Attack on mount control code of commercial on-the-fly disk encryption software and efficient countermeasure

C. B. Roellgen

Global IP Telecommunications, Ltd. & PMC Ciphers, Inc.

August 21, 2008

Abstract

On-the-fly disk encryption software generally creates a virtual encrypted disk within a file and mounts it via a software driver as a real disk to the file system of a computer. In order to mount a virtual encrypted disk, a user interface application that is part of the software package, passes volume file path information, the selected encryption algorithm as well as the password to the software driver. By intercepting this so-called mount request, all required information to gain access to an encrypted volume is made available to an adversory in an extremely convenient way. In order to apply this attack successfully, it is assumed that the adversory is able to manipulate the kernel of the targetted operating system – an assumption that is increasingly realistic in the political context of the first decade of the 21st century.

Key words: Attack, on-the-fly, OTFE, on-the-fly, disk, encryption, IOCTL, DeviceIoControl, kernel, mount, device input and output control interface, interception, key exchange, Diffie-Hellman, asymmetric, block cipher, polymorphic cipher

1. Introduction

In the classical attack scenario, commercial on-the-fly (OTFE) disk encryption software makes highly secure virtual volumes available to users since almost two decades. The classical attack scenario is mainly limited to an adversory who gets hold of encrypted volume image files e.g. by stealing a notebook computer that contains such encrypted image files. Currently available OTFE software has consequently been programmed to make attacks on volume image files very time-consuming e.g. by storing little or no plaintext information in such files in order to give the adversory no hint about the selected encryption algorithm, keys, etc..

Hackers are considered as a possible threat in some attack scenarios. In order to make the task of a hacker difficult, passwords are cached by some encryption software and buffers are overwritten with a fixed bit pattern prior to deallocating memory. No malicious software can thus get hold of possibly secret information by reallocating memory locations that had previously been used by the encryption driver or the user interface of an OTFE software.

None of the existing scenarios take into account that the operating system itself could contain malicious code.

2. Preferred targets in an operating system to log password information

Intelligence agencies, regardless of their provenience and regardless of their mission, are extremely well funded. A certain hunger for civillian information is evident when looking back at the first few years of the 21st century.
It is logical and highly likely that hackers or assigned organisations take advantage of all available synergy effects to maximize the effectiveness of their work.
It would certainly be an advantage if the manufacturer of an operating system from a certain country would

cooperate with a certain intelligence agency. There exists no proof for this assumption, but the likelyhood for such two bodies to cooperate secretly is close to 100%.

Certain preconditions although need to be met:

- The attack needs to remain secret
- No or only little data transfer via wide area networks can take place as secrecy would otherwise be in danger
- High efficiency of the attack
- Only very valuable information is worth to be logged – above all other things: passwords as they are literally the key to valuable information
- Small size of the gathered information below any detection limit

These preconditions are very tough to meet.
As an example, popular keystroke loggers log a lot of useless information and attack efficiency is poor due to the fact that somebody who really wants to hide password information takes advantage of key files, virtual keyboards and USB tokens. Too much information needs to be buffered and data transfer of that much information via internet would be suspicious.

The logging of mouse movements, keystrokes, screen content, etc. proves to be rather inconvenient compared with logging MOUNT IO Control Codes:
Data encryption software is definitely a primary target for information loggers. Somebody who uses such software definitely wants to hide information from somebody else.
Vulnerability of such software is particularly high when passwords are being entered or when they are transmitted to a software driver. In order to mount a virtual encrypted disk, the user interface application of an OTFE (disk encryption) software passes volume file path information, the selected encryption algorithm as well as the password to a software driver.
The Windows operating system, as well as most or all other operating systems, provide only a single mechanism for doing this: the DeviceIoControl() function.

