REVERSE ENGINEERING SMART CARD MIDDLEWARE

Aleksandar Nikolić, Goran Sladić, Branko Milosavljević
{anikolic, sladicg, mbranko}@uns.ac.rs
University of Novi Sad, Faculty of Technical Sciences, Novi Sad, Serbia

Abstract – *Many times the software source code or the implementation specification is not available and reverse engineering is employed in order to achieve interoperability with other systems or to ensure standards compliance. The goal of this paper is to introduce tools and techniques for reverse engineering of the smart card middleware layer on the Windows platform.*

1. INTRODUCTION

Due to their relatively small form factor and broad range of capabilities, smart cards have been employed in diverse areas such as payment cards, identification and authentication devices, government-issued IDs, hardware security modules and others. There are different types of smart cards that vary in complexity and capability. Simple storage cards are usually used as prepaid cards for various cash transfers. Other, more sophisticated, cards have microprocessors capable of executing tasks and can have specialized components to perform cryptographic operations. The cryptographic smart cards are used for the tamper-resistant storage of private keys and sensitive information and for the isolation of critical computations involving digital signatures, authentication and key exchange.

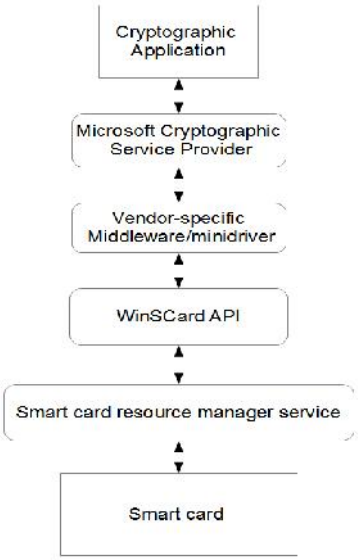

Figure 1. Windows smart card architecture

In order for a smart card to be usable by a third party application, a vendor-supplied driver or a middleware needs to be available. Figure 1 illustrates different layers of the Windows smart card architecture [1]. On the Windows platform, a smart card middleware must follow the Windows Smart Card Minidriver specification [2] which is designed to present a consistent interface to the card. This is not always the case, and card vendors sometimes implement custom and non-standard interfaces.

When the specification or the source code of the middleware is not available, but there is a need to achieve the interoperability with systems not supported by the card vendor or for the security testing and assurance, reverse engineering can be employed. Since smart cards are designed to be tamper-resistant and secure, the reverse engineering of the smart card itself can be time consuming and requires the special equipment. The focus of this paper is to introduce tools and techniques for examining and reverse engineering of the smart card's software layer. Presented tools and techniques have been successfully applied on the Serbian electronic ID middleware.

2. REVERSE CODE ENGINEERING

The term reverse code engineering stands for a process of analysing the system to create representation of the system at the higher level of abstraction. This paper deals with the reverse engineering of the binary code for the x86 family of processors on the Windows platform. The reverse engineering can be accomplished in two ways:

- disassembly using a disassembler
- analysis through observations of a running code using a debugger and other tools

Using the disassembler, the binary code can be converted into assembly instructions. The assembly code is free of abstractions such as variable and function names which are available in higher level languages. The goal of disassembling the binary code is to understand its logic and, if possible, reconstruct the source code. This process is aided by tools that can split the code into the constituent functions and present the analyst with a higher view of code flow by, for example, the presenting function call graphs and the highlighting loops. One of the most common disassemblers for windows is IDA Pro (Interactive Disassembler Pro) [3]. IDA Pro aids the reverse engineering process by automatic code analysis and knowledge of the standard API functions for identifying cross-references and recognizing known data types and functions.

When analysing a complex code, the static analysis is often not enough, and observations of the running code must be made. This can be achieved by observing a process under the debugger which allows for observing the executed code, stopping the execution when needed and examining content of the registers and memory. The most popular debuggers on Windows are Windows Debugger, which can be used for the kernel level debugging, and OllyDBG [4] which is specifically developed for reverse engineering and has a wide array of useful plugins. Other popular tools are, for example, ProcMon [5], for file and registry monitoring and Wireshark [6] for network activity monitoring.

