Paper

Cryptographic Protection of Removable Media with a USB Interface for Secure Workstation for Special Applications

Jan Chudzikiewicz and Janusz Furtak

Military University of Technology, Warsaw, Poland

Abstract—This paper describes one of the essential elements of Secure Workstation for Special Applications (SWSA) to cryptographic protection of removable storage devices with USB interface. SWSA is a system designed to process data classified to different security domains in which the multi-level security is used. The described method for protecting data on removable Flash RAM protects data against unauthorized access in systems processing the data, belonging to different security domains (with different classification levels) in which channel the flow of data must be strictly controlled. Only user authenticated by the SWSA can use the removable medium in the system, and the data stored on such media can be read only by an authorized user by the SWSA. This solution uses both symmetric and asymmetric encryption algorithms. The following procedures are presented: creating protected a file (encryption), generating signatures for the file and reading (decryption) the file. Selected elements of the protection systems implementation of removable Flash RAM and the mechanisms used in implementation the Windows have been described.

Keywords—filter driver, removable media protection, symmetric and asymmetric encryption.

1. Introduction

Nowadays, the most comfortable Removable large-capacity data devices are connected to the system via the bus Universal Serial Bus (USB). Such devices include flash memory RAM with a capacity of several tens of GB and hard disk drives with a capacity of several TB. The popularity of these devices forces the need for mechanisms to ensure an adequate level of protection of data stored on them. This is important in the case of sensitive data which have a significant impact on the safety of the institution. This fact is particularly important in the systems where confidential data are processed. It is hard to imagine a contemporary computer system in which the data storage devices cooperating with the system through the USB bus are not available. This observation also applies to Secure Workstation for Special Applications[1] (SWSA) [1], [2].

[1]SWSA is a computer system in which multi-level protection mechanisms have been implemented. In this system, the stored and processed data (objects) are classified due to the required level of security. Users of the system (subjects) have specific authorization to work with classified data. In order to ensure confidentiality and integrity of data for subjects are used mechanisms of mandatory access control to objects.

In ordinary systems to protect data on the media Flash RAM, the most commonly used software (e.g., USB Flash Security, Secure Traveler, Rohos Mini Drive, etc.) must be installed on the media prior to its use. During the installation of such software, in Flash RAM is created an encrypted volume which is accessed by using then defined password. The power of safeguard of the medium using this type of software depends on the used symmetric encryption algorithm and key length. This type of security is sufficient in the case of a loss or theft of the media. The use of such solutions in systems with multilevel security (MLS) which include the SWSA is insufficient for the following reasons:

– system does not provide the possibility of the control of an access to data copied to such a secured removable media, in particular, an entity with the lower clause cannot be blocked by system, while trying to read to the data with a higher classification contained on the medium;

– transfer of data between different entities that use such media possesses problems arising mainly from the need to provide the transfer of the medium and an encryption key with help which the medium has been encrypted;

– entity that creates a copy of the data on the media does not assure that the data are only available for the appropriate recipient and the recipient does not have an assurance that data is received from the expected sender.

The article presents a solution enabling to such a preparation of data stored in Flash RAM, so that the recording medium can be used to secure the transfer of data files, during which the sender of data (i.e., the creator of the protected media content) is assured that data will be available only for designated recipient, and the recipient is assured that the received data comes from the expected sender. The described mechanism uses both symmetric and asymmetric encryption algorithms and asymmetric. The presented solution uses a filter driver [3]–[5].

In this solution, it is assumed that in terms of operating system data can be processed in two directions: from plain

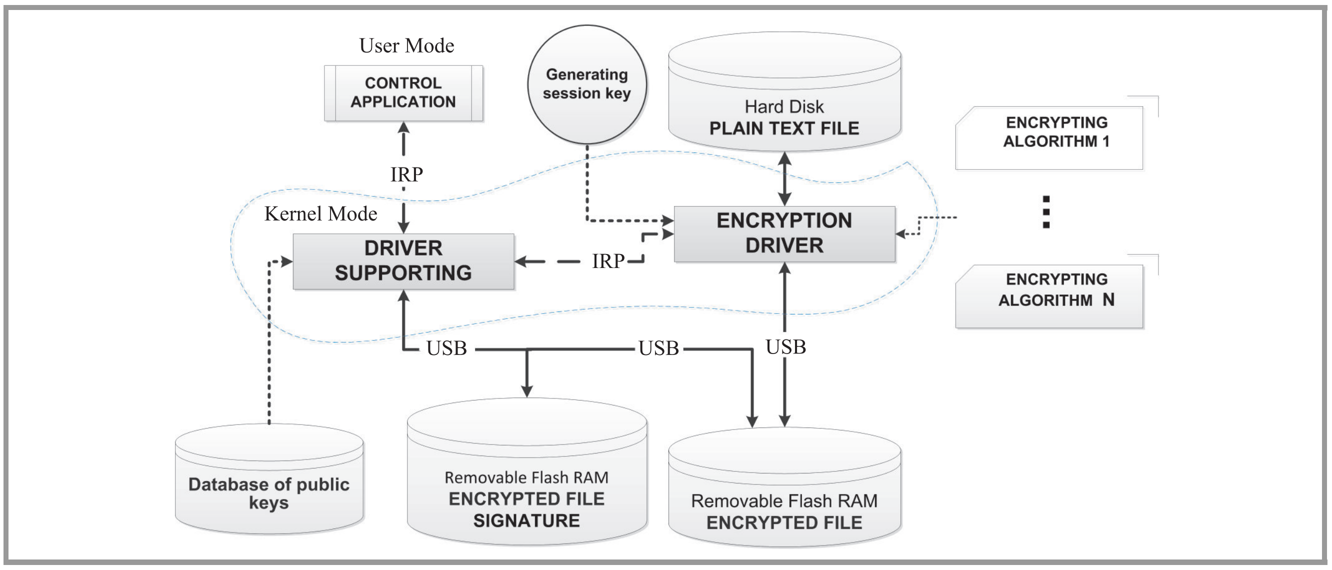


Fig. 1. Schematic diagram of securing data stored on removable media.

text form stored on your hard disk to secure form on removable media (e.g., Flash RAM, hard drive) connected to the system via the USB bus and, conversely, from secure form on removable media to plain text form on the hard drive. Do not allow the possibility of using the software for the direct exchange of data files between removable media (e.g., flash memory) connected to the system via the USB bus.

