got HW crypto?
On the (in)security of a Self-Encrypting Drive series

Gunnar Alendal
alendal@nym.hush.com

Christian Kison
Ruhr-Universität Bochum
QuaB.S0@gmail.com

modg
modgx@gmx.de

28th September, 2015

Abstract

Self encrypting devices (SEDs) doing full disk encryption are getting more and more widespread. Hardware implemented AES encryption provides fast and transparent encryption of all user data on the storage medium, at all times. In this paper we will look into some models in a self encryption external hard drive series; the Western Digital *My Passport* series. We will describe the security model of these devices and show several security weaknesses like RAM leakage, weak key attacks and even backdoors on some of these devices, resulting in decrypted user data, without the knowledge of any user credentials.

Keywords: Hardware cryptography, weak key attack, weak authentication attack, hardware RNG

1 Introduction

The Western Digital *My Passport* and *My Book* devices are external hard drive series connecting to host computers using USB 2.0, USB 3.0, Thunderbolt or Firewire, depending on model. These consumer off-the-shelf hard drives are available world wide. Many of the models advertise the benefit of hardware implemented encryption. These hard drives comes pre-formatted, pre-encrypted and are supported by various free software from Western Digital, both for Windows and Mac, to manage and secure the hard disks. Setting a password to protect user-data is one important security feature.

After researching the inner workings of some of the numerous models in the *My Passport* external hard drive series, several serious security vulnerabilities have been discovered, affecting both authentication and confidentiality of user data. We developed several different attacks to recover user data from these password protected and fully encrypted external hard disks. In addition to this, other security threats are discovered, such as easy modification of firmware and on-board software that is executed on the users PC, facilitating *evil maid* and *badUSB* attack scenarios, logging user credentials and spreading of malicious code ([1],[10]).

The rest of the paper is structured as follows: Section 2 introduces the desired security concept of the WD *My Passport* (MP) series. The section will give an overview of the different hardware models and introduce the host sided virtual CD (VCD) and SCSI commands to communicate with the devices. Furthermore the Key Encryption Key (KEK) and Data Encryption Key (DEK) generation is covered.

Section 3 shows security weaknesses and security threats that are identical for every WD *My Passport* that we encountered. In detail we show a way to dump the encrypted Data Encryption Key (eDEK) and VCD manipulation. The following sections deal with the different hardware chips we encountered, sorted by vendor.

Section 4 shows the JMicron chips.

Section 5 and 6 include the Symwave and resp. the PLX chip, which were found to have weaknesses that can be used as backdoors.

Section 7 explains the attacks against Initio chips.

Last but not least do we conclude our work in Section 8.

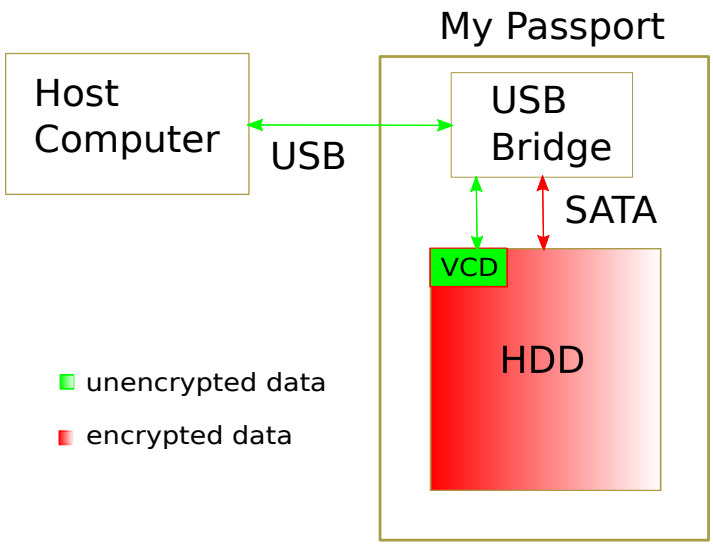
All results presented in this paper are from independent research.

2 Basic security design

My Passport is a family of portable external hard drives, all containing a 2,5" Hard Disk Drive (HDD) in an enclosure. A special feature is a single optimized PCB board with an interface-bridge and mounted headcontroller. This saves production costs and makes it less interchangeable with other external SATA HDDs. Although any user can, depending on the model, communicate over USB, Thunderbolt or Firewire with the device, we will concentrate on the USB models during this paper.

We encountered at least 6 different My Passport (MP) hardware versions, each presented in this paper. The USB bridge has an EEPROM for storing it's firmware and different configuration data. This EEPROM is in most cases marked "U14" on the PCB, making it easy to identify. On SYMW6316 this is marked "U8". The basic USB communication can be seen in Figure 1 and will be discussed in more detail in Section 2.2.

Figure 1: *My Passport* SED, with a HW encrypting USB Bridge



During normal operation the MP allows the user to set a password for data protection and integrity from a mounted VCD. The VCD is stored in a partition on the HDD and will run with administrator privileges on Windows, and user privileges on Mac hosts. The user authentication and encrypted data storage is identical on all devices over the whole series. To be more specific, we encountered differences in how the device stores and does user authentication, but the interface to the host machine is identical. Authentication and data confidentiality will be discussed in more detail in Section 2.3.

Analyzed hardware versions of the MP series are shown in Section 2.1

2.1 USB bridges supporting HW AES

The encryption and decryption in many of the *My Passport* models are done by the USB bridge that connects the host computer via USB to the SATA interface of the 2.5" HDD. In this bridge, the data is encrypted and decrypted after user authentication. It will block access to the SATA interface until user authentication is successful.

The *My Passport* series contains many different models, like the *My Passport Ultra* and *My Passport Slim*. The different models contain different USB bridge chips, but these chips can be different even for two disks of the same model name. On newer models it's not the USB-bridge that's performing the actual encryption and decryption, but the SATA controller itself. These specific models are discussed in 4.2 and 7.2. They are to some extent future work, as not every implementation detail is known by the time of writing.

Examples of some USB bridges used in the MP series are listed in Table 1.

Table 1: *My Passport* USB bridge examples

Manufacturer	Model	CPU architecture	HW AES
JMicron	JMS538S	Intel 8051 based	Yes
JMicron	JMS569	Intel 8051 based	No
Symwave	SW6316	Motorola M68k	Yes
Initio	INIC-1607E	Intel 8051 based	Yes
Initio	INIC-3608	ARC 600 based	No
PLX Technology	OXUF943SE	ARM7	Yes

This paper focuses on all the models listed in Table 1, simply because of availability to the researchers for testing. The MP drives are matched to chip model by it's VID:PID[1]. The VID and PID are reported from the MP disk to the host computer when plugged in.

A list of different VID:PIDs known to be using the JMS538S, SW6316, INIC-1607E and OXUF943SE chips are listed in Table 2. Note that the results in this paper has been verified on disks with VID:PID 1058:0730, 1058:0740, 1058:0748, 1058:071D and 1058:070a but the rest of the VID:PIDs in the table are all candidates to be vulnerable to the corresponding attacks described in this paper.