A native executable code on Windows comes in two forms, an executable file, or a dynamic loadable library (DLL) file, both of which are specified by the PE (Portable Executable) file format [7]. All functions in the DLL can be exported. The exported functions can be used outside the DLL and their names, arguments and locations are available. Information about the names, arguments and locations of non-exported functions are usually stripped from the DLL before distribution. Next section deals with identifying non-exported functions in the smart card middleware.

3. IDENTIFYING KEY MIDDLEWARE FUNCTIONS

When faced with a smart card of a unknown specification, the middleware functions that directly communicate with the card should be examined. The middleware is responsible for abstracting the card to the programmer and examining these abstractions gives further insight into the smart card's design.

In order for the Windows applications to be able to communicate with the smart card, a smart card middleware (minidriver) that implements the required interface must be installed. The smart card middleware is distributed as a DLL (Dynamic Loadable Library) file which exports a specific set of functions defined by the minidriver specification.

Before accessing the card in any way, the middleware must be initialized. Initialization is done by calling *CardAcquireContext* [8] function. *CardAcquireContext* must be exported by the middleware and is usually the only exported function.

In the case when the middleware binary code is distributed without the debugging symbols, (which usually is the case) the names of the non-exported functions are not available. In IDA Pro, unknown functions are named by their address prefixed with *sub*, short for subroutine. By analysing the *CardAcquireContext* function's disassembly in IDA Pro and adding information available from the specification, locations and names of key, the middleware functions can be determined. It's prototype is specified by the minidriver specification and is shown in Listing 1.

```
DWORD WINAPI CardAcquireContext(
  __in  PCARD_DATA pCardData,
  __in  DWORD dwFlags
);
```

Listing 1. CardAcquireContext prototype.

PCARD_DATA is a pointer to the *CARD_DATA* C structure [9] which, after the function returns, contains the context information used during communication between a smart card and the service provider. The structure includes the function pointers to the all functions that the minidriver implements. Table 1 shows the partial list of the function pointers.

Function pointer name	**Description**
pfnCardCreateFile	Creates a new file
pfnCardReadFile	Reads a file
pfnCardWriteFile	Writes to a file
pfnCardAuthenticatePin	Checks if the pin is valid
pfnCardRSADecrypt	Decrypts the supplied data

Table 1. Some of the function pointers in CARD_DATA structure

The functions in Table 1 are usually not exported by the middleware DLL, which means that their locations are unknown prior to analysis.

```
mov     esi, [ebp+cardData]
mov     eax, [esi]
mov     dword ptr [esi+3Ch], offset sub_10005DE0
mov     dword ptr [esi+40h], offset sub_10005F10
mov     dword ptr [esi+68h], offset off_10003FA0
mov     dword ptr [esi+6Ch], offset off_10005990
mov     dword ptr [esi+88h], offset sub_10004820
mov     dword ptr [esi+70h], offset off_10004DD0
mov     dword ptr [esi+84h], offset off_10004BC0
mov     dword ptr [esi+7Ch], offset sub_10004BD0
mov     dword ptr [esi+80h], offset off_10005980
mov     dword ptr [esi+8Ch], offset sub_10004E30
mov     dword ptr [esi+90h], offset sub_10005020
```

Figure 2. Function pointers assignment.

As discussed, *CardAcquireContext* is responsible, amongst other tasks, for setting the function pointers in the *CARD_DATA* structure. In the assembled code, this will usually be represented by the series of the pointer assignment operations to the appropriate offsets into the *CARD_DATA* structure. Figure 2 shows an example of the disassembled code responsible for the function pointer assignment.

The IDA Pro disassembler supports loading the custom header files which can be used for describing the known structures. The C header file can be used to define *CARD_DATA* structure in IDA. By setting the ESI register (Figure 2) as a start of the *CARD_DATA* structure, the *CardAcquireContext* function is easier to reverse engineer as offsets into the structure are recognized and labelled in the disassembled code as shown in Figure 3.