The process of securing the data exchange using removable media should satisfy the following functional requirements:

- it should be implemented in a manner transparent to the user,
- it should not cause any noticeable loads of the operating system to the user,
- it should not have a significant impact on the speed of read and write data onto data media,
- it should allow the use of various encryption algorithms to ensure the required level of confidentiality,
- it should be built on removable media, protected file and the signature for this file (signature should contain securely stored data necessary to encrypt/decrypt the protected file, and to ensure the integrity of the file,
- it should allow to perform any operation allowed for data storage media, such as volume, surface checking for errors, and defragment the disk-based data.

These requirements force the use of the process of securing the data separate modules (drivers), operating at the kernel-level of operating system [4]–[7]. Schematic representation of a solution being developed in the environment of Windows systems family is shown in Fig. 1.

Described solution is available for a user of the secure workstation through the control application (CApp). The main elements of the built system are interacting drivers: encryption driver [3] and driver supporting, which are compatible to the Windows Driver Model [6]. Both elements work in kernel mode, operating system and communicate with each other using the internal mechanisms of the operating system (in the figure they are labeled as IRP) [6], [8]. These mechanisms are described in details in Section 2.

The purpose of the encryption driver (EnD) is the realization of the process of encryption/decryption of data and determination of its hash value for these data. The driver supporting (DSu) sets the signature for protected data (according to the algorithm presented in Section 3) and mediates the transfer of messages/commands between EnD and CApp. The other components of the system are: the .DLL library that provides the functionality of the implemented encryption algorithms, module of generation of session key, and the database of public keys of users.

The product of the process securing the original file consists of two files: a file with encrypted data and the file containing the signature for the encrypted file. Both files can be stored on one medium or each file on a separate medium. Choosing a storage location of the signature file is defined by the user through CApp. It should be noted that saving the encrypted file and the signature file on separate media increases the security of stored data, but it is inconvenient to use.

2. Filter Drivers in Windows

Construction of Class Windows operating system assumes the use of two modes: user mode and kernel mode [6], [8]. Architecture of such a system is shown in Fig. 1. Modular design allows for an easy expansion of system functions (which is clearly visible in Fig. 2, showing the components operating in kernel mode), while the use of hardware ab-

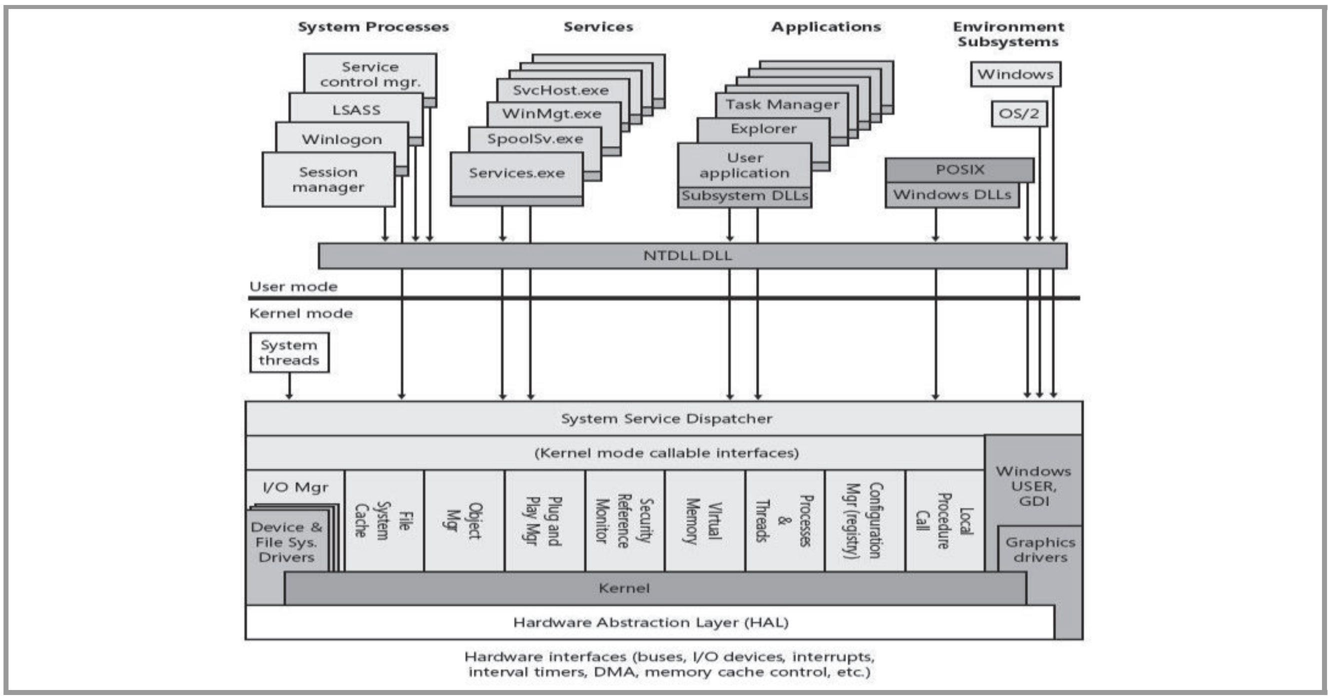


Fig. 2. Architecture of operating systems Windows [6].

