
0010  03 9d 05 84 40 00 80 06  d6 e7 0a 11 03 66 0a 11   ....@... .....f..
0020  03 68 c0 50 0d 3d 51 12  b8 67 e6 65 38 97 50 18   .h.P.=Q. .g.e8.P.
0030  01 00 35 a8 00 00 17 03  03 03 70 a3 0b 5c bf 6b   ..5..... ..p..\.k
```

Frame (939 bytes)   Decrypted SSL data (829 bytes)

s2   Packets: 2116 · Displayed: 2116 (100.0%) · Load time: 0:0.212   Profile: Default

Figure 4.22. Decrypted RDP session

CHAPTER 5. CONCLUSIONS

This thesis answered the question of whether connection parameters exist in the memory of modern Windows systems to retroactively decrypt sessions, affirming that the requisite parameters do exist. A process for reliably extracted the master key paired with a unique connection parameter was developed. This process was then implemented in a technical solution atop one of the community accepted memory forensic frameworks. The author achieved this result by systematically identifying and reverse engineering data structures per the outlined methodology, with several deviations that will likely be incorporated into future work. Finally, ancillary findings about the tools used, and other related structures were identified, which may also contribute to future work.

5.1 Summary of Outcomes

A summary of the outcomes of the thesis is as follows:

- Cached TLS/SSL master keys were identified and related to the session ID or session ticket
- An automated method for extracting these artifacts into a form compatible with Wireshark was developed as a PoC atop of the Volatility framework
- Ancillary structures that provided additional context about the connection, including the public certificates, cipher suite were identified
- Methods that leverage symbol parsing, the vftable for object scanning, and validation functions / error codes were discussed to improve upon the original methodology
- Other structures, such as the session key structure and the client-server random values, were identified, but not linked contextually to specific sessions

5.2 Contributions

Prior to this research there was no publicly available or discussed mechanism for extracting master keys and private keys from main memory of Windows hosts, and no mechanism existed for decrypting ephemeral connections retroactively on Windows hosts. The author has contributed both a method that is generalizable across Windows Vista to Windows 10 which enables retroactive decryption of ephemeral connections and provided a PoC implementation of this method through one of the main memory analysis frameworks, Volatility. This implementation addresses the use cases documented in the Significance section of Chapter One.

In so doing, the author also documented otherwise undocumented structures used by Schannel and the cryptographic libraries for managing key material and session related artifacts. This information can be leveraged by other analysts to develop further research and contribute to the field.

The implementation developed is easy to use and dumps output to a Wireshark compatible format for decryption by default. This is designed to be approachable for practitioners such as LEOs who may have had limited technical training or exposure.

The cache items can live up to 10 hours by default, which may also outlive the current connection identification methods employed in Volatility and Rekall. There may also be 20,000 entries by default for both the client and server caches, which could provide a wealth of information previously not accessed. Session Tickets were also noted to live 10 hours by default, which indicates that the key to decrypt them should live on the server for at least 10 hours as well.

The methods described enable linking the TLS version and cipher suite to unique connection identifiers. Tying cipher suites and protocol versions to connections can help identify exploitative or anomalous connections. For example, SSL downgrade attacks like “POODLE” and weak cipher suite parameters like those used in “LogJam” and “FREAK” would be apparent when observing the caches.

The public certificate and server name indicator (SNI) seems to exist in the client cache for connections that are made to virtual hosts sharing a single IP address. Authors of the MiTLS website, which includes members of the Microsoft Research team and is dedicated to identifying and mitigating attacks against SSL and TLS, note that "Because of the popularity of cloud hosting and content delivery networks, it is increasingly common for webservers to serve several websites on the same IP address" (miTLS, n.d., p. 1). Other current OS-based connection identification mechanisms that exist within Volatility only operate at Layer 4 of the OSI model and below. This means that they cannot currently make the distinction between multiple virtual hosts co-located at the same IP address when observing connections.

The presence of an entry in the server cache of a host that is supposed to be a client could be an indicator of compromise, and this is almost guaranteed to live for the full 10 hours of the cache default. The information in the server cache of a client would also indicate what type of connection was served – for example, the RDP server component uses a specific signing key pair with the identifier "TSSecKeySet1," a reference to "Terminal Services." For the client cache, in addition to the information available from the public certificate, the Session ID can also possibly give information on whether the host on the other end of the connection was a Windows server.