[1] VID: Vendor ID - PID: Product ID

Table 2: *My Passport*/*My Book* models utilizing JMS538S, SW6316, OXUF943SE or INIC-1607

Product Name	VID:PID	Chip model
My Passport Essential	1058:0740	JMS538S[2]
My Passport Ultra	1058:0741	JMS538S
My Passport Essential	1058:0742	JMS538S[3]
My Passport Ultra	1058:0743	JMS538S[3]
My Passport Enterprise	1058:0744	JMS538S[3]
My Passport for Mac	1058:0746	JMS538S
My Passport	1058:0748	JMS538S[2]
My Passport	1058:074a	JMS538S[3]
My Passport	1058:074c	JMS538S
My Passport Metal	1058:074d	JMS538S
My Passport	1058:074e	JMS538S
My Passport Slim	1058:0844	JMS538S
My Passport Slim	1058:0845	JMS538S[3]
My Passport Air	1058:0846	JMS538S
My Book AV-TV	1058:1026	JMS538S[3]
Elements 2.5 SE	1058:1042	JMS538S[3]
Elements 2.5	1058:1048	JMS538S[3]
My Book	1058:1140	JMS538S
My Book	1058:1142	JMS538S[3]
My Book Studio	1058:1144	JMS538S
My Book for Mac	1058:1148	JMS538S
My Book	1058:114a	JMS538S[3]
My Passport Essential	1058:0730	SW6316[2]
My Passport Essential	1058:0732	SW6316[3]
My Book Extreme	1058:1123	SW6316[3]
My Book	1058:1130	SW6316
My Book	1058:1132	SW6316[3]
My Passport Essential	1058:070a	INIC-1607E[2]
My Passport Elite	1058:070b	INIC-1607E
My Passport Studio	1058:070c	OXUF943SE
My Passport Studio	1058:071c	OXUF943SE
My Passport Studio	1058:071d	OXUF943SE[2]
My Book Studio	1058:1112	OXUF943SE
My Book Studio	1058:111c	OXUF943SE
My Book Studio	1058:111d	OXUF943SE

2.2 Communicating with *My Passport* devices

Any host computer communicates with the devices using standard SCSI commands over the USB, Thunderbolt or Firewire interface. The MP devices supports several Vendor Specific SCSI Commands (VSC), in addition to the common set of SCSI commands the devices must support [3]. Some VSC byte-ranges

[2]Tested

[3]Might not utilize encryption

allow any vendor specific opcode, to implement their own set of commands needed to fully operate their devices. These VSC commands are rarely documented by the different vendors.

Example VSCs for MP devices are $status$, $unlock$ (authentication), $erase$, $setpassword$, $changepassword$, $removepassword$.

To be able to send a raw VSC to any SCSI device, we need to know the Command Descriptor Block (CDB) [8] used to send the different SCSI commands. A list of some VSCs supported by MP and the corresponding CDBs are listed in Table 4. We can use the sg_raw util from the linux *sg3_utils* package [9] to send CDBs to a SCSI device.

We show an example using this utility by sending the $status$ command to a MP device attached to $/dev/sdb$ in Figure 2. In the returned data, $byte_0$ is a static tag byte $0x45$, $byte_3$ means locked state (00 == no password set, 01 == locked, 02 == unlocked, 06 == locked out), $byte_4$ is the current AES mode used, $byte_7$ is the KEK size, $bytes_{8-11}$ is four RNG bytes, $byte_{15}$ are number of AES modes supported and $bytes_{16-}$ are AES modes supported by this particular device (see Table 3).

Table 3: *My Passport* HW AES mode list

hex value	AES mode
10	AES-128-ECB
12	AES-128-CBC
18	AES-128-XTS
20	AES-256-ECB
22	AES-256-CBC
28	AES-256-XTS
30	FDE - Full Disk Encryption[4]

Figure 2: *My Passport* VSC $status$ example

```
root@linux:~/WD# sg_raw -r 1k /dev/sdb c0 45 00 00 00 00 00 30
SCSI Status: Good

Sense Information:
sense buffer empty

Received 18 bytes of data:
 00       45 00 00 00 20 00 00 20   00 97 eb 62 00 00 00 02
 10       10 20                                             .
root@linux:~/WD#
```

The interface, utilizing these standard SCSI commands and VSCs, is standardized for all MP devices. This simplifies the Western Digital software development and guarantees compatibility with all the different bridge models utilized. This means the basic communication and to some extent security model presented in this paper is identical for all chips. Depending on the chip model and manufacturer, the actual firmware implementation of the USB bridge might be different, but the set of supported VSCs remain constant.

[4]HDD itself handles authentication and encryption. The USB bridge is a pure USB to SATA bridge.

Table 4: *My Passport* VSC examples

Command alias	CDB	Associated data transferred
status status (OX)	c0 45 00 00 00 00 00 30 00 c0 45 00 00 00 00 00 00 20	
handystore	d8 00 00 00 00 01 00 00 01	
unlock	c1 e1 00 00 00 00 00 00 28	45 00 00 00 00 00 00 20 <32-byte KEK>
setpw	c1 e2 00 00 00 00 00 00 48	45 00 00 01 00 00 00 20 00 * 32 <32-byte new KEK>
removepw	c1 e2 00 00 00 00 00 00 48	45 00 00 10 00 00 00 20 <32-byte current KEK> 00 * 32
changepw	c1 e2 00 00 00 00 00 00 48	45 00 00 00 00 00 00 20 <32-byte current KEK> <32-byte new KEK>
erase	c1 e3 <4 * RNG> 00 00 28	45 00 00 01 20 00 01 00 <32-byte DEK material>

2.3 Authentication and data confidentiality

For those *My Passport* drives that support HW encryption by the USB bridge (see Table1), the chip uses an AES-128 or AES-256 bit key to encrypt user data before storing it on the HDD. This data-key will from now on be called the *DEK* . The models tested in this paper come with a default, factory set, AES-256 DEK. The factory default setting lacks a user provided password for obvious reasons and thus gives everybody access to decrypted data. Transparent to the user, user data is always encrypted using this key. Any user can now choose to set a password, which will produce a KEK, used to protect the DEK and some additional data. For this case the device requires a successful password authentication before giving access to the user data (see Figure 3).

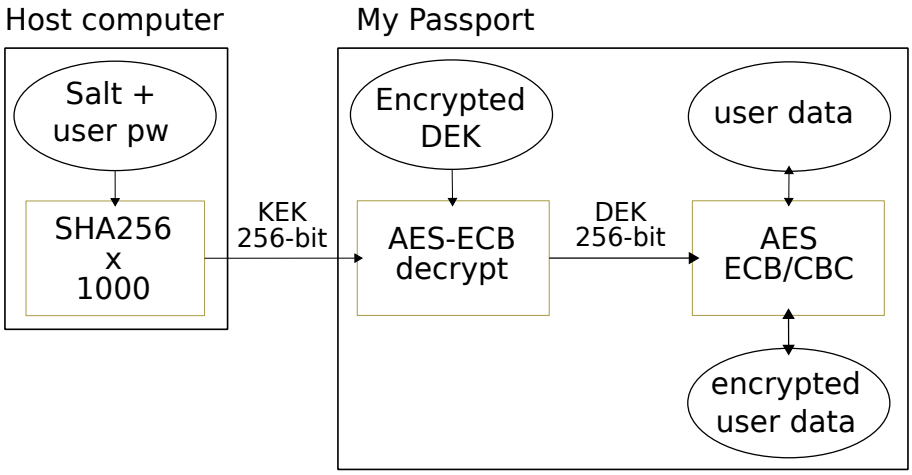
A factory set DEK is protected with a default, *static* "password" or hashvalue as we will see later in Section 2.4. This Key will be called KEK_s. Note that KEK_s is constant over all tested chips and can be found hardcoded in the bridge's firmware. This basically means no protection for the default case, as expected.