3. The DeviceIoControl() function is the ideal target

The DeviceIoControl() function, which is part of the Windows kernel, provides a device input and output control (IOCTL) interface through which an application can communicate directly with a device driver. The DeviceIoControl() function is a general-purpose interface that can send control codes to a variety of devices. Each control code represents an operation for the driver to perform. For example, a control code can ask a device driver to return information about the corresponding device, or direct the driver to carry out an action on the device, such as writing data to a disk.

A number of standard control codes are defined in the Windows SDK header files. In addition, device drivers can define their own device-specific control codes. The following example shows all control codes of a popular open source OTFE software as well as the data structure that is used for the MOUNT control code:

Apidrvr.h :

```
/* Public driver interface codes */

#define MOUNT                466944 /* Mount a volume or partition */
#define MOUNT_LIST           466948 /* Return list of mounted volumes */
#define OPEN_TEST            466952 /* Open a file at ring0 */
#define UNMOUNT              466956 /* Unmount a volume */
#define WIPE_CACHE           466960 /* Wipe the driver password cache */
#define HALT_SYSTEM          466964 /* Halt system; (only NT when compiled with debug) */
#define DRIVER_VERSION       466968 /* Current driver version */
#define CACHE_STATUS         466988 /* Get password cache status */
#define VOLUME_PROPERTIES    466992 /* Get mounted volume properties */
#define RESOLVE_SYMLINK      466996 /* Resolve symbolic link to target */
#define DEVICE_REFCOUNT      467000 /* Return reference count of root device object */
#define DISK_GET_PARTITION_INFO      467004
#define DISK_GET_GEOMETRY    467008
#define UNMOUNT_ALL          475112 /* Unmount all volumes */
```

```
#define TC_FIRST_PRIVATE      MOUNT   /* First private control code */
#define TC_LAST_PRIVATE       UNMOUNT_ALL /* Last private control code */

/* Start of driver interface structures, the size of these structures may
   change between versions; so make sure you first send DRIVER_VERSION to
   check that it's the correct device driver */

#pragma pack (push)
#pragma pack(1)

typedef struct
{
        int nReturnCode;                    /* Return code back from driver */
        short wszVolume[TC_MAX_PATH];       /* Volume to be mounted */
        Password VolumePassword;            /* User password */
        BOOL bCache;                        /* Cache passwords in driver */
        int nDosDriveNo;                    /* Drive number to mount */
        int BytesPerSector;
        BOOL bSystemVolume;                 /* Volume is used by system and hidden from user */
        BOOL bPersistentVolume;             /* Volume is hidden from user */
        BOOL bMountReadOnly;                /* Mount volume in read-only mode */
        BOOL bMountRemovable;               /* Mount volume as removable media */
        BOOL bExclusiveAccess;              /* Open host file/device in exclusive access mode */
        BOOL bMountManager;                 /* Announce volume to mount manager */
        BOOL bUserContext;                  /* Mount volume in user process context */
        BOOL bPreserveTimestamp;            /* Preserve file container timestamp */
        // Hidden volume protection
        BOOL bProtectHiddenVolume;          /* TRUE if the user wants the hidden volume within this
volume to be protected against being overwritten (damaged) */
        Password ProtectedHidVolPassword;   /* Password to the hidden volume to be protected
against overwriting */
} MOUNT_STRUCT;
```

Control codes consist of device type information, access mask, function code and method flag. As an example does the mount IOCTL 466944d = 00072000h define DeviceType 7 (=FILE_DEVICE_DISK), 'any access', function code 0c00h and that all data passed to the driver shall be buffered.
Any function code greater or equal 0800h = 2048d clearly identifies a user IOCTL. Function codes 0-2047 are reserved for Microsoft Corporation. A tamperared version of the DeviceIoControl() function could thus distinguish the MOUNT IOCTL from most other IOCTLs in a timely fashion.

The following code fragment is an excerpt of the source code of a disk encryption software. It shows how the data buffer for the MOUNT IOCTL is filled (green background) and how the DeviceIoControl() function is called (red color). Right after calling DeviceIoControl(), the programmer deletes the password section of the buffer he had passed to the DeviceIoControl() function (yellow color). He was obviously well aware that malicious code could later scan his buffer e.g. to gather password information.