Finally, the methods employed do not rely on proprietary functions to extract credentials, meaning that the extraction is analysis environment agnostic. This enables the possibility of transparently decrypting connections retroactively through the use of virtual machines and memory introspection. One could transparently monitor connections inside of a Windows virtual machine, when those connections leverage Schannel.

5.3 Anecdotes

- Leveraging Microsoft's provided code to execute "BCryptEnumContextFunctions" and return cipher suites in order of preference, all examined systems returned ephemeral suites first, contrary to the cited Schannel documentation.
- Previous research in the literature review focused either heavily or solely on the process itself and not on the underlying data structures. The methodology employed in this research took the reverse approach, which seems to have illuminated overlooked, but valuable structures like the "KPSK" Key storage structure that points to the private key, which other research indirectly touched via some of the Microsoft cryptographic functions.
- The author noted that, in the limited samples observed, a symbol for an "CSchannelTelemetryContext" existed in Windows 10, which did not exist in symbols for the Schannel versions observed in Windows 8 and Windows 7, though these versions were admittedly delimited from the study.
- An open-source python AES implementation was briefly employed to compare the key schedule to the fixed-length 448 byte field in the "MSSK" structure, verifying that, in the case of AES 256, the first 240 bytes match the key schedule, but the remaining bytes are unaccounted.

5.4 Future Work

Given these findings, there are still more questions to be answered and work to be done. This technique developed is self-validating in that, if it fails or is erroneous, it will not correctly decrypt the session. Still, it should be more rigorously evaluated by other practitioners to identify any flaws or acknowledged gaps in the implementation and meet the standards required of a forensic tool. A list of future work the author has identified follows:

- Identify any relationship between the client random values and the master key in order to process session tickets in a way that is suitable for use with Wireshark
- Set up a rigorous testing paradigm for evaluating the current method thoroughly across platforms and cipher suites to identify edge cases
- Examine the functions identified as related to the numbered SSL magic values to gain further insight about the ssl7 structure
- Write a Rekall plug-in that leverages symbols for efficiency and accuracy over the current method
- Further explore extracting private keys from memory alone using DPAPI-NG
- Add functionality to the plug-in to scan all processes with Schannel loaded for session key structures
- Identify ticket encryption scheme and add functionality to the LSASSLKey plug-in to decrypt session tickets

LIST OF REFERENCES

LIST OF REFERENCES

Bhargavan, K., Delignat-Lavaud, A., Pironti, A., Langley, A., & Ray, M. (2015, September). *Transport Layer Security (TLS) session hash and extended master secret extension* (RFC No. 7627). 48377 Fremont Blvd., Suite 117, Fremont, California 94538, USA: Internet Engineering Task Force. Internet Requests for Comments. Retrieved from http://www.ietf.org/rfc/rfc5246.txt

Checkoway, S., Niederhagen, R., Everspaugh, A., Green, M., Lange, T., Ristenpart, T., … Fredrikson, M. (2014). On the practical exploitability of Dual EC in TLS implementations. In *23rd USENIX security symposium (USENIX security 14)* (pp. 319–335).

Cohen, M. (2015a). *Rekall memory forensics framework.* http://www.rekall-forensic.com/about.html.

Cohen, M. (2015b). *Rekall memory forensics framework.* http://www.rekall-forensic.com/index.html.

Daigniere, F. (2013). *TLS 'secrets'* [Conference]. https://media.blackhat.com/us-13/US-13-Daigniere-TLS-Secrets-WP.pdf. (Presentation at BlackHat USA 2013)

Delphy, B. (2013). *Windbg et l'extension de mimikatz!* (Blog No. November 25). http://blog.gentilkiwi.com/securite/mimikatz/windbg-extension.

Delphy, B. (2016a). *Mimikatz.* https://github.com/gentilkiwi/mimikatz.

Delphy, B. (2016b). *Mimikatz.* https://github.com/gentilkiwi/mimikatz/tree/master/mimikatz/modules/sekurlsa/crypto/kuhl_m_sekurlsa_nt6.c.

Dierks, T., & Allen, C. (1999, January). *The TLS protocol version 1.0* (RFC No. 2246). 48377 Fremont Blvd., Suite 117, Fremont, California 94538, USA: Internet Engineering Task Force. Internet Requests for Comments. Retrieved from http://www.ietf.org/rfc/rfc2246.txt

Dierks, T., & Rescorla, E. (2008, August). *The Transport Layer Security (TLS) protocol version 1.2* (RFC No. 5246). 48377 Fremont Blvd., Suite 117, Fremont, California 94538, USA: Internet Engineering Task Force. Internet Requests for Comments. Retrieved from http://www.ietf.org/rfc/rfc5246.txt (http://www.rfc-editor.org/rfc/rfc5246.txt)

Diffie, W., Van Oorschot, P. C., & Wiener, M. J. (1992). Authentication and authenticated key exchanges. *Designs, Codes and cryptography*, *2*(2), 107–125.

