Vulnerability Analysis of a Commercial .NET Smart Card

Behrang Fouladi[1], Konstantinos Markantonakis[2] and Keith Mayes[2]

[1]Microsoft Security Response Center, UK
behrang.fouladi@microsoft.com
[2]Smart Card Centre
Royal Holloway, University of London, TW20 0EX, UK
{k.markantonakis, keith.mayes}@rhul.ac.uk

Abstract. In this paper we discuss the operating system security measures of a commercial .NET smart card for mitigating risks of malicious smart card applications. We also investigate how these security measures relate to the card resident binary by analysing its proprietary file format to develop a new vulnerability research tool for .NET card applications. This tool enables us to modify compiled card applications for creating vulnerability research test cases. We then present the details of the vulnerabilities in the target .NET virtual machine (VM) which have been discovered using this tool. The vulnerabilities relate to potential misuse of administrator privileges, therefore, we conclude with recommending countermeasures to be implemented in the card manager application and .NET VM to fix those vulnerabilities.

Keywords: .NET Smart Card, Embedded .NET, Smart Card Firewall, VM Vulnerabilities, File Format.

1 Introduction

Modern smart cards are commonly referred to as "multi-application" cards, as they are capable of storing and running more than one application. This increases user convenience and lowers the costs of issuing and managing multiple cards for different applications. However, such a smart card system would require more complex security features in order to isolate execution and the data access context of on-card applications from each other. The card's operating system should also allow secure installation of new applications to the card without having to reissue it. There are currently three major multi-application smart card platforms in the market that allow the card issuers or content providers to securely load applications to the smart card during issuance and post-issuance. These platforms are MULTOS [1], Java Card [2] and .NET card [3].

Card applications are written in a high level programming language such as Java, Visual Basic or C# and then the applications are compiled to an intermediary format known as "bytecode". Bytecode instructions consist of a set of numeric codes, values

and references that need to be translated to machine code instructions before being executed by the smart card CPU. The VM is responsible for loading and translating the bytecode application to CPU instructions. The VM concept allows a card application to be executed on different smart card CPUs without the need to re-program it for different CPU architectures. The VM can also perform verifications and security checks on the bytecode before the translation of the bytecode to CPU instructions in order to prevent malicious code attacks. Finally, it can isolate different on-card applications from each other, only allowing controlled data exchange and sharing.

The aforementioned security principles, if implemented correctly, mitigate the risks of malicious code. However, successful attacks against smart card and desktop computer VMs have been demonstrated which exploit the implementation vulnerabilities to escape the application sandbox or bypass safe memory access controls [4], [5]. Therefore, the security of the VM and malicious bytecode threats are interesting topics for smart card vendors and security researchers. The security of the Java Card VM has been the subject for several researchers [6], [7], which demonstrated the exploitation of weaknesses in bytecode verification and translation in order to gain access to the contents of the smart card memory. However no public vulnerability research on .NET smart card VM could be found prior to this work. We aim to perform a detailed security evaluation of this platform to unveil possible vulnerabilities and provide recommendations to address those issues.

2 The .NET Smart Card

The .NET card is widely used multi-factor authentication token in Microsoft Windows based systems. A .NET card supplier will typically also supply a software development kit (SDK) which integrates with Microsoft Visual Studio and enables developers to build card applications in .NET programming languages such as C# and Visual Basic. The .NET card VM is capable of interpreting and executing Microsoft Intermediate Language (MSIL) instructions. Thus, it allows developers to write smart card applications in any .NET programming language that can be compiled into MSIL code. This gives the users similar flexibility and program portability to that provided by the Java Card VM.

A notable advantage of the .NET smart card over a typical Java Card is that the developer does not need to design and use Application Protocol Data Unit (APDU) commands that are necessary for Java Card applications to communicate with the reader terminal. The .NET smart card framework provides a proxy interface based on .NET remoting technology which allows developers to call on-card application services using TCP or HTTP as a transport protocol without having to issue and process low level APDU commands and responses. Behind the scenes, the proxy interface converts client requests and on-card application responses to the corresponding APDU commands as demonstrated in Figure 1. This research work focused on a commercially available and widely used .NET card which from now on will simply be referred to as the target .NET card.