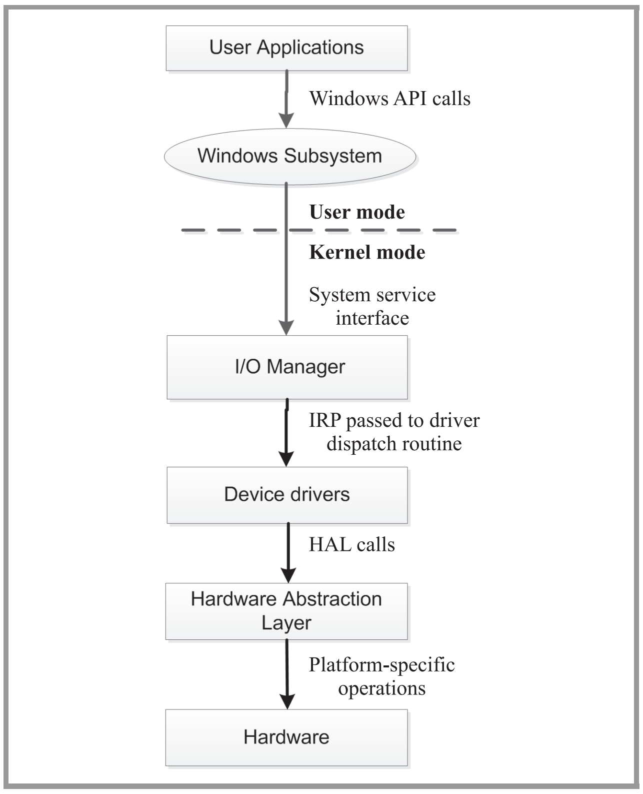


Fig. 3. Process of communication between user application and the driver [6].

straction layer (HAL) provides portability between different hardware architectures.

From the point of view the task of securing the content of removable storage media in the SWSA and the drivers of these devices (in the part relating to user mode) are relevant only the following components:

– user application (the CApp in the solution which is presented in this paper);

– Environment Subsystem (Windows);

– ntdll.dll system library that allows a communication with the elements working in kernel mode.

Environment subsystems form a working environment for applications running on them. It translates the application call to the system and its resources to the primary functions of Windows. The process of communication between user application and driver uses packets IRP that are created

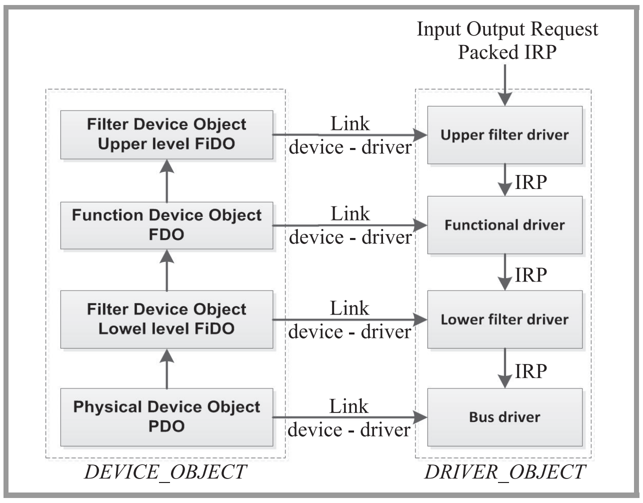


Fig. 4. Windows Driver Model [6].

by the I/O Manager on the basis of the generated request type. The process of handling requests from the user's application in which the IRP packet is generated is shown in Fig. 3.

The Windows Driver Model assumes that the devices are controlled by a stack of drivers working together, each of which is responsible for the implementation of other tasks of the device. The driver stack model is shown in Fig. 4. In this model, there are always two drivers: a bus driver at the bottom of the stack and the functional driver which defines the utility functions of the device. The model allows for the possibility of using additional filter drivers that are placed in the stack and allow you to monitor and modify the contents of packages I/O requests directed to device. DEVICE_OBJECT is a representative of a particular device, such as flash RAM, and is associated with a driver (DRIVER_OBJECT) that supports it.

Driver objects are created by the I/O Manager when the driver is loaded. Drivers are responsible for creation of

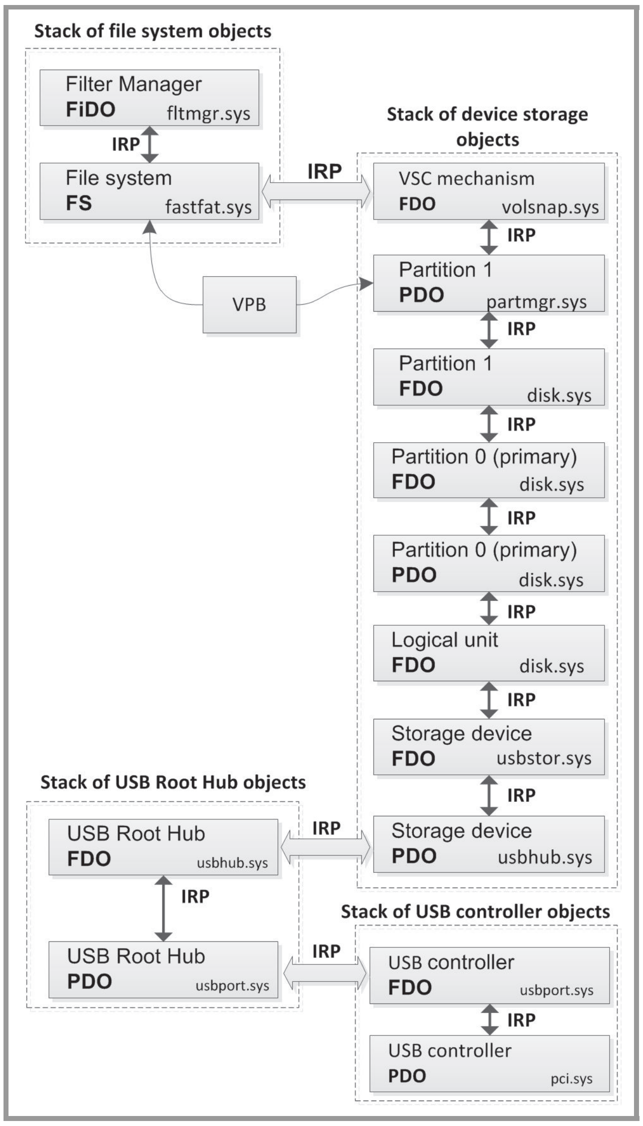

Stack of file system objects
Filter Manager FiDO fltmgr.sys
IRP
File system FS fastfat.sys
IRP
Stack of device storage objects
VSC mechanism FDO volsnap.sys
VPB
Partition 1 PDO partmgr.sys
Partition 1 FDO disk.sys
Partition 0 (primary) FDO disk.sys
Partition 0 (primary) PDO disk.sys
Logical unit FDO disk.sys
Storage device FDO usbstor.sys
Stack of USB Root Hub objects
USB Root Hub FDO usbhub.sys
Storage device PDO usbhub.sys
USB Root Hub PDO usbport.sys
Stack of USB controller objects
USB controller FDO usbport.sys
USB controller PDO pci.sys

Fig. 5. Objects stack for the storage device connected to the system via the USB [3].