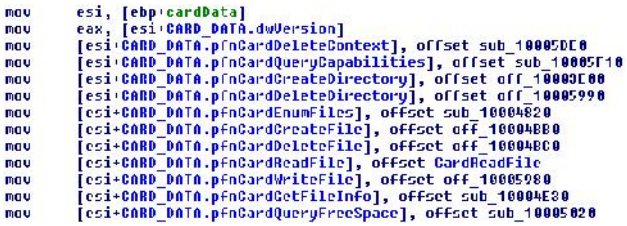

Figure 3. Recognized structure filed access

In the code in Figure 2, the *ESI* register points to the base of the *CARD_DATA* structure. By comparing the offsets from the *ESI* in Figure 2 with *CARD_DATA*, it can be inferred what the newly assigned pointer represents. For example, the offset 0x3c corresponds to *pfnCardDeleteContext*, so the *sub_10005DE0* function is actually the *CardDeleteContext* function. The offset 0x78 is *CardCreateFile*, 0x7c is *CardReadFile*, and so on. Functions like *CardAuthenticatePin* (offset 0x50), *CardSignData* and similar are of special interest as they give insight into the used card's cryptographic capabilities. After examining other parts of *CardAcquireContext*, its source code can be reconstructed. The reverse engineering of the newly identified functions can be approached in the similar manner since their argument types and names are also specified by the minidriver specification. Identification of the type and format of the APDU [10] (Application Protocol Data Unit) commands used by the card, and the cryptographic algorithms used in the middleware are the most important elements which should be obtained by the reverse engineering.

4. IDENTIFYING CRYPTOGRAPHIC PRIMITIVES

Smart cards can be used in various cryptographic protocol schemes. In the case of the lack of support for some cryptographic primitive in the card itself, that primitive can be implemented in the middleware. For example, if the card doesn't support a certain cryptographic hashing algorithm, before data is signed on the card it must be first hashed by the middleware. Identifying the known cryptographic algorithms used in the middleware is imperative for successful reverse engineering.

In the case when a known cryptographic library is linked into the middleware, the IDA Pro's FLIRT (Fast Library Identification and Recognition Technology) engine can be used. It has a database of signatures of well known libraries and can recognize them in unknown binaries. In the case when a unknown or custom implementation of cryptographic algorithms is used, the individual algorithms instead of the whole libraries must be matched. One approach is to use static tools to analyse the binary stream of the given software without executing it. Most static tools use signatures to detect the presence of the particular cryptographic primitive in the compiled code. Due to the fact that there are no big variations in the different implementations of the same algorithm, signatures can match the specific code snippets, constants and known structures. PeiD KANAL has a large database of signatures for the known cryptographic algorithms as well as for compression and encoding algorithms and can export results into IDA Pro. FindCrypt works as the IDA Pro plugin, and the OllyDBG plugin CryptoScanner can be used during process runtime to scan the process memory. The evaluation results against the same middleware used as example presented in Figure 1 are shown in Table 2. Detailed comparison of signature based tools has been made by Grobert in [11].
AES and DES algorithms have been detected by their Sbox structures. MD5 and SHA-1 have been detected by their magic constants and initialization values. Findings can then be imported into IDA Pro where address cross referencing can be used to find parts of the code using the detected data.

	KANAL	FindCrypt	CryptoScanner
MD5	+	+	+
SHA-1	+	+	+
AES	+	+	+
DES Crypt1	+		
DES Crypt2			+

Table 2. Algorithm detection results

Static tools are very limited as they rely purely on the signatures; therefore dynamic approaches have been developed. In [11] author uses Pin for dynamic binary instrumentation to produce a trace of program's execution. By analysing the trace, the high-level information can be reconstructed and the cryptographic algorithm can be detected.

Research presented in [12] shows higher percentages of the arithmetic and bitwise instructions used in cryptographic than non-cryptographic code which can be leveraged to detect even previously unknown and custom algorithms. Lutz [13] has shown that by using the dynamic analysis decryption code can be detected based on the observation that decryption process decreases and encryption increases data entropy.