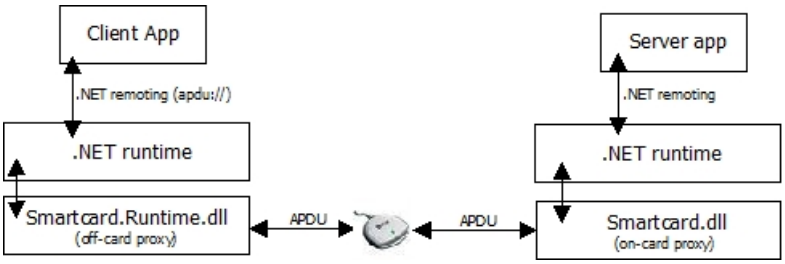

Fig. 1. Target .NET card isolates APDU layer from the application

Since a smart card micro-chip has limited memory and processing resources, it cannot load and run the standard .NET programs compiled by Microsoft Visual Studio. These programs need to be converted to the target .NET Smart Card Framework's card resident binary format before being loaded on to the smart card. The details of this proprietary file format and its security implications will be discussed in section 3. The card resident binaries also need to be digitally signed, so that the card operating system can ensure their integrity and authenticity. The vendor supplied SDK plug-in for visual studio performs both conversion and signing operations automatically after standard Visual Studio compilation.

2.1 .NET Virtual Machine Security Model

The Microsoft .NET framework security model [8] is based on two access control mechanisms: Role Based and Evidence Based access control. The role based access control makes authorisation decisions based on the identity and roles/groups to which the user is assigned. The evidence based access control uses evidences associated with application code to make authorisation decisions. In .NET framework, executable code is maintained and managed in the form of logical units named “Assemblies”. An assembly forms a container for a program’s objects (classes, user-defined types, methods and fields) and data resources such as embedded fonts, graphic files, etc. An assembly can consist of multiple “EXE” and “DLL” files which are called “code modules”. As a result, it forms a deployment unit which contains all required code modules for appropriate execution of the application while simplifying the application installation and update. Evidences are pieces of information that identify .NET assemblies. Some forms of evidences such as .NET Strong Names (SN) and publisher signatures are resistant to forgery and provide a stronger form of code identification compared to an application’s URL or hash code. Assemblies with common or similar evidences are categorized into logical units called Code Groups. “Permission Sets”, which define an access type to a protected resource or the ability to perform an operation, are then assigned to these code groups by the system administrator.

The target .NET card only supports the evidence based security mechanism with assembly SN used as identity evidence. Assembly SN is an important concept in the target .NET VM’s security. It also relates to a vulnerability discussed in section 4. As such, the following section describes it in more details.

2.2 Assembly Strong Name

The Microsoft .NET framework introduced the SN mechanism to generate unique assembly identities which can be used as code authentication tokens to make security decisions by the .NET runtime as well as ensuring the code integrity. The RSA public key signature algorithm, which can provide data integrity and origin authentication, is at the heart of the .NET SN mechanism. In order to bind the SN to the assembly file, an RSA digital signature for the SHA1 hash value of the file content is computed during the build process by using a private key. This key will only be known to the assembly developer. The SN and RSA digital signature are then written into the assembly file's manifest which can later be parsed by the .NET runtime to load the signer's public key. This can be used to verity the assembly's digital signature, thus ensuring code integrity and origin identification. An assembly manifest is usually embedded inside ".EXE" or ".DLL" files together with program code, but it can also be stored inside a standalone file.

The Microsoft .NET framework usually uses a 128 byte RSA key to sign an assembly which is stored inside the assembly's manifest section. The public key is not repeated again in the assembly's strong name. Instead, the .NET framework uses public key identification value called "Public Key Token" (PBKT) which is 8 bytes in size and calculated by the following algorithm:

1. Use SHA-1 hash algorithm to compute the hash value of the public key. The output is 160 bits.
2. The last 8 bytes of the generated SHA-1 hash value are used for calculating the PBKT.
3. The sequence for these bytes are reversed, providing us with the PBKT value

The target .NET VM also uses the above SN mechanism to sign and identify on-card application assemblies. The SN and digital signature is embedded inside the card resident binary file by the .NET compiler.
The digital signature verification is performed by the card's operating system when a new application is loaded onto the smart card. We noticed that the use of a 1024 bit RSA key and the SHA-1 algorithm are no longer recommended from a security best practice viewpoint, however this is not the focus of this paper.