Once a MP disk is plugged in a host computer, the respective USB bridge will test if the disk is password protected. This is done by trying to unlock the encrypted DEK, using KEK_s from Table 5. If successful, the DEK is fed to the Hardware AES engine and the chip presents the host with a new logical disk unit (LUN[5]). The host only receives decrypted data and will mount the disk as a regular unencrypted HDD.

Failing to decrypt the DEK with KEK_s means the disk is protected by a user supplied password (usually). In this case the USB bridge presents a different LUN to the host computer containing a VCD to the host. This contains a GUI-based software to unlock the disk. This unlock software runs on the host computer (currently Windows and Mac supported) and asks the user for a password to unlock the disk. The user supplied password given to the software is used to generate a KEK that is used to decrypt the DEK , as described in Figure 3.

[5]Logical Unit Number

Figure 3: Standard authentication and encryption setup for *My Passport* devices



To summarize, given the correct password the correct KEK will be generated and the correct DEK will be decrypted and the LUN with user data is mounted with transparent on-the-fly decryption of data. Note that this might be different for chips which do not support HW AES encryption. Here the AES encryption is moved in the SATA chip. The USB-to-SATA bridge might perform user authentication / KEK verification (INIC-3608) or it will forward the KEK to the SATA chip for verification (JMS569) using standard ATA command *SECURITY UNLOCK - F2h* [13].

2.4 The Key Encryption Key (KEK) derivation

The KEK is a hash, derived from the user password. As discussed in Section 2.3, only the correct KEK unlocks the external MP.

This means that an attacker in possession of the KEK, can authenticate successfully to the disk and decrypt all data. He sends the VSC command $unlock$ from Table 4 and gains access to the mounted decrypted data. The KEK is generated by software on the host and sent via a VSC to the USB chip, an attacker does not need to know the actual user password which produces this KEK. The default KEK length is 256 bits (32 bytes).

Table 5: *My Passport* default KEKs (”PI” keys)

Algorithm protecting DEK	Default KEK (hex)
AES-256-ECB	03 14 15 92 65 35 89 79 32 38 46 26 43 38 32 79 FC EB EA 6D 9A CA 76 86 CD C7 B9 D9 BC C7 CD 86
AES-128-ECB	03 14 15 92 65 35 89 79 2B 99 2D DF A2 32 49 D6

If the user chooses to set a password, the KEK is generated from mixing a salt and a password to produce an AES-256 KEK:

- **The Key Derivation Function (KDF)**: Generation of the KEK is using SHA256 with the salted password as input, iterated x times. The USB bridge firmware supports configuring this iterator, like the salt. So each password set could define it's own iterator count x.

  The default iterator count of 1000 is hardcoded in the Western Digital software used to manage the HDDs. The function of the iterator x is to make password guessing infeasible, by being computationally expensive; running x times for one password guess.

- **The salt**: The USB bridge firmware supports setting a random salt for each password set. This salt can, and should, be retrieved from the disk via a VSC before doing authentication. The primary function of a salt is to defend against pre-computed dictionary attack hashes and pre-computed rainbow table attacks. The salt does not need to be kept secret.
  This salt is default set to "WDC." for any MP device. Further, this salt is hardcoded in the Western Digital software used to manage the disks. The salt never changes, even if the user changes his/her password.
  This means that the current use of the salt does not function as designed.

- **The password**: The password is user supplied and is appended to the salt.

The static iterator, together with the static salt allows an attacker to pre-generate large sets of possible KEKs from a password dictionary, and save valuable cracking time. Example of generating a KEK from user password "abc123" is shown in Algorithm 1.

Algorithm 1 KEK generation from password "abc123"

$Counter = 0$
$KEK =$ "$WDC.abc$123"
while $counter < 1000$ **do**
  $KEK = \text{SHA256}(KEK)$
  $counter+ = 1$
end while

The final KEK (hex) = `82 44 bc 08 9c 4a ab 5e 53 aa ec 57 ae 90 19 a7 3f 3c a0 6e de 80 7a 70 5b bb a7 10 cf 7c 3a c8`

In our opinion, the KEK should never be stored, regardless of the circumstances.

2.5 The DEK - Data Encryption Key

The DEK is the holy grail.

The DEK is the AES key used to encrypt all user data. Every user writeable sector on disk is encrypted using this key. This excludes the unencrypted VCD partition and the HDD Service Area (SA).

If an attacker can get hold of the DEK, he/she can decrypt all user data on the disk, regardless of the user password and KEK.

An attacker obtaining the correct DEK can bypass the USB bridge, read the raw disk sectors and decrypt user data manually. Reading the raw encrypted disk sectors can be done for example by connecting a PC-3000 [5] to the serial interface, disabling/removing the bridge firmware chip (EEPROM U14) to

enable USB-SATA direct bridge (see Section 3.1) or by soldering the SATA interface directly on the PCB [4].

The DEK is for some chips stored encrypted in the USB EEPROM, U14, in a hidden sector on the hard drive and in some cases in the SA of the hard drive. Within the SA the encrypted DEK, eDEK, is stored in module 0x25 (ROYL) for JMS538S and INIC-1607E and module 0x38 for SW6316. The encrypted DEK blob is easily identified, starting with the magic ASCII bytes `WDv1` for JMS538S, `SYMW` for the SW6316, `SInE` for OXUF943SE and `WDx01x14` for INIC-1607E models. For any other unknown chip it should be easily identifiable, searching for a small blob with high entropy within the last sectors of the HDD . Even when the location of the DEK is known, dumping it might be slightly different for various models and HDD sizes.

See Appendix B for examples of encrypted and decrypted DEK blobs.

3 Security weaknesses for every analyzed *My Passport*

This chapter will discuss security weaknesses discovered on every analyzed device, facilitating attacks to recover the eDEK and unauthorized tampering by an attacker with no knowledge of user credentials. The discussion of the weaknesses of the individual devices to recover the DEK is covered in Sections 4, 5, 6 and 7.

3.1 Dumping the eDEK

If we are able to get our hands on the eDEK, we can create a backup of our own eDEK, to be kept in a safe place. But this obviously also allows effective off-device password brute force attacks, like discussed in Section 2.4.

As already mentioned, the USB bridge has a supporting EEPROM, named U14 (U8 on SYMW6316) on the PCB. This memory contains crucial info for the USB bridge namely, the firmware, possibly a copy of the eDEK and various configuration blocks. One obvious method of dumping it is to de-solder it and to read out the eDEK directly, as the U14 is a very simple and widely used EEPROM.

On some models we were able to read out the eDEK through an undocumented VSC. However this only works if the device is unlocked. Nevertheless this did not work on all models so we looked for other ways. It turns out that reading out the hidden HDD sectors is easier than de-soldering the U14 chip.

3.1.1 JMS538S

Note that if we remove or disable the U14 on the PCB, the USB bridge will fall back to a ”blind” USB to SATA bridge mode, not being able to load the custom firmware. This is similar to soldering the SATA interface to the PCB [4]. A model with VID:PID 1058:0748 will show up as a device with VID:PID 152d:0539 instead, indicating vendor name *JMicron Technology Corp. / JMicron USA Technology Corp.* ($0x152d$) and product id 539. With the bridge in ”blind” USB to SATA mode, the hard drive reveals it's true sector count, which is a few thousand sectors more than is reachable through the Western Digital firmware. The reason is that the USB bridge, loaded with Western Digital firmware, keeps some sectors at the end of the disk hidden and not user reachable. The USB bridge stores another copy of the eDEK in this hidden area of the hard drive.