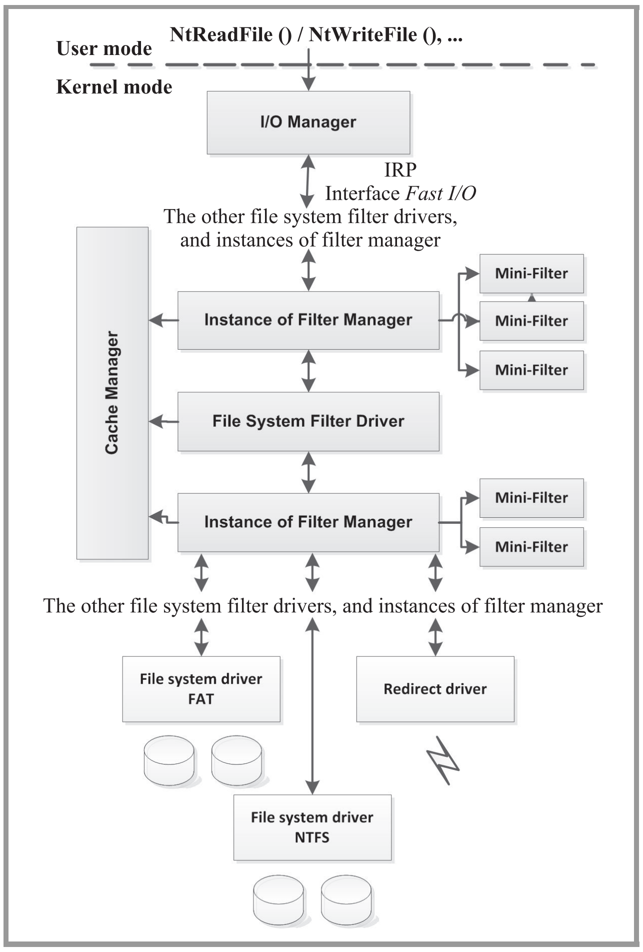

NtReadFile () / NtWriteFile (), ...
User mode
Kernel mode
I/O Manager
IRP
Interface Fast I/O
The other file system filter drivers, and instances of filter manager
Cache Manager
Instance of Filter Manager
Mini-Filter
Mini-Filter
Mini-Filter
File System Filter Driver
Instance of Filter Manager
Mini-Filter
Mini-Filter
The other file system filter drivers, and instances of filter manager
File system driver FAT
Redirect driver
File system driver NTFS

Fig. 6. The drivers stack for data storage devices using mechanisms of Filter Manager [4].

the DEVICE_OBJECT representing the system devices. Creating an object of this type occurs when the AddDevice procedure is called by the I/O Manager. Depending on the role of the driver, created object can represent:

– Physical Device Object (PDO) – representing the connection between the device and the bus;

– Functional Device Object (FDO) – functional driver uses it to determine functions of the device;

– Filter Device Object (FiDO) – filter driver uses it to process data from the packet I/O requests, and in particular their encryption, which is used in the described solution.

The I/O Manager can start sending packets I/O requests to the supported device after all DEVICE_OBJECT objects will be created.

Implementation of the described drivers model for the storage device connected to the system via the USB bus is more complex. Figure 5 shows objects stack associated with drivers of such a device. The presented diagram shows that on the top of the objects stack there is an object of filters manager. Below are the objects corresponding to the FAT32 file system (marked on the diagram as an object of type FS). An attention should be paid to the file system drivers that are not explicitly included in the driver stack. The association between driver of file system FS and object representing a partition in the storage device is executed by a data structure Volume Parameters Block (VPB).

The presented solution uses a kernel component called the Filter Manager. Filter Manager performs a significant part of the tasks. Otherwise it would have to be performed by the filter drivers. As a result, using the filter manager (called mini-filters) is simpler and easier to implement. Filter Manager is available in all systems ranging from Windows Server 2003 to Windows XP with Service Pack 2. Figure 6 shows the drivers stack for data storage devices using mechanisms of Filter Manager.

3. The Process of Creating and Reading a Protected File

In the process of creating a protected file on removable flash memory (that is a creating an encrypted file and the signature of this file), and reading (decrypting) the file from the removable flash memory are the necessary attributes of the user who created the protected file (this user will be called the sender), and user for whom the protected file was created (this user will be called the recipient). When creating a protected file, sender is role plays a user logged into the system and he specifies a file recipient using CApp (you can select one from among the users who meet the requirements of SWSA closely related to the multilevel security of system). When reading a protected file with the use of CApp, the logged user plays the recipient role, and the sender attributes are read from signature file after the successful decryption of this file, using the private key of the logged on user. Permissible is a situation in which the logged user is simultaneously the sender and the recipient of data.

The process of creating a protected file includes the step of encryption, and then creating a signature for that file. However, during the process of reading a protected file in a first step, the attributes needed for decrypting this file are obtained from the signature. In the second step, the file is decrypted.

3.1. Creating a Protected File

The process of writing the file, including file encryption and hash generation is performed by the EnD. Operation of EnD has been presented in [3]. The diagram describing the process of writing the file is shown in Fig. 7. Dashed line in that figure indicates operations implemented by the EnD. During the process of file encryption, the value of the hash function is determined to ensure the integrity of the file.