Dlgcode.c :

```
int MountVolume (HWND hwndDlg,
                                int driveNo,
                                char *volumePath,
                                Password *password,
                                BOOL cachePassword,
                                BOOL sharedAccess,
                                MountOptions *mountOptions,
                                BOOL quiet,
                                BOOL bReportWrongPassword)
{
        MOUNT_STRUCT mount;
        DWORD dwResult;
        BOOL bResult, bDevice;
        char root[MAX_PATH];

        if (IsMountedVolume (volumePath))
        {
                if (!quiet)
                        Error ("VOL_ALREADY_MOUNTED");
                return -1;
        }

        if (!IsDriveAvailable (driveNo))
```

```
	{
		Error ("DRIVE_LETTER_UNAVAILABLE");
		return -1;
	}

	// If using cached passwords, check cache status first
	if (password == NULL && IsPasswordCacheEmpty ())
		return 0;

	ZeroMemory (&mount, sizeof (mount));
	mount.bExclusiveAccess = sharedAccess ? FALSE : TRUE;
retry:
	mount.nDosDriveNo = driveNo;
	mount.bCache = cachePassword;

	if (password != NULL)
		mount.VolumePassword = *password;
	else
		mount.VolumePassword.Length = 0;

	if (!mountOptions->ReadOnly && mountOptions->ProtectHiddenVolume)
	{
		mount.ProtectedHidVolPassword = mountOptions->ProtectedHidVolPassword;
		mount.bProtectHiddenVolume = TRUE;
	}
	else
		mount.bProtectHiddenVolume = FALSE;

	mount.bMountReadOnly = mountOptions->ReadOnly;
	mount.bMountRemovable = mountOptions->Removable;
	mount.bSystemVolume = mountOptions->SystemVolume;
	mount.bPersistentVolume = mountOptions->PersistentVolume;
	mount.bPreserveTimestamp = mountOptions->PreserveTimestamp;

	mount.bMountManager = TRUE;

	// Windows 2000 mount manager causes problems with remounted volumes
	if (CurrentOSMajor == 5 && CurrentOSMinor == 0)
		mount.bMountManager = FALSE;

	CreateFullVolumePath ((char *) mount.wszVolume, volumePath, &bDevice);

	if (!bDevice)
	{
		// UNC path
		if (volumePath[0] == '\\' && volumePath[1] == '\\')
		{
			_snprintf ((char *)mount.wszVolume, MAX_PATH, "UNC%s", volumePath + 1);
			mount.bUserContext = TRUE;
		}

		if (GetVolumePathName (volumePath, root, sizeof (root) - 1))
		{
			DWORD bps, flags, d;
			if (GetDiskFreeSpace (root, &d, &bps, &d, &d))
				mount.BytesPerSector = bps;

			// Read-only host filesystem
			if (!mount.bMountReadOnly && GetVolumeInformation (root, NULL, 0,  NULL, &d,
&flags, NULL, 0))
				mount.bMountReadOnly = (flags & FILE_READ_ONLY_VOLUME) != 0;

			// Network drive
			if (GetDriveType (root) == DRIVE_REMOTE)
				mount.bUserContext = TRUE;
		}
	}

	ToUNICODE ((char *) mount.wszVolume);

	bResult = DeviceIoControl (hDriver, MOUNT, &mount,
		sizeof (mount), &mount, sizeof (mount), &dwResult, NULL);

	burn (&mount.VolumePassword, sizeof (mount.VolumePassword));
	burn (&mount.ProtectedHidVolPassword, sizeof (mount.ProtectedHidVolPassword));

	if (bResult == FALSE)
	{
```

```
		// Volume already open by another process
		if (GetLastError () == ERROR_SHARING_VIOLATION)
		{
			if (mount.bExclusiveAccess == FALSE)
			{
				if (!quiet)
					MessageBoxW (hwndDlg, GetString ("FILE_IN_USE_FAILED"),
						lpszTitle, MB_ICONSTOP);

				return -1;
			}
			else
			{
				if (quiet)
				{
					mount.bExclusiveAccess = FALSE;
					goto retry;
				}

				// Ask user
				if (IDYES == MessageBoxW (hwndDlg, GetString ("FILE_IN_USE"),
					lpszTitle, MB_YESNO | MB_DEFBUTTON2 | MB_ICONEXCLAMATION))
				{
					mount.bExclusiveAccess = FALSE;
					goto retry;
				}
			}

			return -1;
		}

		// Mount failed in kernel space => retry in user process context
		if (!mount.bUserContext)
		{
			mount.bUserContext = TRUE;
			goto retry;
		}

		if (!quiet)
			handleWin32Error (hwndDlg);

		return -1;
	}

	if (mount.nReturnCode != 0)
	{
		if (mount.nReturnCode == ERR_PASSWORD_WRONG)
		{
			// Do not report wrong password, if not instructed to
			if (bReportWrongPassword)
				handleError (hwndDlg, mount.nReturnCode);

			return 0;
		}

		if (!quiet)
			handleError (hwndDlg, mount.nReturnCode);

		return 0;
	}

	BroadcastDeviceChange (DBT_DEVICEARRIVAL, driveNo, 0);

	if (mount.bExclusiveAccess == FALSE)
		return 2;

	return 1;
}
```