2.3 The Target .NET card Code and Data Security

Each assembly file on the .NET smart card will have a PBKT derived from its RSA public key, using the algorithm mentioned in section 2.2. Assembly files with the same PBKT, form code groups to which code access and file system access permissions are assigned by the card's administrator user. The PBKT also identifies on-card applications to each other, so every application can configure access lists to grant or deny access to its library files by other card applications. The target .NET card documentation describes its evidence based security model as follows: "If an on-card assembly (A1) needs to access another assembly (A2), either both assemblies must have the same public key token, or assembly A1, whose public key token is PBKT1, must be granted

access to assembly A2 by adding public key token PBKT1 as an attribute on assembly A2".

The target .NET VM creates process boundary units called "Application Domains" for every on-card application, which is sometimes referred to as an "Application Firewall", because it enforces strict data and code access controls between on-card applications. Code and data from one application domain cannot be directly accessed from within another application domain unless they are exposed via data and code sharing API.

The .NET card applications can use two types of on-card data storage: saving data as .NET objects inside assemblies and data storage on the card's file system. The application domain protects data stored inside assemblies such as remaining balance value or PIN codes from unauthorised access by other on-card applications. The developer does not need to configure any access lists for this purpose, as the application domains are managed automatically by the card's .NET VM.

In some cases, the application developers may want to store data on a card's file system instead of using application domain storage. For instance, an application that requires frequent code update can store data on the file system to separate it from the code file which may be subject to frequent change. By doing so, the application owner does not need to worry about preserving the data before a code update. File system storage can also facilitate data sharing between multiple on-card applications. Applications can directly open and read/write data files without the overhead of performing inter-domain data sharing. The file system objects are protected by evidence based security model using the PBKT that were discussed in section 2. Every file or directory on the file system has two associated permissions sets: the Public and Private Permission sets. The Public permission set defines what operations (Manage, Write or Read) every on-card application is allowed to perform on the file or directory. The Private permission set defines permissions assigned to individual public key tokens. For instance, one can assign "Read" and "Write" permissions to the code group identified with PBKT of "26272048f12bffaf".

3 Analysis of The Card Resident Binary Format

The target .NET card cannot load and execute original .NET assemblies that are compiled by Microsoft Visual Studio. Those assemblies are converted to "card resident binary" format (the target format) before being loaded to the smart card. This is a proprietary and unpublished file format developed to allow loading and execution of MSIL code on resource constrained devices such as smart cards. The target .NET card vendor has developed a plug-in for Microsoft Visual Studio which parses the PE format and .NET metadata directories of the original .NET assembly compiled by the Microsoft .NET framework SDK and creates the equivalent card resident binary in the target format. The resulting file should contain .NET metadata information as well as the digital signature and SN, which were discussed in section 2. Therefore, it is important to know the structure and format of the target binary file in order to perform an appropriate vulnerability analysis on the target .NET operating system and VM.

3.1 The Target .NET Converter

The converter plug-in for MS Visual Studio is developed using the Microsoft .NET framework, but its MSIL code is obfuscated to make code analysis more difficult. To determine if it still feasible for an attacker to analyse the converter plug-in code and understand the card binary compilation process, we chose to take a hybrid approach in analysing the card binary file format via the following steps:

1. Examination of the obfuscated code of the converter plug-in in order to understand the conversion process and how it operates on the original .NET assembly metadata and code.
2. Mapping original .NET assembly metadata to the card binary file metadata by tracing and matching the relevant .NET objects in both of them

After analysing the converter plug-in code, we located an interesting method code inside the "Converter" class. This code found and deleted two temporary files after the conversion process was completed. The MSIL code of this method was modified not to delete those temporary files, for further analysis. After compiling a simple on-card application using our modified converter plug-in, two temporary files were found in the same directory as that of the compiled application. One of the files contained the compiled card binary, however the other was a more interesting text file (which we will refer to as the map file) which listed all converted .NET namespaces, types, methods and field names from the original .NET assembly and corresponding codes. The map file contained useful information which saved lots of time during the analysis phase.