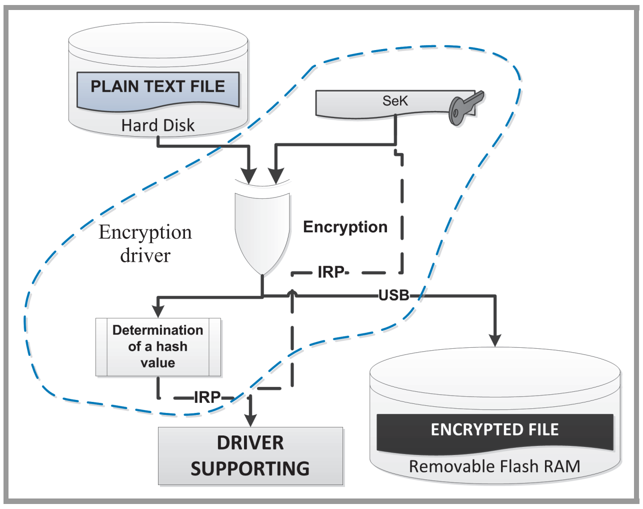


Fig. 7. The process of writing data to removable flash memory.

The determined value of the hash function and the generated session key after completion of record are transferred to the DSu in order to generate a signature for the stored data. The process transferring of the hash function value and the session key transferring is implemented using the system mechanisms marked in Fig. 7, as the IRP.

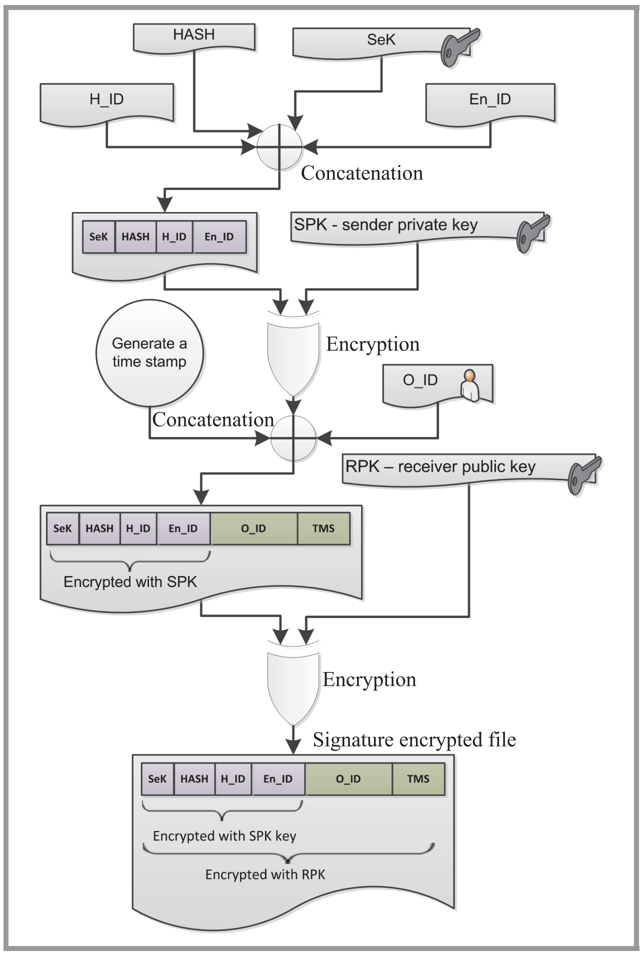


Fig. 8. The algorithm of signature generation for protected file.

3.2. Determining the Signature

For each of the protected file the signature is generated which contains the information needed to read this file. Signature of the file contains the following fields:

- SeK – random key to encrypt/decrypt the secure file,
- HASH – value of hash function which is determined on the basis on the content of protected file after encrypting this file,
- H_ID – identifier of the algorithm used to generate the hash,
- En_ID – identifier of the algorithm used to encrypting,
- O_ID – identifier of the logged user (the sender) who initiated the operation of data write – this identifier is required to determine the public key of sender when the file is read,
- TMS – time stamp of file creation – this value corresponds to the date of file creation.

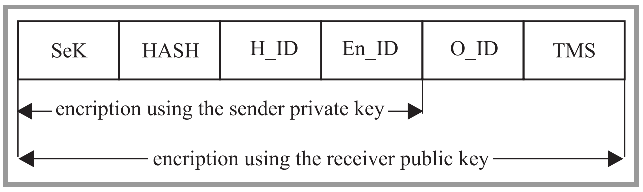

SeK
HASH
H_ID
En_ID
O_ID
TMS
encription using the sender private key
encription using the receiver public key

Fig. 9. The structure of signature secure file.

The process of signature creation proceeds according to the diagram shown in Fig. 8, and the structure of signature is shown in Fig. 9.

3.3. Reading a Protected File

The process of reading of the file requires that the signature to be read before and then decrypted. These activities are performed by the logged user (recipient of file) using CApp. The process starts with decrypting the signature file using the private key of the logged user, then reading time stamp and user identifier (O_ID) which assumed the role the sender creating a protected file. The time stamp protects the encrypted file before moving it to another medium that it was originally written on. Incompatibility of date and time stored in the time stamp and date and time, when the file was created, causes displaying the message and terminating the procedure of file reading. Along compatibility of the parameters, the next part of the signature is decrypting using the user public key of which identifier (O_ID) has been read. The next steps of file decoding are schematically shown in Fig. 10.

In Fig. 10, the operations performed by the EnD are marked using thick dashed line, and the operations performed by the DSu are marked using the thinner line (two dots dash).

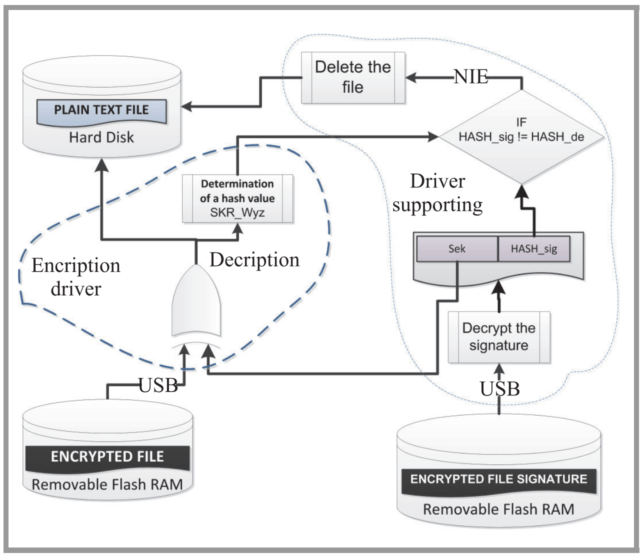


Fig. 10. The process of data reading from an external file.