As the programmer immediately overwrites and deletes the password buffer, it is obvious that he has a certain level of mistrust in the environment in which his software will later run. He obviously knows that it would be too easy for malicious software to allocate memory and to scan this memory for password information. He although trusts or has to trust the operating system. He willingly passes extremely sensitive information in the clear to the only function in the whole operating system that dispatches commands to drivers. He further uses a function code that most probably nobody else in this world shares with him.

All in all this is an ideal target for a hacker or an assigned organisation to gather first class secret informations. No other real-life attack comes with all of the following advantages:

- The operating system is not slowed down because only a few private function codes need to be analyzed.
- Password and volume information is available in a highly condensed plaintext format.
- Only very few informations need to be logged over weeks and months.
- Solely ultimately interesting information will ever be logged.
- There exists no work-around.
- Single source of data.
- Very low cost to implement the attack.
- Programming requires no special skills.
- Attack is hidden perfectly in the operating system.
- Distribution to computer users via updates of the operating system.
- Encryption products can be as secure as they could ever be and it's still possible to break them at a touch of a button.
- Most professional file and disk encryption products are equally affected. There is even no need to know any source code of these products.
- Logged data won't be longer than a few kilobytes or one megabyte and can be read out quickly and conveniently e.g. when inspecting notebook computers at the airport customs by customs officials.

The only disadvantage is that help from the manufacturer of the operating system is almost a requirement.

4. Is the MOUNT Control Code Attack a hypothetical or a real-life attack

The MOUNT Control Code Attack, as presented in this paper, works – without any doubt – perfectly as it's easy to conceive and potentially easy to implement.

It is probably not used so far. Either it had already been invented by the Intelligence Community or this community might now read this paper attentively and perhaps like the idea.
People who's business it is to spy on other people only set a first class source of information aside if they have no access to it, if they are incredibly stupid or if it's against the law.

Would it be against the law?
Probably yes. It certainly isn't illegal to spy on foreigners. As almost everybody in this world is a foreigner to other countries, there are many "targets" legally available.

Is the attack accessible to a major Intelligence Agency?
It is highly uncertain that hackers and most contracting bodies are capable of taking advantage of this kind of attack as the necessary modifications of the operating system can hardly be realized without the help of the original manufacturer. It's not impossible, although.
No manufacturer in this world would be pleased when asked to manipulate his product(s). It is still pretty likely that this attack has already been realized. It could be a decisive source of information. If not yet realized, somebody might turn this hypothetical attack into a real life attack one fine day if there exist no countermeasures against it.