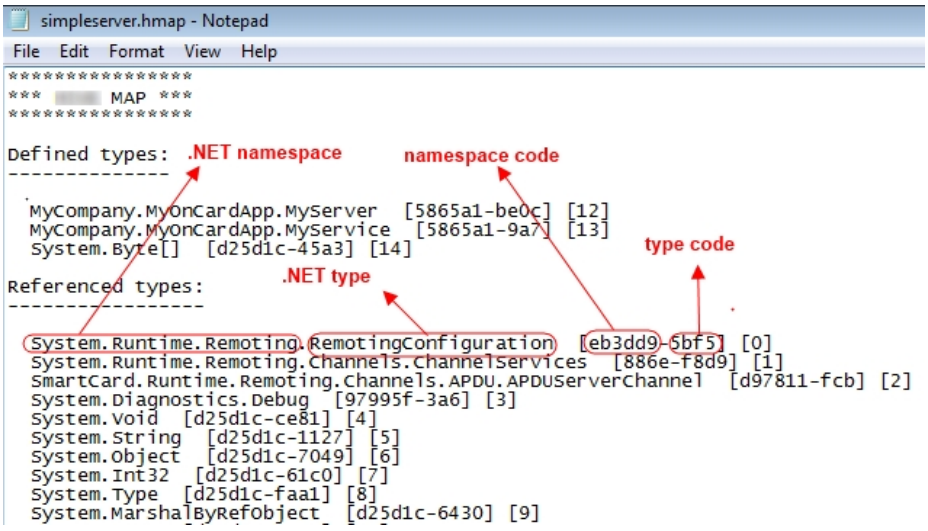

Fig. 2. – Example of a map file

3.2 Target Binary File Format

The card binary format did not have a lot in common with the Microsoft .NET assembly format, except the object reference and definition tables. An overview of the file structure is shown in Figure 3. It consisted of three fixed sized headers, four object reference tables, and three object definition tables followed by program code and RSA signature. The reference tables include the identification code of referenced namespace, type, method or fields and the number of times these were referenced in the program.

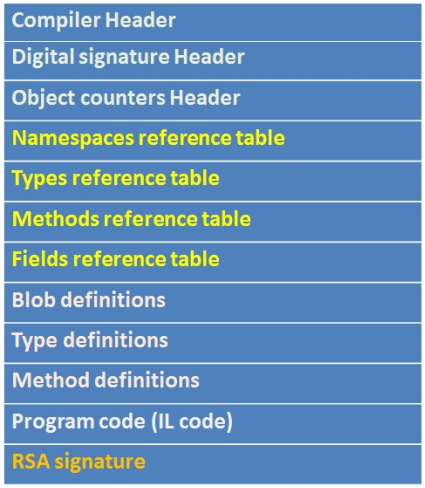

Fig. 3. Target .NET card binary structure

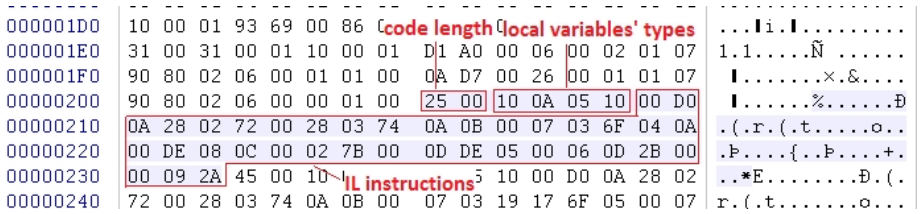

Fig. 4. – A method body definition in the target .NET card binary

Finally, the target binary file ends with the RSA digital signature which has the same size as the RSA modulus (128 bytes).

3.3 Analysis Tool Software

Analysis of the target binary format enabled us to identify and understand key metadata information such as the digital signature header, object reference and definition tables that are used by the target .NET VM to load and prepare on-card applications for execution. Manipulating these headers and metadata tables and observing the modified assembly execution, would be an effective technique to discover possible vulnerabilities

in the .NET smart card operating system or VM. However, this would require custom software capable of loading compiled card applications, allowing the user to view and modify different headers, metadata tables and code sections of the target binary and finally, re-signing the modified application with a given RSA private key so that the modified card resident binary includes a valid digital signature. We have developed such software which facilitates the visualization, manipulation and re-signing of the target .NET card applications. The software tool is written in the C# programming language and does not depend on the target .NET SDK or converter plug-in. The user interface visualizes the structure of the loaded card binary resident file and enables the user to navigate through each header, metadata table or code block and view or modify the relevant data block in the hex editor. The application automatically highlights the target binary data blocks as the user navigates though different sections. It also decodes the referenced namespace and type codes to the .NET namespace and type names, so that the user can easily make a rough idea about application functionality without reading the code section. The "Tools" menu has "Verify Signature" and "Re-Sign" options that allows the user to validate the RSA digital signature of the loaded application and sign it with a different RSA key pair.