During the data reading, the value of hash function (HASH_de) is determined. If the value HASH_de is different from the values obtained from the signature (HASH_sig), a message is displayed and the decrypted file, which was saved on hard disk, is being deleted.

4. Implementation Mechanisms of Encryption/Decryption

This section presents the implementation of selected elements of drivers EnD and DSu. In particular, it describes the pieces of code relating to the registration of the EnD driver in the driver stack of the file system, and initiate a logical device by the DSu driver and its associated data structure that stores the signature components of the protected file.

4.1. Implementation the Selected Elements of Encryption driver (EnD)

The driver registration process in the stack of file system drivers is realized when loading it into the system by the I/O Manager. The I/O Manager calls the *DriverEntry* driver function, which will be the entry point to the driver [6]. In this function, as the first step shown in Fig. 11, *Lookaside* list of type is initialized, in which there are held objects representing the context of substitution data buffers.

In the next step, the functions made available by the filter manager are called [4], [6]. First, the driver is registered in the system by these functions and then filtering process is launched (Fig. 12). After registering a mini-filter using *FltRegisterFilter* function, the filter manager takes over the management of the filter.

```
. . .
   ExInitializeNPagedLookasideList(&BufferSwapCo
      ntextList, NULL, NULL, 0,
      sizeof(BUFFER_SWAP_CONTEXT),
   BUFFER_SWAP_CONTEXT_TAG, 0);
. . .
```

Fig. 11. Initializing a *Lookaside* list.

```
. . .
   Status = FltRegisterFilter(IN DriverObject, IN
      &FilterRegistration,
      OUT &KMSECData.FilterHandle);
   Status = FltStartFiltering(IN
      KMSECData.FilterHandle);
 . . .
```

Fig. 12. The filter registering and activating filtering process.

From this moment, a mini-filter captures all messages sent to the functional driver of file system. One of these messages may be a message about trying to mount a new volume. Then *InstanceSetup* function is called. This function checks whether there is a mounted volume on a storage device connected via USB and includes the correct file system type. If these conditions are met, the volume is connected and its context[2] is initialized. Otherwise, to the filter manager is returned the STATUS_FLT_DO_NOT_ATTACH status code, which block the connection of the mini-filter to the volume. A piece of code referring to the previous steps is shown in Fig. 13.

```
. . .
   if ((Flags !=
      FLTFL_INSTANCE_SETUP_MANUAL_ATTACHMENT) &&
          ((VolumeDeviceType ==
      FILE_DEVICE_DISK_FILE_SYSTEM) &&
        (VolumeFilesystemType ==
FLT_FSTYPE_FAT)))
   {
       if (KMSECIsUSBDevice(FltObjects-
       >Volume))
       {
            . . .
       }
   }
   else
   {
       Status = STATUS_FLT_DO_NOT_ATTACH;
   }
. . .
```

Fig. 13. Verification of volume data.

Initiating the volume context requires allocating the necessary quantity of non-paged memory, which is done by calling the *FltAllocateContext* function. After the correct memory allocation, this function returns STATUS_SUCCESS. One should pay attention to the need to verify this value. A piece of code executing this operation is shown in Fig. 14. If one omits this check, in case of failure of memory allocation can cause an unstable system.

[2]The volume context is defined by the volume status, which is described in a data structure containing the properties of a volume. In this structure the basic data about a mounted mass storage device are described.

```
. . .
   Status = FltAllocateContext(FltObjects->
      Filter, FLT_VOLUME_CONTEXT,
      sizeof(VOLUME_CONTEXT), NonPagedPool,
      &pVolumeContext);
   if (!NT_SUCCESS(Status))
   {
      leave;
   }
. . .
```

Fig. 14. Appointment of the volume context.

```
. . .
   Status = FltGetVolumeProperties(FltObjects->
      Volume, &VolumeProperties,
      sizeof(VolumeProperties),
      &LengthReturned);
   . . .
   pVolumeContext->SectorSize =
      max(MIN_SECTOR_SIZE,
      VolumeProperties.SectorSize);
. . .
```

Fig. 15. Setting the volume sector size.

For correct implementation of the operation encryption/decryption the data included in the volume context is necessary. These data can be downloaded each time during encryption/decryption from the volume context, but the storage of this data in the driver structure accelerates the process of encryption/decryption. Therefore, the recommended solution is a single download of this data by using function *FltGetVolumeProperties*, as it is shown in Fig. 15.

4.2. Implementation the Selected Elements of Supporting Driver (DSu)

In the structure of DSu driver are used drivers marked as LEGACY DRIVER that runs in kernel mode. This solution has been accepted because the driver doesn't need to create the device objects dynamically. This driver, due to the need for communication with other components of the system, creates one logical device object which is also used to store base elements of signature. At the moment of loading, the driver in the system is being called by I/O Manager the *DriverEntry* that has a driver function which in the first step initiates an *MajorFunction* array. These functions will be handled by I/O's request addressed to the driver. A piece of code which implements an operation of MajorFunction array initialization is shown in Fig. 16.
Then in the *DriverEntry* function is being called *IoCreateDevice* function which creates a device object in the non-paged memory area. Device object is not related to any physical device occurring in system, but is used only

```
. . .
for (i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; i++)
   DriverObject->MajorFunction[i] =
       USB_WSP_LEGDefaultHandler;
   DriverObject->MajorFunction[IRP_MJ_CREATE] =
       USB_WSP_LEGCreateClose;
   DriverObject->MajorFunction[IRP_MJ_CLOSE] =
       USB_WSP_LEGCreateClose;
   DriverObject->DriverUnload =
       USB_WSP_LEGUnload;
   DriverObject->MajorFunction[IRP_MJ_WRITE] =
       USB_WSP_LEGWrite;
   DriverObject->MajorFunction[IRP_MJ_READ] =
       USB_WSP_LEGRead;
   DriverObject->
       MajorFunction[IRP_MJ_DEVICE_CONTROL] =
       USB_WSP_LEGIoControl;
   DriverObject->MajorFunction[IRP_MJ_SHUTDOWN] =
       USB_WSP_LEGShutdown;
. . .
```