So we can access another copy of the eDEK, if we trigger the blind USB to SATA mode. We need to make the USB bridge think the U14 EEPROM is unreachable during power-up, when the firmware should be loaded. This can be accomplished by removing the chip, or by simply making the USB bridge fail to read from it during power-up. Failing to read from the U14 can be accomplished by shorting the *data IN (DI)* or *data OUT (DO)* pins of the EEPROM to ground, and with this enforce blind mode. Although this method is not recommended by responsible hardware technicians, none of the disks tested had their U14 EEPROM damaged during the numerous tests performed during our research.

Reading out the eDEK is now done by reading out the last few thousand hidden sectors from the hard drive, searching for the magic `WDv1` ASCII bytes.

One big advantage of this approach is that one of the *data IN (DI)* or *data OUT (DO)* is reachable on the back of the exposed PCB on the *My Passport* device, without detaching the PCB from the hard drive. So

simply removing the hard drive from the enclosure and knowledge of the location of the *data IN (DI)* or *data OUT (DO)* is all that is needed to reach the eDEK.

3.1.2 INIC-1607E

On the tested model, disabling the U14 EEPROM trick makes the bridge go into ”download mode”, accepting new, and freely patched firmware to the U14 EEPROM.

This is of course a weakness in itself, since now any attacker can update the firmware on any locked disk, executing an *evil maid* attack [1]. However, we used this trick to get hold of the eDEK without having to use expensive tools like PC-3000. We simply made a 3-byte patch of the firmware (including the 1 byte CRC update) which gives us back the eDEK when the $handystore$ VSC is executed. Normally this VSC reads a hidden sector on the HDD where the non-sensitive user parameters are stored (password hint, salt and KDF iterator). One can simply change the disk offset to the location of the eDEK instead.

3.1.3 OXUF943SE

On the tested OXUF943SE model (PID: 0x071d) no U14 trick is necessary, since the OXUF943SE is located on a separate PCB from the hard drive. Reading the hidden sectors can be done simply by connecting to the HDD SATA interface directly, reading out the last few thousand sectors. Additionally it is possible to calculate the eDEK from an intermediate leaked decrypted DEK (which is not the real DEK). This is can be done an undocumented SCSI VSC that probably was not meant to expose RAM, but can be abused for this purpose. The VSC command to read RAM is not included, to comply with the responsible disclosure model [11]. The RAM leakage is discussed in more detail in Section 6.1.

3.1.4 SYMW6316

Disabling the U8 EEPROM on this chip makes it go into bridging mode, too. However, we didn't fully investigate a way to expose the eDEK through this mode, but the code for this mode is contained in the U8 EEPROM. We couldn't read data directly from the HDD. For this chip we get the eDEK from a locked disk through the serial interface and the use of HDD recover tools like PC-3000.

3.2 Unauthorized modification of firmware and Virtual CD

In addition to the standardized Western Digital set of VSCs, all chip models have their own, non-standard, set of low level VSCs to support e.g. firmware and Virtual CD (VCD) upgrades.

Our research discovered that the firmware and VCD iso that is flashed to the *My Passport* devices are not digitally signed and revealed the existence of VSCs that can be used to update both the USB bridge firmware and VCD. The U14 EEPROM and VCD located on hard drive can also be modified directly by an unauthorized attacker. In the case of the U14 EEPROM, a 1- or 2-byte CRC-checksum is the only measure to ensure data integrity.

This facilitates an *evil maid* [10] type of attack for stealing user passwords, e.g. by tampering the firmware to log valid KEKs used on the device, for later collection by the attacker. Another arising

attack vector is a trojan based attack like in the *badUSB* [1] project or *Stuxnet* [12] to attack the whole host system and spreading of malicious code in the network. This does require modification of the U14 EEPROM chip directly, since the VSC to modify firmware will not work on a locked disk.

This was tested successfully on a JMS538S device, modifying the firmware to change how the *My Passport* identified itself to the host computer, and by modifying the VCD to execute *cmd.exe* instead of the Western Digital *unlock* executable. Note that the *unlock* executable by default requires admin privileges on Windows, which makes it a dangerous executable to modify. Modifying this has also been successfully tested and verified to work.

One big challenge with infecting the firmware, is that it is very hard to clean. It's also harder for any attacker to self-destruct any firmware changes, to cover tracks.

A malicious attacker can easily spread to new hosts by infecting the firmware of any connected and unlocked device.

The VSC commands to modify the firmware and VCD are not included, to comply with the responsible disclosure model [11].

4 JMicron bridges

WD uses two different JMicron bridges in their different hardware models. While the JMS538S, used in slightly older models, supports the chip hardware AES encryption, the newer JMS569 does not have any supported hardware co-processors or accelerators.

4.1 JMicron JMS538S - Cracking crypto the poor RNG way

The JMicron JMS538S USB to SATA bridge is used by many MP models. It is in fact one the oldest MP bridge known and has an AES-128 and AES-256 accelerated ASIC to support data en- and decryption. This allows on-the-fly data encryption for USB transfer speeds. The JMicron will be discussed in more detail in the next few sections, as this MP inherits a rather complex vulnerability compared to the other models.

4.1.1 Authentication phase

The authentication is done by trying to decrypt the encrypted DEK (`WDv1`) blob with the given KEK and check for a known ASCII string "DEK1" at the start of the decrypted DEK blob. If the correct string is found, authentication is considered successful and the DEK contained in the decrypted blob is fed to the HW AES crypto engine. The disk LUN is presented to the host computer, like described in Section 2.3. The authentication scheme can be found in Algorithm 2. An examples of an encrypted and decrypted `WDv1` blobs can be found in Appendix B.

As we did not find any further information regarding the DEK or KEK, we consider this authentication scheme safe. To further analyze the security of the JMS538S, we concentrate at the actual DEK generation. Key generation and RNG schemes are a crucial part of any secure cryptographic setup. If key generation fails, cryptography fails.

Algorithm 2 generic KEK validation for JMS538S, OXUF943SE and INIC-1607E

```
function ValidateKEK(KEK)
    DEK_E = GetEncryptedDEK()
    DEK_D = HW_AES_decrypt(KEK, DEK_E)          ▷ AES Key=KEK, Data=DEK
    if isValidDEK(DEK_D) then
        HW_AES_SetKey(DEK_D)
        MountUserDataDisk()
        WrongTryCounter = 0
        return True
    else
        WrongTryCounter++
        if WrongTryCounter > 5 then
            LockDevice()                        ▷ Device needs restart (Power cut)
        end if
        return False
    end if
end function
```

4.1.2 DEK generation and low entropy key material

A common question with all encryption schemes is how they generate the secret key material. Done properly, it should be as infeasible to attack as bruteforce of the actual key. The MP devices comes with pre-installed factory default keys, set in production. In early stages of the research we couldn't say anything about the generation of these pre-computed keys, leading to the results in the following subsections. However, it later turned out that even the factory set keys are vulnerable. We'll come to that in section 4.1.6.

In theory, nothing prevents the vendor from storing factory set keys, or creating them in a predictable way to be reproduced by an attacker. This might lead paranoid users to create new key material, a new DEK on their devices. Furthermore, in the case that a user forgets his/her password, there needs to be an option to reset and erase the device. This results in losing all user data stored, but making new use of the device.