4 .NET Smart Card Vulnerability Research

The card resident binary elements such as headers, .NET metadata tables and program code sections are directly parsed by the target .NET VM, thus they can be used as test vectors in order to trigger vulnerabilities in the target format parser engine or to provide false information such as spoofed application identities to the card's operating system. The target .NET VM and operating system is proprietary software running in a tamper resistant chip which protects the code even against invasive attack. Therefore, an attacker's strategy may be to find vulnerabilities in the card's VM by perhaps loading and executing manipulated card resident binaries and observing the VM's error codes and unexpected behaviours. This iterative testing technique of feeding random mutations of a valid input to the target program and observing the results is referred to as "Fuzz Testing" and has been used widely by security researchers for vulnerability discovery, especially in closed-source applications or systems whose internal structure and processes are undocumented or unknown. We decided to attempt "Fuzz Testing" of the target binary format for vulnerability discovery. This approach involved generating test cases by modifying a number of candidate sections in the target file instead of running a mutation engine over all sections. After building and uploading a number of test cases for both groups and observing the response from the smart card, we chose the following two sections as test candidates:

1. Digital signature header containing the application's PBKT. The PBKT is a critical value, because the evidence based security model of the .NET card is built around it.

2. Intermediate language (IL) instructions in the program code section which are used to allocate memory buffers or read and write to those buffers. Manipulating the arguments to those instructions could result in unauthorized access to the smart card RAM or EEPROM content.

The test tool software was used for modifying template target files and digitally signing the test cases. The following sections provide the details of the vulnerabilities that we have discovered by manipulating on-card application files.

4.1 Public Key Token Spoofing Vulnerability

We discussed the evidence based security model of .NET smart card in section 2 and explained the concept of the PBKT which was used to identify card applications to the .NET VM and other on-card applications. The PBKT is derived from the assembly's RSA public key by applying a SHA-1 hash algorithm as explained in section 2.2. The RSA public key is embedded inside the signature header of the application file and therefore the card's VM can compute the PBKT anytime by applying the SHA-1 hash algorithm (which is a fast cryptographic operation compared to asymmetric encryption algorithms). However, we noticed that the PBKT value was also stored in the target file signature header along with the RSA public key parameters, and these could be manipulated by the user before being uploaded to the smart card.

As .NET VM uses the PBKT to make security decisions such as granting or denying an application's access requests to file system objects, this could have critical security implications. For instance, an application (M) which has manipulated public key token value of $PBKT_M$, might be able to access data files owned by another application (A) whose public key token is $PBKT_A$. We prepared a vulnerability test scenario to find out if it is possible to upload card resident binaries with a modified PBKT to the .NET smart card. This would then allow the application to bypass the card's evidence-based security system and access unauthorized file system resources. This test involved the following steps:

1. The on-card application (M) was written to read a data file on the .NET smart card which it was not authorized to access. The target .NET VM threw "UnauthorizedAccessException" and denied the file read request which was the expected behaviour. Application (M) was digitally signed by the $PrivK_M$ and had a public key token value of $PBKT_M$. The data file it was trying to read, belonged to application (A) which was previously uploaded to the card and was signed with a different private key ($PrivK_A$) and having the public key token of $PBKT_A$. The access list of the target data file did not grant read access to $PBKT_M$ which prevented application (M) from accessing it.
2. Using the developed analysis tool software, we modified the PBKT value (8 bytes) at offset 0x52 of application (M) and change it to $PBKT_A$. Then it was signed by $PrivK_M$ key and saved to a new assembly file.
3. We loaded the manipulated application (M) to the .NET smart card successfully and the content manager service did not issue any errors or warnings. At this point, we

successfully "spoofed" the identity of the target application (A). This allowed us to gain access to its data file.

4. Once the manipulated application (M) was deployed on the card and executed, it did not receive any security exceptions and was able to read the data file of application (A). This confirmed that the target .NET VM had trusted the PBKT value embedded in the target binary to make security decisions. This value can, however, be modified by a malicious user in order to bypass the card's data security system.