Dolan-Gavitt, B. (2007). The VAD tree: A process-eye view of physical memory. *Digital Investigation*, *4*, 62–64.

Dreijer, J., & Rijs, S. (2013, December). *Perfect forward not so secrecy.* https://os3.nl/_media/2013-2014/courses/ssn/projects/perfect_forward_not_so_secrecy_report.pdf.

Elliot, S. (2014). *RDP Replay* (Blog No. October 30). http://www.contextis.com/resources/blog/rdp-replay/.

Freier, A., Karlton, P., & Kocher, P. (2011, August). *The Secure Sockets Layer (SSL) protocol version 3.0* (RFC No. 6101). 48377 Fremont Blvd., Suite 117, Fremont, California 94538, USA: Internet Engineering Task Force. Internet Requests for Comments. Retrieved from http://www.ietf.org/rfc/rfc6101.txt

Garfinkel, S. L. (2013). Digital media triage with bulk data analysis and bulk_extractor. *Computers & Security*, *32*, 56–72.

Geffner, J. (2011). *Exporting non-exportable RSA keys* [Conference]. https://media.blackhat.com/bh-eu-11/Geffner/BlackHat_EU_2011_Geffner_Exporting_RSA_Keys-WP.pdf. (Presentation at BlackHat Europe 2011)

Goh, E.-J., & Boneh, D. (2001, October). *SSLv3/TLS Sniffer (proxy server): Documentation page* [Tool Documentation]. https://crypto.stanford.edu/~eujin/sslsniffer/documentation.html.

Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., ... Felten, E. W. (2008). Lest we remember: Cold-boot attacks on encryption keys. In *17th USENIX security symposium (USENIX security 08)* (pp. 45–60).

Huppert, P. (2015). *Volatility.* https://github.com/volatilityfoundation/community/blob/master/PhilipHuppert/rsakey.py. (RSAKey community plugin)

Inman, K., & Rudin, N. (2002). The origin of evidence. *Forensic Science International*, *126*(1), 11–16.

Intel Corporation. (2015, December). Intel® 64 and IA-32 Architectures software developer's manual (Computer software manual No. 325462-057US).

Jaqueme, L. (2015). *Volatility.* https://github.com/volatilityfoundation/community/blob/master/Lo%C3%AFcJaquemet/vol_haystack.py. (HayStack community plugin)

Klein, T. (2006, February). *All your private keys are belong to us. extracting RSA private keys and certificates from process memory.* http://www.trapkit.de/research/sslkeyfinder/keyfinder_v1.0_20060205.pdf.

Kornblum, J. (2011). *Finding aes keys* (Blog No. January 18). http://jessekornblum.livejournal.com/269749.html.

Ligh, M. H., Case, A., Levy, J., & Walters, A. (2014). *The art of memory forensics: Detecting malware and threats in Windows, Linux, and Mac memory.* John Wiley & Sons.

Microsoft. (n.d.-a). *Certificate file formats* (Technet Article No. cc770735). One Microsoft Way, Redmond, WA 98052-7329, USA: Microsoft Corporation. https://technet.microsoft.com/en-us/library/cc770735.aspx.

Microsoft. (n.d.-b). *Coding style conventions* (Developer Network Article No. 378932). One Microsoft Way, Redmond, WA 98052-7329, USA: Microsoft Corporation. https://msdn.microsoft.com/en-us/library/windows/desktop/aa378932.aspx.

Microsoft. (n.d.-c). *Common data types* (Document No. cc230309). One Microsoft Way, Redmond, WA 98052-7329, USA: Microsoft Corporation. https://msdn.microsoft.com/en-us/library/cc230309.aspx.

Microsoft. (n.d.-d). *Cryptographic primitives* (Document No. bb204776). One Microsoft Way, Redmond, WA 98052-7329, USA: Microsoft Corporation. https://msdn.microsoft.com/en-us/library/windows/desktop/bb204776.aspx.