The availability of countermeasures might be an important reason to stop using it or for not turning this attack into reality. In order for this to happen it would be necessary to have a large user base migrate to truly secure data encryption software.

5. Countermeasure against the MOUNT Control Code Attack

Disk encryption and certain file encryption software packages cannot operate without software drivers. As the only way to communicate with the driver is the DeviceIoControl() function, there exists no efficient way to circumvent this kernel function. Using the Windows registry or a file to transfer the password would be a bad and highly unprofessional solution as password data would then be possibly permanently stored on a storage device.

The only truly ultimate solution is to take advantage of an extremely well-suited public key exchange protocol: the Diffie-Hellman key agreement.
Prior to the execution of a task that requires optimum protection, both kernel-mode encryption driver and user-mode application (which controls the encryption driver) negotiate a key that is private to both pieces of software. Password information, which is exchanged via the DeviceIoControl() kernel function, is thus totally inaccessible to the operating system.

Diffie-Hellman key exchange uses modular exponentiation to yield a unique key that is only known to the two parties which exchange the key. In the following explanation it is assumed that the key exchange is initiated by the user-mode control client and the client communicates with the driver. The client chooses a long integer number a at random and calculates α using the following formula:

$$\alpha = s^a \bmod p \quad ;$$

p is a fix and publically known long prime number
s is a fix and publically known primitive root mod p
a is freely chosen by the client. The client keeps a secret.
α is the public result of the computation performed by the driver.

α is sent to the driver. s and p are known to the driver. The driver chooses a long integer number b at

random and computes β using the following formula (same formula as above):

$\beta = s^b \bmod p$; p is a fix and publically known long prime number
s is a fix and publically known primitive root mod p
b is freely chosen by the client. The driver keeps b secret.
β is the public result of the computation performed by the client.

The driver performs another computation prior to completing the IRP:

$k = \alpha^b \bmod p$; k is the negotiated key. The driver keeps b and k secret.

The driver completes the IRP and sends β to the client.

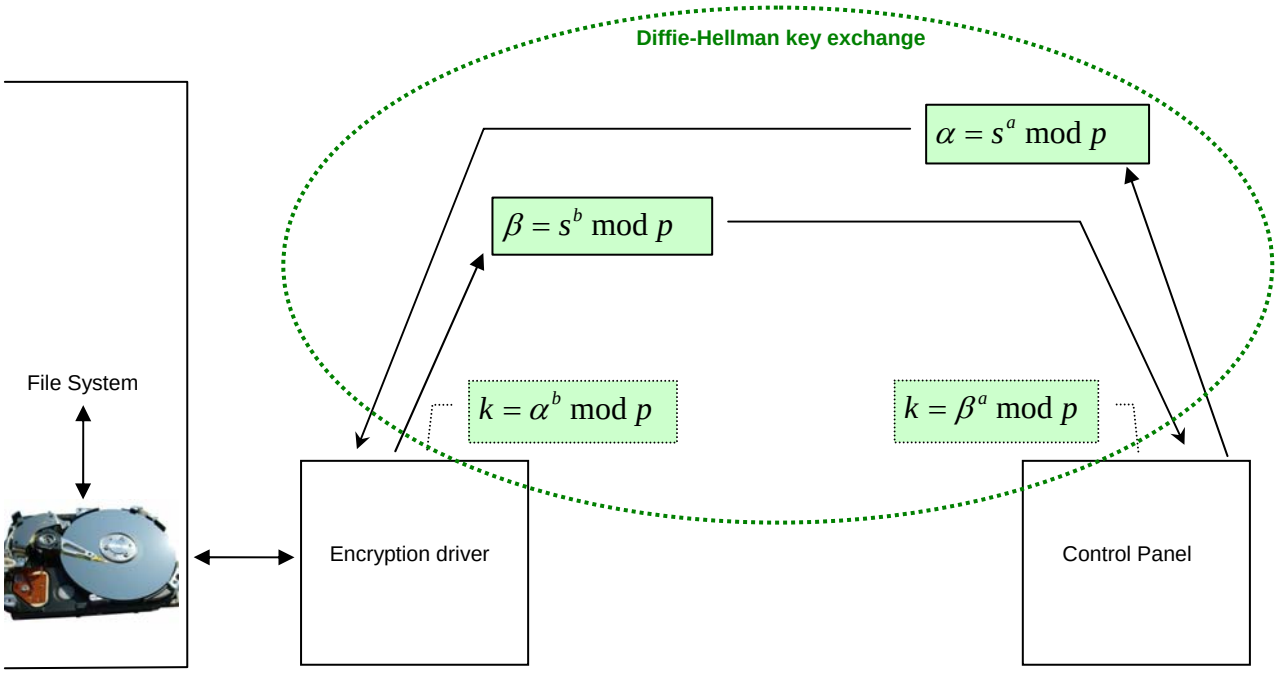
The client computes k as well through the following formula:

$k = \beta^a \bmod p$; k is the negotiated key. The client keeps a and k secret.

Both parties now share an information that is not accessible to malicious software sniffing driver communication.

For the sake of completeness here's the proof that both keys k are identical:

$$k = (s^b)^a \bmod p = (s^a)^b \bmod p$$



In contrast to RSA, it is impossible to willingly specify the message that is shared by both parties. Key k generated by Diffie-Hellman is a random number and it's identical for both parties.

Both Diffie-Hellman, as well as RSA require an extremely large amount of machine instructions on universal computers so that they are solely suitable to exchange key material.

Diffie-Hellman key exchange does not provide authentication of the communicating parties and is thus vulnerable to the so-called Man-in-the-Middle Attack. A person in the middle may establish two distinct Diffie-Hellman key exchanges, one with the client and the other with the driver, effectively masquerading as

client and driver, and vice versa. This allowing the attacker to decrypt (and read or store), then re-encrypt the messages passed between driver and client.

The Man-in-the-Middle Attack is generally a very big threat for Diffie-Hellman protocols like SSL as a remote server might be programmed to mount this attack. All software components of disk encryption software although are running on the very same machine. Assuming that the DeviceIoControl() function was tamperated in a way that it mounts a Man-in-the-Middle Attack on a specific disk encryption software taking advantage of the Diffie-Hellman key exchange, man-in-the-middle inevitably requires a substancial amount of CPU time to perform an additional key exchange. As it's the very same machine on which the attack is running, the attack would be extremely easy to detect by measuring the transport time span. As a matter of consequence, no Man-in-the-Middle Attack could ever be mounted successfully for a disk encryption software.

6. Conclusion

A new real-life attack against a broad range of encryption software products has been identified and described by us. Likelihood for many commercially available OTFE (on-the-fly encryption) softwares to be susceptible to this kind of attack is very high. The attack exploits the fact that highly secret information is exchanged in the form of plaintext by the components of disk encryption software through a single kernel function. This DeviceIoControl() function is potentially one of the best sources for first class secret data. Likelihood for this function to be manipulated is considerably high.

An efficient countermeasure based on the Diffie-Hellman key exchange against this potential weakness is proposed in this paper. As no workaround against this countermeasure exists, the MOUNT Control Code Attack cannot be applied to disk encryption software and file encryption software that is equipped with the countermeasure presented in this paper.

The OTFE (disk encryption) software TurboCrypt 2008 is the first product of its kind to feature this countermeasure.

Developers of other OTFE products will most probably follow and implement this countermeasure with time. To our knowledge is the described method free of patents and the author can confirm that he hasn't applied for protection of this intellectual property.

For more information: http://www.pmc-ciphers.com