Fig. 16. Initializing the *MajorFunction* array.

```
. . .
   RtlInitUnicodeString(&DeviceName,L"\\Device\\U
   SB_WSP_LEG");
   RtlInitUnicodeString(&Win32Device,L"\\DosDevic
   es\\USB_WSP_LEG0");
   status = IoCreateDevice( pDriverObject,
              sizeof(DEVICE_EXTENSION),
              &DeviceName, FILE_DEVICE_UNKNOWN,
              0, TRUE, &pDevObj );
   if (!NT_SUCCESS(status))
          return status;
   if (!pDevObj)
          return STATUS_UNEXPECTED_IO_ERROR;

. . .
```

Fig. 17. Initializing the device object.

as an element that stores necessary data to generate the signature. Figure 17 shows a piece of code responsible for creating the device object.

```
. . .
typedef struct _DEVICE_EXTENSION
{
  PDEVICE_OBJECT pDevice;// pointer to the device object linked
                         // with the that structure

  UNICODE_STRING ustrSymLinkName;// the internal name of
                                 //the device
  PVOID Hash; // pointer to the hash value for the encrypted file
  PVIOD SessionKey; // pointer to the session key
  PVIOD IDCryptAlg; // pointer to the identifier of the encryption
                    //algorithm
  PVIOD IDHashAlg; // pointer to the ID of hash generation
                   //algorithm
  PVOID IDOperator; //pointer to the current user/ID
  PVOID IDReciver; // pointer to the recipient ID
  PVOID TimeStamp; // pointer to a time stamp
  PVOID deviceBuffer; // pointer to the final signature
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
. . .
```

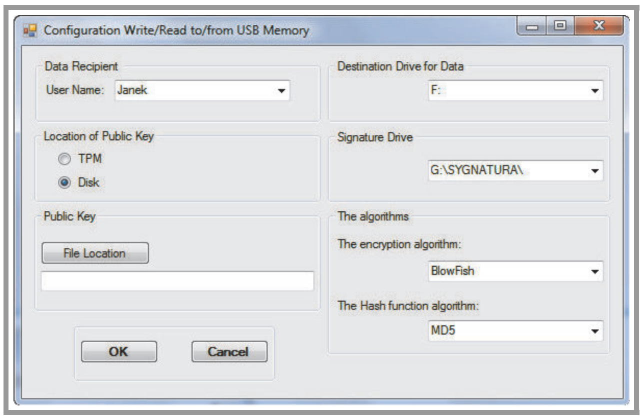
Fig. 18. Definition of the extension structure.

IoCreateDevice function call is preceded by a definition of the internal name (which is functioning on the kernel mode of system) and the symbolic name used as the references to the driver from user mode. Verification of correctness of the device object creation is realized by checking whether the value of status returned by the *IoCreateDevice* function is equal to the value of pointer to the device object – *pDevObj*. Validation of creating a device object is achieved by checking whether the status value returned by the function *IoCreateDevice* is consistent with the value of a pointer to the device object – *pDevObj*. If the returned value of status will be different from STATUS_SUCCESS the function terminate working and returns the error code of input/output to the parent function. After successful creation of the device object, the function returns the handle to it, which will be then used to realize the later references to this device. Then the initialization process of the extension structure is realized which definition is shown in Fig. 18.

A communication between the encryption driver (EnD) and the driver supporting (DSu) is always initiated by the EnD driver. The type of request directed to the driver supporting depends on the direction of copying data, which is initiated by the currently logged user on the system.

4.3. Handling for Creating and Reading a Secure File

Logged user (file sender) configures the parameters of the process of creating and reading a protected file using CApp which window is shown in Figs. 19 and 20, respectively.



Configuration Write/Read to/from USB Memory

Data Recipient
User Name: Janek

Destination Drive for Data
F:

Location of Public Key
TPM
[✓ Disk]

Signature Drive
G:\SYGNATURA\

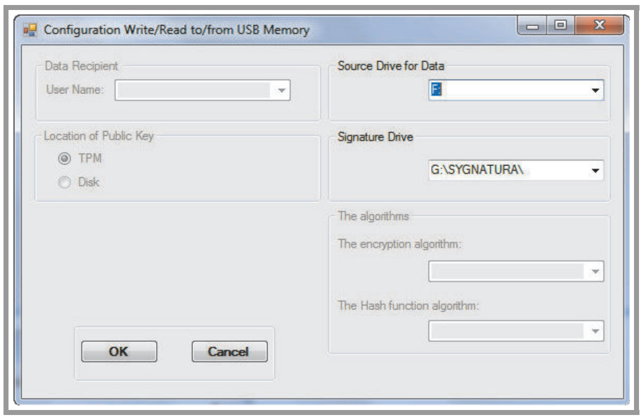
Public Key
File Location

The algorithms
The encryption algorithm: BlowFish
The Hash function algorithm: MD5

Fig. 19. The window of control application CApp for write data.

First of all, the process of creating protected file requires connection one or two (depending on where the file with the signature will be stored) removable Flash RAM memories to a computer through USB interface. The devices are automatically detected by EnD which transmits information about them via the DSu to CApp.

The logged user should determine the parameters required for encrypting the file and generating a signature. He does this by selecting (see the Fig. 19):


Configuration Write/Read to/from USB Memory

Data Recipient
User Name:

Location of Public Key
[✓ TPM] Disk

Source Drive for Data
F:

Signature Drive
G:\SYGNATURA\

The algorithms
The encryption algorithm:
The Hash function algorithm:

OK Cancel

Fig. 20. The window of control application CApp for read data.