Microsoft. (n.d.-e). *Memory limits for Windows and Windows Server releases* (Developer Network Article No. aa36677). One Microsoft Way, Redmond, WA 98052-7329, USA: Microsoft Corporation. https://msdn.microsoft.com/en-us/library/windows/desktop/aa366778.aspx.

Microsoft. (n.d.-f). *Memory pools* (Document No. aa965226). One Microsoft Way, Redmond, WA 98052-7329, USA: Microsoft Corporation. https://msdn.microsoft.com/en-us/library/windows/desktop/aa965226.aspx.

Microsoft. (n.d.-g). *Names of classes, structs, and interfaces* (Document No. ms299040). One Microsoft Way, Redmond, WA 98052-7329, USA: Microsoft Corporation. https://msdn.microsoft.com/en-us/library/ms299040.aspx.

Microsoft. (n.d.-h). *PEB structure* (Document No. aa813706). One Microsoft Way, Redmond, WA 98052-7329, USA: Microsoft Corporation. https://msdn.microsoft.com/en-us/library/windows/desktop/aa813706.aspx.

Microsoft. (n.d.-i). *s (search memory)* (WinDbg Document No. ff558855). One Microsoft Way, Redmond, WA 98052-7329, USA: Microsoft Corporation. https://msdn.microsoft.com/en-us/library/windows/hardware/ff558855.aspx.

Microsoft. (2003a, July). *How TLS/SSL works* (Document No. cc783349). One Microsoft Way, Redmond, WA 98052-7329, USA: Microsoft Corporation. https://technet.microsoft.com/en-us/library/cc783349.aspx.

Microsoft. (2003b, March). *TLS/SSL tools and settings* (Document No. cc776467). One Microsoft Way, Redmond, WA 98052-7329, USA: Microsoft Corporation. https://technet.microsoft.com/en-us/library/cc776467.aspx.

Microsoft. (2010, July). *How to create a user-mode process dump file in Windows Vista and in Windows 7* (Knowledge Base Article No. 931673). One Microsoft Way, Redmond, WA 98052-7329, USA: Microsoft Corporation. https://support.microsoft.com/en-us/kb/931673.

Microsoft. (2014a, July). *CNG features* (Document No. bb204775). One Microsoft Way, Redmond, WA 98052-7329, USA: Microsoft Corporation. https://msdn.microsoft.com/en-us/library/windows/desktop/bb204775.aspx.

Microsoft. (2014b, July). *Key storage and retrieval* (Document No. bb204778). One Microsoft Way, Redmond, WA 98052-7329, USA: Microsoft Corporation. https://msdn.microsoft.com/en-us/library/windows/desktop/bb204778.aspx.

Microsoft. (2014c, July). *Prioritizing Schannel cipher suites* (Document No. bb870930). One Microsoft Way, Redmond, WA 98052-7329, USA: Microsoft Corporation. https://msdn.microsoft.com/en-us/library/windows/desktop/bb870930.aspx.

Microsoft. (2014d, July). *Transport Layer Security protocol* (Document No. dn786441). One Microsoft Way, Redmond, WA 98052-7329, USA: Microsoft Corporation. https://technet.microsoft.com/en-us/library/dn786441.aspx#BKMK_SessionResumption.

Microsoft. (2015a, July). *DER encoding of ASN.1 types* (Document No. bb648640). One Microsoft Way, Redmond, WA 98052-7329, USA: Microsoft Corporation. https://msdn.microsoft.com/en-us/library/windows/desktop/bb648640.aspx.

Microsoft. (2015b, November). *Public and private symbols* (Document No. ff553493). One Microsoft Way, Redmond, WA 98052-7329, USA: Microsoft Corporation. https://msdn.microsoft.com/en-us/library/windows/hardware/ff553493.aspx.

Microsoft. (2015c). *Remote Desktop Connection: frequently asked questions.* http://windows.microsoft.com/en-us/windows/remote-desktop-connection-faq. (Question “Which PCs can I connect to using Remote Desktop Connection?”)

Microsoft. (2015d, July). *Schannel SSP overview* (Document No. dn786429). One Microsoft Way, Redmond, WA 98052-7329, USA: Microsoft Corporation. https://technet.microsoft.com/en-us/library/dn786429.aspx.

Microsoft. (2015e, November). *TLS/SSL settings* (Document No. dn786418). One Microsoft Way, Redmond, WA 98052-7329, USA: Microsoft Corporation. https://technet.microsoft.com/en-us/library/dn786418.aspx.