It should be noted that static tools have difficulty detecting the public-key cryptography algorithms. In some implementations, the middleware can read the private and public keys from the card into the memory. The RSA public and private keys are usually stored in the PKCS#8 (Public-key Cryptography Standard) format and have very specific structure which is ASN.1 (Abstract Syntax Notation One) encoded. Listing 2 shows the ASN.1 syntax for the private key information.

```
PrivateKeyInfo ::= SEQUENCE {
 version                 Version,
 privateKeyAlgorithm
PrivateKeyAlgorithmIdentifier,
 privateKey              PrivateKey,
 attributes          [0]  IMPLICIT Attributes OPTIONAL
}
 Version ::= INTEGER
PrivateKeyAlgorithmIdentifier ::= AlgorithmIdentifier
 PrivateKey ::= OCTET STRING
 Attributes ::= SET OF Attribute
```

Listing 2. PKCS#8 RSA Private Key format

By attaching the debugger to a process using the middleware and searching the memory for the known combinations of version and *PrivateKeyAlgorithm-Identifier* constants, the private keys can be found in the memory. Measuring data entropy can also be used to search the process memory for public or private keys as they have relatively high entropy compared to the regular (usual) data. The memory regions with high entropy that does not fit the public or private key format can be assumed to be encrypted data. Setting the memory access breakpoints on those regions in the debugger can lead to the decryption code when breakpoints get hit.

5. CAPTURING DATA TRAFIC

When the reverse engineering of a complex systems is performed, a great deal of information about the system can be inferred by analysing the data. This is especially true for the smart cards since the communication between the smart card and the middleware is done trough the specific protocol. Generally, the middleware sends the APDU commands with specific parameters to the smart card and the smart card responds with the APDU response. Capturing this data exchange can help greatly in understanding the smart card and middleware design. The APDU structure is defined by the ISO/IEC 7816-4 standard [10]. The command APDU contains a mandatory 4 byte long header and up to 255 bytes of data.

Field name	**Length (bytes)**	**Description**
CLA	1	Instruction class
INS	1	Instruction code
P1 and P2	2	Instruction parameters
Lc	0,1 or 3	Length of command data
Command data	Variable	Command data
Le	0,1,2 or 3	Max. lengthe of response data

Table 3. Command APDU format.

Field name	**Length (bytes)**	**Description**
Response data	At most Le	Response data
SW1 and SW2	2	Command status

Table 4. Response APDU format

The response APDU is sent by the card to the reader and contains 2 byte status word and up to 255 bytes of data. Tables 3 and 4 present the command and response APDU formats respectively. The command status “00 90” in hexadecimal signifies that the command has been executed successfully. In the case when a non standard compliant card is used, the correlations between the identified middleware functions and the observed data traffic can be used to identify the unknown instruction.

On the Windows operating systems, all smart card communication is done trough *WinSCard* API which defines the low level smart card access function. The *ScardTransmit* [14] function sends the command APDUs to the card and returns the response. A solution for monitoring the data transmission from and to the cards is by intercepting the *SCardTransmit* function calls and inspecting the arguments and return values. Function interception is often called function hooking and can be achieved in various ways on Windows. The simplest method is to set a breakpoint on the beginning of the function in the debugger. This method is not very flexible as it works only on the process being debugged and may interfere with the card functionality. A more sophisticated method is to perform the system wide function hooking by injecting a specially crafted DLL into the every process that uses *Winscard* API. This can be done by using the Windows *AppInit_DLLs* mechanism. When a DLL is set as *AppInit_DLL* in the Windows registry, it will be loaded whenever a new process is started. Thus crafted DLL can replace the original *SCardTransmit* function in order to record sent and received data. The function hooking works by overwriting the start of the original function with the unconditional jump to the replacement function in such way that the hooked function can still be called from the hooking function. The replacement function must have exactly the same return value and arguments as the original one. The *SCardTransmit* function prototype is shown in Listing 3.

```
LONG WINAPI SCardTransmit(
  _In_       SCARDHANDLE hCard,
  _In_       LPCSCARD_IO_REQUEST pioSendPci,
  _In_       LPCBYTE pbSendBuffer,
  _In_       DWORD cbSendLength,
  _Inout_opt_  LPSCARD_IO_REQUEST pioRecvPci,
  _Out_      LPBYTE pbRecvBuffer,
  _Inout_    LPDWORD pcbRecvLength
);
```

Listing 3. SCardTransmit function prototype.

The third *SCardTransmit* argument, *pbSendBuffer* (Listing 3), contains the command APDU that will be sent to the card, and the sixth argument is the pointer to a byte array where the response APDU will be received. The function hooking of the original *SCardTransmit* includes the following steps:

1) Record data that will be sent to the card (*pbSendBuffer*).
2) Call original the *SCardTransmit* function.
3) Record the data received from the card (*pbRecvBuffer*).
4) Return the status returned by the original *SCardTransmit* function.