Malicious applications could also exploit this vulnerability to bypass the code security system which defines code access permissions on PBKTs. For instance, suppose that application (A) was linked to a library assembly (.DLL file) and the code access permissions only allows code groups identified with $PBKT_A$ to execute the library's code. A malicious application (M) could exploit the aforementioned vulnerability to spoof the identity of application (A) and gain unauthorized code execution access to the target library. The root cause of this vulnerability was that the card manager service verified the RSA digital signature of the uploaded target file, but did not check if the PBKT value in the digital signature had been altered, by recalculating it on the card and comparing it with the value embedded in the uploaded target binary.

We demonstrated a PBKT spoofing attack against a "Password Wallet" application that was developed during our work to gain better knowledge about the target .NET SDK and the card application development process. The Password Wallet application demonstrates the use of the .NET smart card for secure storage of web sites' credentials. The user can only access the accounts by entering the correct PIN code which is a combination of alpha-numeric and uppercase characters. The accounts' usernames and passwords are stored on the card in a binary file called "store.dat" in the form of serialized .NET objects [9]. The malicious application "GrabtheWallet", was signed with a different private key to the one used by the Password Wallet and, attempted to access the accounts data file (store.dat). The request was denied by the .NET VM and "UnauthorizedAccessException" was thrown by the VM. We used our analysis tool software to change the PBKT of this application to the Password Wallet application PBKT, and re-signed it using the same key used for signing the GrabtheWallet.exe application. The manipulated application was successfully loaded onto the smart card. It had the exact same size and functionality, but had spoofed the PBKT of the Password Wallet application. This application was able to gain unauthorized access to the accounts data file (store.dat) of the password wallet application as demonstrated in the following figures:

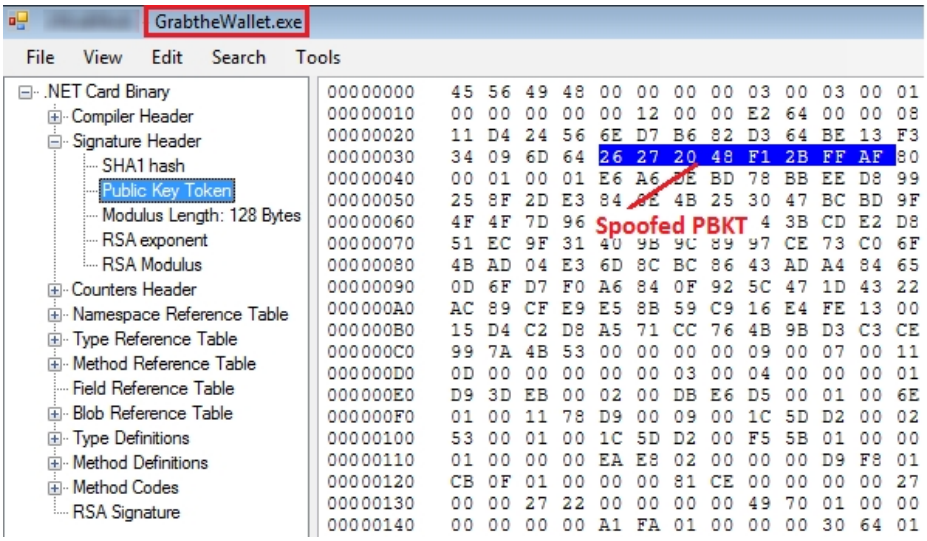

Fig. 5. Modifying the PBKT value using the analysis tool

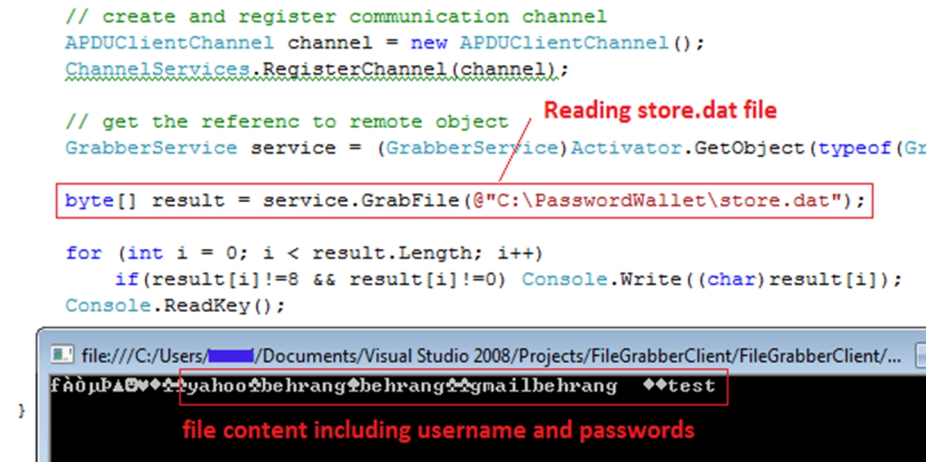

Fig. 6. Unauthorized access to PasswordWallet data file

4.2 Virtual Machine Memory Corruption Issues

The metadata tables and program code section are parsed and processed by .NET VM to load referenced objects from other assemblies, create defined objects, allocate and initialize memory buffers and control the execution flow of the loaded program. Therefore they can expose a large attack surface to malicious application code. The IL code verifier examines those metadata tables and IL code sections for safe memory access and correct program execution flow before installing the loaded application into the card's persistent memory. Therefore, a well implemented code verifier would be able

to counter malicious code threats effectively. However, implementation errors such as placing unnecessary trust on data that can be manipulated by an attacker could be exploited to cause an unsafe code to pass the verification process. We discovered such vulnerability in an internal routine of the target .NET base library (mscorlib.dll) which was used for array initialization.

The RuntimeHelpers.InitializeArray method performs fast copying of static data defined in the program code to the array objects. This method had a public access modifier, and could be called by any on-card application. It had the following declaration:

```
public static void InitializeArray(Array array,
RuntimeFieldHandle fldHandle);
```

The array argument is the empty Array object which should be initialized by the data pointed to by fldHandle. This handle value represents a field defined in the FieldDefs metadata table of the target binary which was mentioned in section 3.3. Each row of the FieldDefs table has the following columns:

```
[Field id][Field modifiers][TypeRefs or BlobDefs index]
```

If the field modifier contains the hasRVA (Relative Virtual Address) modifier, then the field refers to a virtual memory location. A virtual memory address is relative to the program load base address, which is the location that the program executable file is loaded to by the .NET VM into the RAM memory. For instance a field's virtual relative address could be 0x400 and the program could be loaded to the physical memory address of 0x10000000. The physical address of the field would be 0x10000400. Usually the RVA is relative to the program base load address, but it can also be relative to other sections' base load addresses in the target file. In this case, the base address was staring address of the blobs definition table. It means that the index column will point to a row in the blobs definition table that contains the data by which the array should be initialized. Runtime field handles can be obtained using the ldtoken (load token) IL instruction and providing the field index number in the field definition table. The following IL pseudo code demonstrates this instruction:

```
newarray <0x03,byte>  //create an empty byte array of
3 bytes length and push its address to the stack
ldtoken <1>   //convert the FieldDefs table token of
row number 1 and push the handle to the stack
call InitializeArray //call array initialization rou-
tine to copy data pointed by field number one to the
array
```

We found that the InitializeArray method did not check if the field handle passed in the second argument points to a field with the hasRVA modifier that refers to a relative memory address in the program's address space. As a result, it was possible to point the fldHandle to types without the hasRVA modifier set in order to copy the memory content of the VM internal structures into a user defined byte array. Some limited practical attempts to exploit this vulnerability to access smart card memory content were not successful and resulted in damage to the smart card operating system.

We also found a second potential vulnerability in the array initialization routine which could result in violation of the type safety of the .NET VM. The root cause of this vulnerability was that the InitializeArray method did not perform a security check to ensure that the provided array argument was of a primitive type (integer, byte, char and boolean types). Primitive types are data types that are commonly used by the programmers and already exist in the .NET base class library where the non-primitive types are defined by the programmer and contain memory references to the user defined objects in the memory. The target .NET VM supports both primitive and non-primitive type arrays. The elements of a non-primitive array are automatically initialized to null after allocation and the elements are later assigned with the memory references to the user-defined objects. If those object references can be manipulated by a malicious code, then it would be possible for an attacker to access arbitrary memory locations and violate the type safety of the VM.