Microsoft. (2015f). *Upgrade to Windows 10: Faq.* http://windows.microsoft.com/en-us/windows-10/upgrade-to-windows-10-faq. (Question “What is the free upgrade to Windows 10”)

miTLS. (n.d.). *Cross-layer protocol attacks.* https://mitls.org/pages/attacks/VHC.

Mozilla. (2015). *NSS key log format.* Mozilla Developer Network. Retrieved from https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/Key_Log_Format

Myerson, T. (2015). *Windows 10: Preparing to upgrade one billion devices* (Blog No. July 2). https://blogs.windows.com/windowsexperience/2015/07/02/windows-10-preparing-to-upgrade-one-billion-devices/.

NIST. (2001, November). *Announcing the Advanced Encryption Standard (AES)* (FIPS Publication No. 197). 100 Bureau Drive, Stop 1070, Gaithersburg, MD 20899-1070, USA: National Institute of Standards and Technology (NIST). Federal Information Processing Standards (FIPS) Publication. Retrieved from http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

Petroni, N. L., Jr., Walters, A., Fraser, T., & Arbaugh, W. A. (2006, December). Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digit. Investig.*, *3*(4), 197–210. Retrieved from http://dx.doi.org/10.1016/j.diin.2006.10.001 doi: 10.1016/j.diin.2006.10.001

Picasso, F. (2015). *Happy dpapi!* (Blog No. January 13). http://blog.digital-forensics.it/2015/01/happy-dpapi.html.

Picod, J.-M. (2016). *Dpapick.* https://bitbucket.org/jmichel/dpapick/src/37a929678b0c5d6a6f025e26d262fbfcebaeebb7/DPAPI/Probes/certificate.py. (Certificate Probe python file)

Polk, T., McKay, K., & Chokhani, S. (2014). *Guidelines for the selection, configuration, and use of Transport Layer Security (TLS) implementations* (SP No. 800-52 Revision 1). 100 Bureau Drive, Stop 1070, Gaithersburg, MD 20899-1070, USA: National Institute of Standards and Technology (NIST). NIST Special Publication. Retrieved from http://dx.doi.org/10.6028/NIST.SP.800-52r1 doi: 10.6028/NIST.SP.800-52r1

Russinovich, M. E., Solomon, D. A., & Allchin, J. (2005). *Microsoft Windows internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000* (Vol. 4). One Microsoft Way, Redmond, WA 98052-7329, USA: Microsoft Press Redmond.

Russinovich, M. E., Solomon, D. A., & Ionescu, A. (2012). *Windows internals* (Vol. 6). One Microsoft Way, Redmond, WA 98052-7329, USA: Microsoft Press Redmond.

Salowey, J., Zhou, H., Eronen, P., & Tschofenig, H. (2008, January). *Transport Layer Security (TLS) session resumption without server-side state* (RFC No. 5077). 48377 Fremont Blvd., Suite 117, Fremont, California 94538, USA: Internet Engineering Task Force. Internet Requests for Comments. Retrieved from http://www.ietf.org/rfc/rfc5077.txt

Shamir, A., & van Someren, N. (1998). Playing 'hide and seek' with stored keys. In *Financial cryptography* (pp. 118–124).

Shirey, R. (2007, August). *Internet security glossary, version 2* (RFC No. 4949). 48377 Fremont Blvd., Suite 117, Fremont, California 94538, USA: Internet Engineering Task Force. Internet Requests for Comments. Retrieved from http://www.ietf.org/rfc/rfc4949.txt

Taubert, T. (2014). *Botching forward secrecy. the sad state of server-side TLS session resumption implementations* (Blog No. November). https://timtaubert.de/blog/2014/11/the-sad-state-of-server-side-tls-session-resumption-implementations/.

Volatility. (2015). *Volatility.* https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/dumpcerts.py. (DumpCerts plugin)

Walters, A., & Petroni, N. L. (2007). *Volatools: Integrating volatile memory into the digital investigation process* [Conference]. https://www.blackhat.com/presentations/bh-dc-07/Walters/Presentation/bh-dc-07-Walters-up.pdf. (Presentation at BlackHat DC 2007)

Warren, T. (2015). *Why Microsoft is calling Windows 10 'the last version of Windows'* (No. May 7). http://www.theverge.com/2015/5/7/8568473/windows-10-last-version-of-windows.

Wireshark. (2015). *Wireshark.* https://github.com/wireshark/wireshark/blob/master/epan/dissectors/packet-ssl-utils.c.