The *SCardTransmit* function's return value can either be *SCARD_S_SUCCESS* (indicating the successful write to the card) or an error code. Another advantage of the *SCardTransmit* function hooking technique is that it is independent of the language the analysed process is written in, as long as it's smart card communication is performed using *WinScard* API.

The SmartcardSniffer tool [15] is developed for the function hooking and recording of transmitted data. It is implemented in the C programming language using the the Mhooks hooking library. During the communication between the Windows process and the smart card, a log file is created, named after the process, in which is all data recorded as a hexadecimal stream. The transmitted data is prefixed with the ">>>" symbol, while the received data is prefixed with the "<<<" symbol. Each data transmission is recorded in the log file immediately and file access is freed to combine the log analysis with the process instrumentation in the debugger. Listing 4 shows the example of recorder data transmission.

```
Winscard!SCardTransmit:
>>> 00:A4:08:00:02:0F:02
<<< 6F:38:62:36:83:02:0F ... 90:00
Winscard!SCardTransmit:
>>> 00:B0:00:00:06
<<< 00:0D:02:00:63:00:90:00
Winscard!SCardTransmit:
>>> 00:B0:00:00:60
<<< 00:0D:02:00:63:00:0A ... 90:00
Winscard!SCardTransmit:
>>> 00:B0:00:60:09
<<< 00:53:43:11:06:02:00:53:43:90:00
Winscard!SCardTransmit:
>>>
00:A4:04:00:0B:A0:00:00:03:97:43:49:44:5F:01:00
<<< 6A:82
```

Listing 4. Sample log file.

The first APDU command in the listing 4 has the instruction A4 which, by the ISO/IEC 7816-4 standard, specifies the *SELECT FILE* instruction responsible for selecting a particular elementary file on the smart card with the name "0F 02". The last two bytes of the response APDU are "90 00" which indicates the success. By enumerating all unique *SELECT FILE* instructions executed, a list of the elementary files present on the card can be created. The next three commands have the INS byte set to B0 which corresponds to the *READ BINARY* instruction. The *READ BINARY* instruction can read the maximum of 255 bytes of data at a time so multiple requests with incrementing offsets would be needed to read a large file. The file can then be reconstructed by the log. The Last command is the *SELECT FILE* instruction which response indicates an error has occurred. The extensive list of known command APDU instructions can be found in [16].

6. SERBIAN ELECTORNIC IDENTITY CARD

The Serbian electronic identity card is a smart card that contains a basic owner information, a photograph and a personal digital certificate that can be used to create digital signatures and authentication. As such, the card is capable of performing multiple cryptographic operations as well as securely store private keys. The official Čelik API [17] and middleware has been released for the Windows platform. Currently, there is no official support for other platforms.

Presented tools an techniques have been applied in analyzing the middleware. The goal was to implement the platform independent library which would enable the use of the card on systems other than Windows.

To access the protected elementary files and to execute some privileged instructions a card user must first be authenticated. Authentication is usually done by supplying a PIN (Personal Identification Number) which is verified by the card. The *VERIFY APDU* instruction is used to send PIN to the card. The first step in reverse engineering the electronic ID card is to reconstruct this authentication mechanism.

In the case of the Serbian electronic ID, the card owner is supplied with a password instead of PIN by the card issuer. Reason for this is to enable the user to change the password, since the card PIN cannot be changed easily. Thus, it can be concluded that PIN must somehow be derived from the password. By using the presented technique for identification of the key middleware functions the location of the *CardAuthenticatePin* function can be found. Combining the program debugging and the static analysis it can be found that the SHA-1 cryptographic hash algorithm is used during the password/PIN authentication. Additional data required for deriving the PIN from the password can be found in the log file recorded by SmartcardSniffer. PIN is stored encrypted on the card in the "0F 03" elementary file and can be read without a prior user authentication. It is encrypted using the XOR based custom cipher in which the cipher key is constructed as the password hash. When the password is changed, the actual PIN remains the same, but it's stored encrypted using the new password.