The MP host and virtual CD software supports erasing the drive after typing the wrong password five times. If the user decides to erase, the $erase$ VSC must provide new key material to the device. The generation of key material follows the basic steps outlined in Algorithm 3. Basically the device is fed key material from two sources: the host computer and the embedded device itself.

The VSC used to erase the drive has a flag to signal the *My Passport* device to use or discard key material received from the *My Passport* on-device random number generator, RNG, using the host supplied key material as a raw DEK directly. The default setting, not configurable from software, is set to include both sources of key material.

Algorithm 3 DEK generation on JMicron 538S

function GENERATEDEK($HostSuppliedBLOB$)

// Key material bytes provided by host computer
$KeyBytesHost = HostSuppliedBLOB[8 : 8 + KeyLength]$

// Host computer decides if on-board RNG generated key material should be mixed in
$bMixKeyBytesHost = HostSuppliedBLOB[3]$

if $bMixKeyBytesHost == 0x01$ **then**
  for $i = 0; i < KeyLength; i++$ **do**
    // Mix key material from host computer with My Passport on-board RNG
    $DEK[i] = KeyBytesHost[i] \oplus$ HWRNGBYTE()
  end for
else
  for $i = 0; i < KeyLength; i++$ **do**
    // Use host supplied key material as raw DEK key
    $DEK[i] = KeyBytesHost[i]$
  end for
end if
return DEK
end function

Both of the key material sources will be analyzed in the following sections.

4.1.3 Key material source 1: Host computer

The MP devices support Windows and Mac as Host-OS. The $erase$ command, resetting the DEK , is initiated from software running on the host computer. It is a dangerous command, so as a form of protection, there is a 4 byte "SYN/ACK" as a challenge-response mechanism. These 4 bytes are handed to the host computer through every $status$ VSC sent to the device. So to be able to erase the disk: first the host computer has to send a $status$ VSC to retrieve a 4-byte "SYN" and then include this in the following $erase$ VSC . The MP device will check if the SYN given out in last $status$ VSC matches the received ACK value as part of the $erase$ VSC .

In addition, the $erase$ VSC sends $KeyLength$ key bytes to the MP. These $KeyLength$ key bytes will be used in the DEK generation scheme. To generate an AES-256 key, the $erase$ VSC provides 32 key bytes. It turns out that 32 bytes are always provided, regardless of the DEK length to be generated.

A key length of 32 bytes sounds a reasonable length of the host-provided key material, given (P)RNG generated data. However, it turns out that these 32 bytes are just *eight repetitions of a 4 byte value*, reducing the entropy source to a 32-bit value. Even for a truly random 32-bit value, a modern computer needs only a second to brute force. This 32-bit value comes from `GetTickCount()` on Windows host computers and `rand()` seeded with `time()` on Mac OS.

The conclusion is that the host computer only provides, at best, a 32-bit unknown value to be used as key material for the DEK generation scheme. This is easily bruteforce-able, but we remind the reader of the `xoring` with an on-board provided RNG explained in Section 4.1.2.

Please note, that this serious security weakness was fixed in software around May 2014 by Western Digital. The authors could not find the fix regarding this vulnerability commented in the release notes for the *WD Security* software version 1.1.0.51 [7]. Nevertheless, as the unlock software, located on the VCD for all MP has the ability to $erase$ the drive, this vulnerability is still present until all VCDs are patched. As the time of writing, no such VCD patch has been released.

4.1.4 Key material source 2: on-board RNG

The JMS538S on-board RNG does not seem to be implemented in the chip's firmware, so the physical implementation of this RNG is unknown and not studied. For evaluating the randomness we collected values generated by this RNG. One obvious and easy way is to collect a set of 4-byte SYN values returned from the $status$ VSC. Plotting the distribution of this set should give a random plot, given these 4-byte values comes from a cryptographically safe RNG.

A simple scatter plot of 10000 4-byte SYN values retrieved from $status$ VSCs sent to a device with VID:PID 1058:0748, is shown in Figure 4.

Comparing this plot to 10000 4-byte values retrieved from */dev/urandom* (Figure 6), gives us a strong hint that the RNG of the JMS583S is not cryptographically safe, showing clear patterns in Figure 4. As we did not expect to get "raw" RNG bytes from the SYN, we looked a little deeper.

A first way for retrieving raw RNG bytes from the on-board RNG is to abuse the $erase$ VSC. We can simply send an $erase$ command with static, $ZERO$, 32-byte host supplied key material. The resulting DEK generated will be from on-board RNG generated bytes only. There is an undocumented $read$ VSC we can use to read back the this newly created eDEK . Since we know the default KEK (see Table 5),

Figure 4: Scatter plot of 10000 4-byte values from $status$ VSC on JMS538S

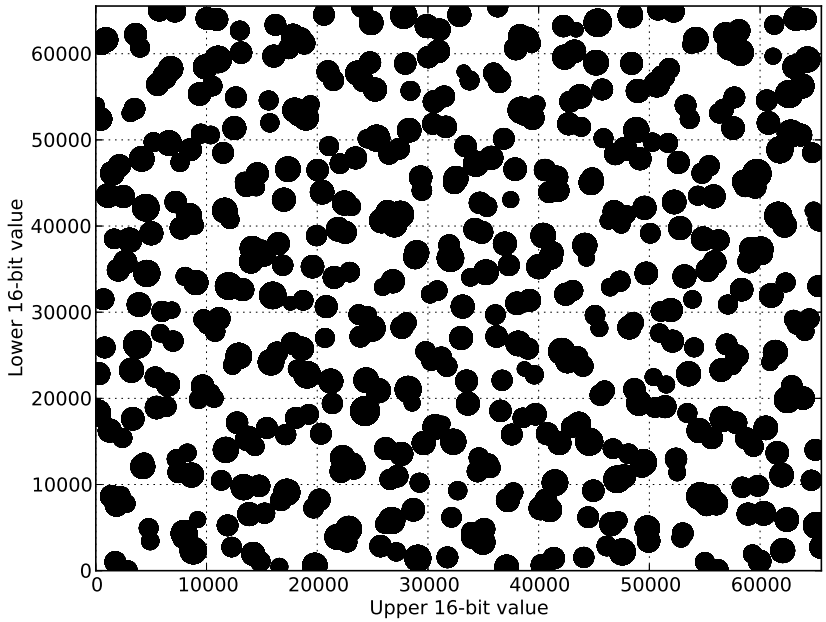


Figure 5: Scatter plot of 10000 **unmasked** 4-byte values from $status$ VSC on JMS538S