- drive in which the protected file will be stored ("Data Drive" field),
- drive and path to the directory in which the file with the signature will be stored ("Signature Drive" field),
- identifier for the algorithm used to encrypt ("The encryption algorithm" field),
- identifier for the algorithm used to generate hash value for the protected file ("The Hash function algorithm" field),
- identifier for user (receiver) encrypted data ("Data Recipient" field),
- location of public key data recipient file ("Location of Public Key" field).

Identifier (O_ID) and the private key of the sender (the elements required to generate the signature) are automatically retrieved from the system. After determining the data configuration, logged user can begin the process of copying the file using, e.g., Windows Explorer. The name of a file which stores the signature will be concatenation of the name of protected file and string "SIG". The process of creating a file with the signature is started after the encryption process is finished and, just as the encryption process, it is invisible to the user. When the next file for the same recipient is being encrypted it does not need to change the configuration data unless the other parameters (that is the identifier of encryption algorithm or identifier of algorithm generating of hash value) will be changed. Always for the next file, a new session key will be automatically generated.
The process of reading protected file requires a connection to a computer through USB interface one or two (depending on where is stored the file with the signature) removable Flash RAM memories. The devices are automatically detected by EnD which transmit information about them via the DSu to CApp. The logged user (recipient of the data) has to specify the drive using CApp on which encrypted file is stored and indicate the file with the signature corresponding to the encrypted file. He accomplishes this by selecting (see the Fig. 20):

- drive on which is stored the protected file ("Data Drive" field),
- drive and path to the directory on which is stored the file with the signature ("Signature Drive" field).

Other parameters required to decrypt the file are determined based on the signature. After initializing by the logged user, the process of copying a file EnD sends to the DSu the name of the copied file and pauses the copy process to the moment when are receives data required to decrypt the file (that is the identifier of encryption algorithm, session key and identifier of algorithm generating of hash value). Based on submitted by the EnD the name of encrypted file, DSu identifies a file containing the signature and performs the process of signature decryption and reading the configuration data. Then performs the verification process read out TMS with the date and time of the creation of an encrypted file. In the case of inequality of these values, message is displayed and the file reading process is interrupted. In the case of equality of those values, other configuration data read from the signature are passed to EnD, which resumes the process of decryption. During the process of decrypting the file, the EnD determines the value of a hash function for that file. After completion of the copying process, EnD transmit to DSu determined value of the hash function for verification. If the designated hash value is not equal to the value read from the signature, a message is displayed and the DSu deletes the file.

5. Conclusion

The SWSA uses proprietary solution for securing data on removable Flash RAM. There have not been applied widely available tools to secure the contents of this type of removable storage media due to the fact that these solutions typically uses only symmetric encryption when writing files. In these solutions, the key needed for encryption/decryption is specified by the user who creates a protected file and it is assumed that this key is known to the user when a protected file is read. The problems associated with the transmission of the key between the users are not taken into account. Such a solution in the SWSA was not useful.
The solution presented in this paper is an unique and more complicated one. The problem with the transmission of the key does not apply users of SWSA, because they are using the advantages of asymmetric encryption which gives assurance secured transfer of encryption key between parties, involved in the exchange of data. In addition, the SWSA uses Trusted Platform Module which supports the creation and management of cryptographic keys.
The developed system requires the user who creates a protected file, to specify only the recipient's file and the file encryption parameters. The recipient of the file can only

be the user authorized by the SWSA. The process of protecting file is closely linked with the mechanisms of systemic support for removable media Flash RAM, and is transparent to the user. When the protected file is read, user is not burdened with any additional activities. In addition, protections are constructed in such a way that the reading of a file is possible only by the authorized user by the SWSA, and only with the medium on which the file was originally saved. An attempt to copy the protected file to a different medium locks the ability to read the file.
The described method for protecting data on removable Flash RAM protects data against unauthorized access in systems processing the data, belonging to different security domains (with different classification levels) in which channel the flow of data must be strictly controlled. The described solution protects data stored on removable Flash RAM in case of loss or theft of the medium, but also makes it possible to secure transfer of that data through an unsecured transmission channel, for example using a courier.

Acknowledgements

This work was supported by The National Center for Research and Development, Project OR00014011.

References

[1] A. Kozakiewicz, A. Felkner, J. Furtak, Z. Zieliński, M. Brudka, and M. Małowidzki, “Secure workstation for special applications”, in *Secure and Trust Computing, Data Management, and Applications*, C. Lee, J.-M. Seigneur, J. J. Park, and R. R. Wagner, Eds., Communications in Computer and Information Science, vol. 187. Berlin: Springer, 2011, pp. 174–181.

[2] Z. Zieliński *et al.*, “Secured workstation to process the data of different classification levels”, *J. Telecom. Inform. Technol.*, no. 3, pp. 5–12, 2012.

[3] J. Chudzikiewicz, “Zabezpieczenie danych przechowywanych na dyskach zewnętrznych”, in *Metody wytwarzania i zastosowania systemów czasu rzeczywistego*, L. Trybus and S. Samolej, Eds. Warszawa: Wydawnictwo Komunikacji i Łączności, 2010, pp. 211–221 (in Polish).

[4] R. Nagar, “Filter Manager”, Microsoft Corporation, Redmond, 2003.

[5] R. Nagar, *OSR’s Classic Reprints: Windows NT File System Internals*. Redmond: OSR Press, 2006.

[6] Technical Documentation ”Microsoft Windows Driver Kit (WDK)”, Microsoft Corporation, Redmond, 2009.

[7] M. E. Russinovich and D. A. Solomon, *Microsoft® Windows® Internals, Fourth Edition: Microsoft Windows Server$^{\text{TM}}$ 2003, Windows XP, and Windows 2000*. Redmond: Microsoft Press, 2005.

[8] W. Oney, *Programming the Microsoft® Windows® Driver Model*. Redmond: Microsoft Press, 2003.

Jan Chudzikiewicz – for biography, see this issue, p. 11.

Janusz Furtak – for biography, see this issue, p. 12.