After the authentication mechanism has been determined, the card's cryptographic functions can be accessed. Every electronic ID card has a digital certificate with corresponding private key stored on the card. To access the card's public-key cryptography capabilities, the previously identified middleware functions need to be analyzed. By recording the middleware and the smart card communication during the signing of two different pieces of data, it can be shown that the communication is exactly the same. This means that data to be signed is never sent to the smart card. Instead, after successful authentication, the user's private key is read from the smart card and data is signed by the middleware. Combining the static and dynamic analysis shows that the AES cryptographic algorithm is used before the data is signed. The private key is stored encrypted, by AES with 256 bit key, on the smart card and is decrypted in the middleware when needed. The CFB (Cipher Feedback) mode of the operation is used in encryption. The secret key, needed to decrypt the private key, is stored on the card and is encrypted by the same algorithm used for the PIN encryption.

As the result of this analysis, the Java library has been implemented allowing the card's cryptographic capabilities to be used on the non-Windows operating systems as well as in the web browsers [18]. More implementation details are available in [19].

7. CONCLUSION

As the heterogeneity of used platforms and operating system (Linux, OS X, Android) increases, the need for portability and interoperability between different platforms becomes imperative. Reverse engineering is one of the ways to achieve interoperability with closed and proprietary systems.

This paper presents tools and techniques that can be used to analyze and reverse engineer smart card middleware of unknown origin as opposed to analyzing the smart card directly.

By having a higher knowledge of the Windows smart card architecture and the middleware requirements, analyst can quickly find the functions of interest. Employing the static tools can help identify cryptographic algorithms present in the middleware code. Analysis of data sent from and to the card can supply information about the card's organization and functionality.

The tool to capture data traffic exchanged between the smart card and the middleware has been developed and presented techniques have been tested by reverse engineering the Serbian electronic identity card.

Further work may include automating the analysis of captured data to recognize certain instructions, to extract files and possibly to detect cryptographic keys in the hex stream.

REFERENCES

[1] Windows Smart Card Technical Reference, *Microsoft Corporation*, 2010.

[2] Windows Smart Card Minidriver Specification, *Microsoft Corporation* , 2012.

[3] Interactive Disassembler PRO, http://www.hex-rays.com/products/ida/index.shtml

[4] Olly Debugger, http://www.ollydbg.de/

[5] Process Monitor, http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx

[6] Wireshark, http://www.wireshark.org/

[7] Microsoft PE and COFF Specification, Microsoft Corporation, 2010.

[8] CardAcquireContext Function, MSDN Library , http://msdn.microsoft.com/en-us/library/dd627600%28v=vs.85%29.aspx

[9] CARD_DATA Structure, MSDN Library , http://msdn.microsoft.com/en-us/library/dd627628%28v=vs.85%29.aspx

[10] ISO/IEC 7816-4, Organization, security and commands for interchange, 2008.

[11] F. Gröbert, Automatic Identification of Cryptographic Primitives in Software, Diploma Thesis, Ruhr-University, Bochum

[12] Z. Wang, X. Jiang, W. Cui, and X. Wang. ReFormat: Automatic Reverse Engineering of Encrypted Messages. Technical report, NC State University, 2008.

[13] N. Lutz. Towards Revealing Attackers' Intent by Automatically Decrypting Net-work Traffic. Master's thesis, ETH Zuurich, 2008.

[14] ScardTransmit Function, MSDN Library http://msdn.microsoft.com/en-us/library/windows/desktop/aa379804%28v=vs.85%29.aspx

[15] Smartcard Sniffer tool , http://code.google.com/p/smartcard-sniffer/

[16] Smart card selected information – APDU List, http://web.archive.org/web/20090630004017/http://cheef.ru/docs/HowTo/APDU.info

[17] Čelik API, http://ca.mup.gov.rs/download.html

[18] Open source library for PKI functions on Serbian eID , http://code.google.com/p/rsidlib/

[19] A. Nikolić, Implementation of a library for digital signing by electronic ID (in serbian), bachelor thesis, Faculty of Technical Sciences Novi Sad, 2011.

ACKNOWLEDGMENTS

Results presented in this paper are partialy funded as the research conducted within the Grant No. III-44010, Ministry of Science and Technological Development of the Republic of Serbia