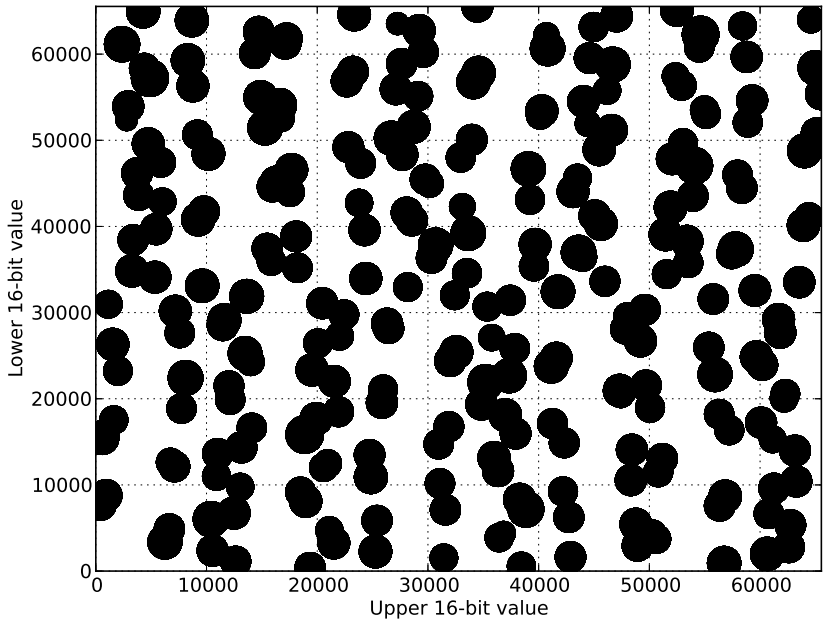
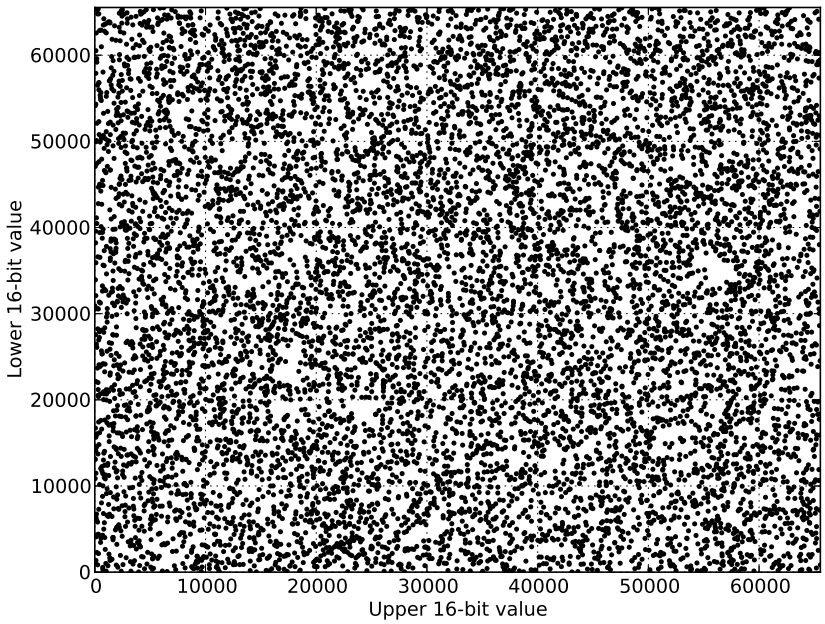


Figure 6: Scatter plot of 10000 4-byte values from /dev/urandom



we can easily decrypt the eDEK, now containing only raw RNG bytes. And in fact, the raw RNG bytes differ from the SYN bytes.

Another, even better, way that can be executed pre-authentication was found later. In fact, when looking deeper into the firmware code we noticed every time the $status$ VSC is called, the raw RNG bytes are masked with a static value. This static 4-byte value, $0x271828af$, is `xored` with the 4-byte RNG output. Further this value is `xored` with the last 4-byte SYN value before sending the value to the host computer.

Using the Algorithm 4, we can unmask the 4-byte values returned in a $status$ VSC response to reveal "raw" RNG bytes from the hardware side. Since this is done pre-authentication, any attacker can generate this from any locked drive.

A simple scatter plot of 10000 *unmasked* 4-byte SYN values retrieved from $status$ VSCs is shown in Figure 5. We can clearly see that many of the returned 4-byte values overlap, creating big circles in the plot. In fact, there are only 255 unique unmasked 4-byte values returned, in a set of 10000.

This leaves us with two different methods to retrieve values from the on-board RNG, one being pre-authentication. The second method abusing the $erase$ VSC can be repeated many times to generate a set of 32-byte values that is actual key material of generated DEKs.

It quickly turned out that the returned 32-byte values overlapped, indicating a very short period for the RNG. In fact, the period for the on-board RNG for JMS538S is only 255. That is, there is probably an on-board LFSR[6] with period of 255, giving a static 255 byte sequence of bytes, leaving us only with a small uncertainty of the state of the LFSR at any given time. Having only 255 possibilities, the complexity of this is close to 2^8. The LFSR can not return the value $0x00$.

[6]Linear Feedback Shift Register

Algorithm 4 4-byte RNG "SYN" generation on JMicron 538S

function GENERATE4BYTESYN($Last4ByteSYN$)

// Generate 1 byte at a time, calling on-board RNG 4 times
$4ByteSYN[0] = Last4ByteSYN[0] \oplus \text{HWRNGBYTE}() \oplus 0x27$
$4ByteSYN[1] = Last4ByteSYN[1] \oplus \text{HWRNGBYTE}() \oplus 0x18$
$4ByteSYN[2] = Last4ByteSYN[2] \oplus \text{HWRNGBYTE}() \oplus 0x28$
$4ByteSYN[3] = Last4ByteSYN[3] \oplus \text{HWRNGBYTE}() \oplus 0xaf$

// Store the generated 4-byte value in a global variable
$Last4ByteSYN = 4ByteSYN$

return $4ByteSYN$
end function

At any point in time an attacker can implement an attack with a complexity of 2^8 *to bruteforce the current state of the 255 byte RNG sequence generated from the on-board LFSR.*

Algorithm 5 on-board RNG sequence generated from several test devices from models 1058:0740 and 1058:0748, all with a JMicron 538S

function RNG()

$state = \Delta_{code} \bmod 255$

$RNGSequence = [0x8A, 0x5C, 0x6A, 0xDD, 0x1F, 0xEA, 0x6E, 0xE2, ...]$[7]

// returns byte at index $state$
return $RNGSequence[state]$
end function

Algorithm 5 shows pseudo-code which mimics the LFSR. The Δ_{code} is a delta that changes depending on the number of CPU instructions executed on the *8051* core executing the firmware code on a JMS538S. Consider this a CPU "tickcount" between each call to the RNG function. This changes the state (offset in RNG sequence) of the LFSR for each opcode executed. Although this sounds hard to predict, this is easily measured through testing or simply by brute force. The code loop that generates RNG bytes used in DEK generation is of course static and can be defined to have $\Delta = 1$, leaving us with the retrieved sequence listed in Appendix A. This delta will be static for all executions of this specific loop with these specific CPU opcodes.

Using this delta, we can compare RNG bytes generated from a different code block, the 4-byte SYN generator code. All unmasked 4-byte values extracted from this VSC (see Algorithm 4) will have a delta of 22 compared to the RNG sequence in Appendix A. Using this, we can test any locked disk for a vulnerable RNG just by testing the 4-byte values from a number of *status* VSC calls. Tests shows we need around 800 calls to the *status* VSC to regenerate the whole RNG sequence from Appendix A.

Since this Δ_{code} depends on the actual code executed in e.g. the DEK mixing loop, this delta will change if the loop code changes by only a single byte. On all the different firmware versions we encountered on

[7] More of this 255 byte LFSR sequence can be found in Appendix A.

our test devices, this code block was identical, making the Δ_{code} static for all tested devices.