If the array argument supplied to the InitializeArray method was of a non-primitive type such as an array of structures, then it was possible to overwrite the memory references in that array with the arbitrary data pointed to by the fldHandle argument. Unlike the pervious potential vulnerability in InitializeArray, exploiting this issue didn't result in damage to the smart card and as the code verification process was not completed successfully, the malicious application couldn't be loaded to the smart card. This limited the impact of the potential vulnerability exploitation to low.

5 Countermeasures

The public key token "spoofing" vulnerability could be mitigated by adding the required PBKT validation routine to the card manager service. This routine should recompute the PBKT from the RSA public key embedded in the uploaded target binary and compare it with the PBKT value in the digital signature header. If the values do not match, the target file has been manipulated and must be rejected. The vulnerabilities in the array initialization routine could be addressed by implementing the following two security checks:

1. The field pointed to by the fldHandle argument must have the hasRVA modifier set.
2. The array elements should not be of reference or non-primitive types.

The above security tests could be expressed using the following C pseudo-codes:

```
//Security checks before initializing the array
if (!(fldHHandle->type->modifier &
FIELD_MODIFIER_HAS_FIELD_RVA)) {
 throw new Exception("field does not have hasRVA modi-
fier!")
}
Type type= get_array_element_type(array);
if ( IS_REFERENCE_TYPE(type) || !IS_PRIMITIVE(type)) {
throw new Exception("cannot initialize array of refer-
ence or non-primitive types");
}
```

The vendor has developed a fix for the PBKT spoofing vulnerability and provided the following risk assessment: "To exploit this vulnerability, an attacker must be able to upload his malicious application on the card, and also get knowledge of the Public Key Token of the targeted application to prepare his malicious application first. To do so, he must gain administrator privilege; this is quite a strong requirement. The Administrator Key is normally set up by a Card Management System (CMS) using strong diversification algorithms based on a Master Key securely stored in a HSM or a smart card based controller. Also, the targeted application must use private file-system storage for its data to be exposed. Therefore, internal (Application Domain) storage is immune to such attack".

"The discovered memory corruption issues in section 4.2 could not be exploited to gain unauthorised access to VM memory or execute arbitrary code and only resulted in denial of service through damaging the card operating system. To exploit those vulnerabilities, an attacker must be able to upload his malicious application on the card. To do so, he must gain administrator privilege; this is quite a strong requirement".

6 Conclusion

This paper records research work investigating potential security vulnerabilities in a commercial .NET smart card product. We developed and presented a .NET card binary analysis tool which was used to discover vulnerabilities in the .NET card VM. The pre-publication findings from this work were shared with the .NET card vendor so that appropriate countermeasures could be taken in a timely manner. In summary, a demonstrable vulnerability was found in the use of public key tokens that could allow malicious applications to have unauthorised access to files and library functions of legitimate applications. However, an attacker would need to have administrator privileges to exploit the vulnerability. The developed test tool can also enable the legitimate manager of a .NET smart card to detect the PBKT spoofing attack or decompile and analysis the card applications developed by third parties in order to discover possible vulnerabilities or backdoors.

References

1. Multos International. Multos Technology. [Online]. http://www.multos.com/technology/
2. Sun Microsystems. Jaca Card Technology. [Online]. http://www.oracle.com/technetwork/java/javacard/overview/index.html
3. Microsoft IT forum 2004..NET-based Smart Cards. [Online]. http://www.prnewswire.com/news-releases/hive-minded-delivers-net-based-smart-cards-75449172.html
4. M. Witteman, "java Card Security," *Information Security Bulletin*, pp. 291-298, October 2003.
5. TippingPoint. (2011, October) Zero Day Initiative, Oracle Java IIOP Deserialization Type Confusion Remote Code Execution Vulnerability. [Online]. http://www.zerodayinitiative.com/advisories/ZDI-11-306/
6. J. Hogenboom and W. Mostowski , "Full Memory Read Attack on a Java Card," Department of Computing Science, Radboud University Nijmegen, 2009.
7. J. Iguchi-Cartigny and J. L. Lanet, "Developing Trojan applets in a smart card," *Journal in Computer Virology*, vol. 6, no. 4, pp. 343-351, November 2010.
8. Microsoft..NET Framework Security. [Online]. http://msdn.microsoft.com/en-us/library/aa720329%28v=vs.71%29.aspx
9. Microsoft. Object Serialization in the.NET Framework. [Online]. http://msdn.microsoft.com/en-us/library/ms973893.aspx