In fact, the test of 4-byte "SYN" RNG values has also been used to verify that a *My Passport* model with VID:PID 1058:0820, using a different JMicron chip (JMS569), in fact uses the same RNG. However this does not seem to be a big weakness for this model, since the JMS569 does not support HW AES and this model only supports "AES mode" $0x30$ (FDE). This mode does not accept any host provided key material when erased. All authentication and encryption is done in the HDD itself.

Moving focus back to the vulnerable JMS538S, we can now reveal how to regenerate any DEK created using a mixture of weak key material from host computer and on-board RNG.

4.1.5 The attack on the $erase$ VSC generated DEK - recovering user data

Combining the host supplied key material with a worst-case complexity of 2^{32} (Windows' GetTick-Count()) with the on-board RNG with a complexity of 2^8, gives us a very practical brute force attack. This means that for every possible 32-bit value generated on the host as part of an $erase$ VSC, we only have to test 255 possible 32-byte sequences (offsets) from the on-board RNG sequence listed in Appendix A.

An attacker can regenerate any DEK generated from this vulnerable setup with a worst-case complexity of close to 2^{40}.

Of course this will be much less if the device was erased from a Mac host. Here eight repetitions of one 31-bit rand() value is used, seeded from a 32-bit time() seed. time() is a UNIX timestamp, which is seconds since 01.01.1970. We can expect to narrow this down to start at e.g. 2007 ($0x45900000$) and end at current date ($0x55000000$), resulting in a complexity around 2^{28}.

The "Windows-host" attack has been implemented as a PoC and since each generated DEK is tested with only one AES decryption, the attack is fast and can even draw benefit from most modern CPUs with AES-NI; hardware implemented AES. Running through the full range takes just hours on a normal high end computer using CPU cycles only. The only requirement is a known-plaintext encrypted 16-byte block from raw disk. E.g. sector 0 on an formatted drive will contain several known plaintext blocks with zeroes since mostly MBR[8] and GPT[9] headers are located here, encrypted with the vulnerable DEK. If ECB mode is used, these will show a 16-byte repetitive pattern. Refer to section 2.5 for an easy way to read raw disk sectors.

Once the DEK is recovered, an attacker can read and decrypt any raw disk sector, revealing decrypted user data. Note that this attack does not need, nor reveals, the user password or KEK.

4.1.6 The attack on the factory set DEKs - recovering user data

The attack on the $erase$ VSC generated DEKs described in the previous sections has one important requirement; the host provided key material needs to be predictable. As we could easily test and study $erase$ VSC commands, this was revealed rather quickly. Considering the fact that the MP devices are pre-encrypted with a factory set DEK_F, we analyze the DEK_F in this section.

[8] Master Boot Record
[9] GUID Partition Table

As a first approach we developed an algorithm to unmask the actual host provided key material from any given DEK. This can be done, as we know the RNG 255 byte sequence used in DEK generation. Furthermore we can actually recover the RNG state (index) at the time of any DEK generation, revealing the true 32 byte RNG byte sequence that was used in the actual factory DEK_F generation (See Algorithm 3). This is an immediate result from the observation in the `WDv1` blob that contains the DEK. At the time of any `WDv1` blob creation, the JMS538S firmware will fill in some RNG bytes in between the DEK bytes. These RNG bytes, which has nothing to do with the actual DEK, leaks the current state of the RNG, and subsequently the RNG state at the time of the factory DEK_f creation. Since the $erase$ VSC is used in setting the factory DEK, we know that the factory created `WDv1` blob was created and encrypted using the default "PI" KEK_s from table 5.

When we have decrypted the factory set `WDv1` blob to recover the factory set DEK_F and the RNG state at the creation time, we can recover the actual factory host provided key material from Section 4.1.3. We have no knowledge of how the secured factory setup looks like, but having a glance at one of the factory key host material gives us a strong hint.

The factory host provided 32 bytes of key material, consisting of eight 31-bit values, in little endian format. This observation gives a strong hint that $rand()$ is in use. Standardized $rand()$ returns 31-bit values. After some further testing we could regenerate these eight 31-bit values with the correct seed set. The seed for $srand()$ is only one 32-bit value, leaving us with the similar scenario as with the vulnerable $erase$ VSC scenario as in section 4.1.2.

From an attacker perspective this allows us to predict the actual seed used in the factory DEK generation setup, as it's just a function call to $time()$, which returns a 32-bit UNIX timestamp value. The implications of this is that we never need to brute force the full 32-bit complexity for the seed to $srand()$. We'll only brute force possible valid dates. So if we assume *My Passport* devices with JMS538S chips were starting to ship from factory at around year 2007, we could start at UNIX timestamp $0x45000000$ (07 September 2006 11:18:24 UTC) and end at around current date at UNIX timestamp $0x55000000$ (11 March 2015 08:42:40 UTC). This leaves us with only $0x01000000$ possible seeds to $srand()$. This has a complexity of 2^{28}.

Another fact that dramatically reduces the possible UNIX timestamp range is the fact that all HDDs are marked with a production date printed on the actual HDD. The factory DEK_F must have been generated close to this date. Our test devices show that the factory DEK_F set was generated within days after the HDD production date. We did not take advantage of this fact since the complexity of the attack was already easy to handle for all possible timestamps. However, this fact might apply to other chips, where the on-board RNG has a higher complexity compared to the JMS538S.

To attack and regenerate any factory set DEK for any JMS538S device using this vulnerable setup, we only need an attack with complexity around $2^{28} * 2^8 = 2^{36}$.

This attack has been implemented as a functional PoC and has been tested successfully for factory set DEK_F regeneration on 15 JMS538S devices with VID:PID $1058:0740$ and $1058:0748$. We did see DEK_F $time()$ timestamps ranging from 2007 to 2013 on our test devices. We came across one older JMS538S device with VID:PID $1058:0748$ that had what seems like a factory set DEK that didn't fit the above model. This HDD had a printed date 08 July 2012.

The complexity of the attack is so small that we in fact can pre-compute all possible factory generated DEKs, encrypt one 16-byte all-ZERO block with it and keep a sorted lookup-table with encrypted all-ZERO blocks and the corresponding seed and RNG index. This will make decryption of any JMS538S device with a vulnerable factory set DEK instant. An attacker only needs to get hold of a single 16

byte *encrypted* all-ZERO AES block from the device for instant DEK lookup and decryption of user data. Getting hold of such a block should be simple as most MBR and GPT partition headers contains multiple all-ZERO 16-byte blocks. Just read out the first sector of the encrypted disk with e.g. shorting the U14 DI or DO pin on power-up (see Section 3.1).

This lookup table with a total size of 1.2 TB has been generated and tested successfully for instant factory DEK_F lookup of 15 JMS538S devices with VID:PID $1058:0740$ and $1058:0748$.

To recover any factory set DEK_F for any JMS538S device, an attacker needs only one encrypted all-zero AES block for instant DEK lookup.

How many devices this affects is unknown to the authors, but we have recovered factory DEK_Fs from 2007 to 2013 from our 15 test devices. This implicates that many, or even most, factory generated DEK_Fs for JMS538S in this period are vulnerable for instant DEK lookup.

The source code for the JMS538S factory DEK_F attack and the code for creation of the JMS538S factory DEK lookup table will not be released to comply the responsible disclosure model [11].

4.2 JMS569

The JMS569 is another USB-to-SATA bridge from JMicron based on the 8051 architecture. This bridge is used in newer MP devices compared to the JMS538S and does not support HW accelerated AES encryption like most of the MP bridges. The only possible "AES Mode" supported is *0x30*, which refers to the FDE option. The host can not supply any host generated key material for the AES key if calling the $erase$ VSC. Everything is generated and set within the HDD. Analyzing this is future work. Nevertheless did we encounter encrypted data on the HDD when accessing the HDD directly with a user set password, as the AES encryption is now done by the HDD directly.

Facing a protected HDD is not new problem for HDD forensics. As there are already existing commercial solutions (e.g PC-3000), we analyzed the HDD directly with those tools. Their approach seems to follow a straight pattern, which allows SA access by overwriting the RAM/ROM and bypass security features like ATA passwords and optionally AES keys.

By forcing SA access and manipulating the SA area *0x124* and *0x127* we were able to unlock the HDD and disable the SATA AES encryption. Note that this works always, independent of the chosen user password and bridge status. We did not look further into the details of AES-key generation and validation as this solution is enough to get full user-data access. This has only been tested on a single available device with VID:PID, 1058:0820.

As our focus was mainly USB bridges supporting HW AES, this chip has not received much attention and should go into future work. Furthermore, most of this attack is based on commercial tools, so we don't provide a detailed attack to evade conflicts with the vendors and to comply the the responsible disclosure model [11].

A list of different VID:PIDs that might be using the JMS569 chip, are listed in Table 6.

5 Symwave 6316 - Revealing a backdoor

The protection of the DEK on the Symwave 6316 is basically security by obscurity.

The KEK is stored on the device. The KEK is protected using a hardcoded AES-256 key. This invalidates the protection of the DEK and is in effect ”security by obscurity”. The DEK is protected by the KEK, as usual. An example of an encrypted and decrypted `SYMW` blob can be found in Appendix B.

First of all, the KEK should never need to be stored on the *My Passport* device, since it is always derived from a user password entered during the authentication phase. This generated KEK should only be verified against the encrypted DEK stored on the device and destroyed after use, valid or not.

Secondly, the usage of a hardcoded key is only as good as how well hidden this hardcoded key is. In the case of Symwave 6316, the hardcoded key is located in the firmware for the SW6316 chip.

The hardcoded key is used in an AES key wrapping algorithm, described in full in RFC 3394 [6] to protect the KEK. Further, the key wrapping algorithm is applied to protect the DEK, using the KEK as key (See Algorithm 6).

In our opinion, the correct way would have been to only use the KEK directly in the AES key wrapping algorithm to protect the DEK, skipping the KEK key wrapping with a hardcoded key all together.

The effect of all of this is that an attacker needs only get hold of the SYMW blob to authenticate and decrypt any user data, regardless of any user password set. The attacker can even choose between authenticating normally to the device using the retrieved KEK, or simply by bypassing the USB bridge (e.g. if USB chip is broken), utilizing the unwrapped DEK on raw disk sectors to decrypt user data. This is described in Section 3.1 in more detail.

Algorithm 6 KEK/DEK decryption on Symwave 6316, utilizing key wrapping defined in RFC 3394

```
// initializations
HardcodedKey =
    29A2607A ........  ........  ........
    ........  ........  ........  ........[10]

// data from the SYMW blob found in U14 and HDD
KEK_wrapped      = SYMW_blob[0x60:0x60 + 0x28]
DEK_wrapped_part1 = SYMW_blob[0x10:0x10 + 0x28]
DEK_wrapped_part2 = SYMW_blob[0x38:0x38 + 0x28]

// unwrapping the KEK
KEK_unwrapped = AES_KEY_UNWRAP(KEK_wrapped, HardcodedKey)

// unwrapping the DEK
DEK_unwrapped_part1 = AES_KEY_UNWRAP(DEK_wrapped_part1, KEK_unwrapped)
DEK_unwrapped_part2 = AES_KEY_UNWRAP(DEK_wrapped_part2, KEK_unwrapped)
```

[10] The full 32 byte hardcoded key is not revealed to comply with the ”responsible disclosure” model.

6 PLX OXUF943SE - Revealing another backdoor

For this chip we are talking about a backdoor that can actually be closed by the user, if the user knows how.

The PLX OXU943SE stores the eDEK in a `SInE` blob on a hidden sector on disk, similar to most chips. The unusual design with this chip is that it stores the **previous** `SInE` blob when setting a (new) password which creates a new KEK and a new `SInE` blob. So for any user that sets the password only once on the OXUF943SE device, there exists a `SInE` blob protected with the known default "PI" KEK_s (see table 5). This leaves the device open for instant decryption if an attacker knows this fact.

On the tested device with VID:PID 1058:071d, reading out the `SInE` blobs is very easy as the OXUF943SE chip is located on a separate PCB than the actual HDD, with a normal SATA interface connecting them. Note that the authors had only this one OXUF943SE device available for testing.

The user can close this backdoor simply by changing the password twice or more times. This will overwrite the "PI" protected `SInE` blob. An example of an encrypted and decrypted `SInE` blob can be found in Appendix B.

6.1 RAM leakage of encrypted DEK

This specific chip is in fact also leaking the encrypted DEK directly from the on board RAM to the host computer, pre authentication.

Although this sounds easy enough, we need to look into how this is actually done, since it's not just sitting in clear RAM. What actually happens is that the OXUF943SE chip is leaking RAM belonging to the authentication code in the firmware, running on the device. If we recall the JMicron authentication scheme (Algorithm 2), the situation is the same here. The user enters a password, a KEK is derived, and this KEK is used as an AES key to decrypt the eDEK. If a valid DEK is decrypted, we have a valid KEK and thereby a valid password.

The chip in fact leaks the intermediate decryption of the DEK, DEK_D, given a password try/KEK. This means that if an attacker types in any password, creating a known KEK, the attacker can read back the intermediate decrypted DEK_D result from using this KEK. Assuming the password typed was wrong, which would be reasonable to assume for an attacker, we can re-encrypt the dumped DEK_D from RAM with our known *wrong* KEK to produce the original encrypted eDEK. This original eDEK is of course still protected with the correct user password, which we still don't know. Nevertheless this leaves the eDEK open for off-device attacks.

To read the device RAM an attacker can use an undocumented SCSI VSC. This command probably was not meant to expose RAM, but can be exploited for this purpose. The VSC command to read OXUF943SE RAM is not included, to comply with the responsible disclosure model [11].

7 INITIO Bridges

We found at least two models with an USB to SATA bridge from INITIO, model 1058:070a uses an INIC-1607E, while INIC-3608 is used in the model 1058:0810. The INIC-3608 does not have a HW accelerated AES engine.

7.1 INIC-1607E - Cracking crypto the poor RNG way, in theory

The INIC-1607E supports HW accelerated AES encryption. As usual the encryption is done using the DEK while the KEK encrypts the DEK. Everything is as described in Section 2.3.

7.1.1 Authentication phase

The authentication phase is very similar to that of JMS538S, as described in Algorithm 2. The KEK is derived from the user password and used to decrypt the eDEK . If there is a magic, 0x275dba35, found at offset $0x190$ in the decrypted eDEK (including header), the KEK is considered valid and the DEK is extracted and fed to the HW AES engine. See Appendix B for an example eDEK with decryption.

We have already discussed an easy trick to get hold of the eDEK in Section 3.1.

As for the JMS538S, there does not seem to be a direct way to bypass this authentication scheme, so again we target the DEK generation for further analysis.

7.1.2 DEK generation and low entropy key material

As we have the same scenario as for JMS538S, see Section 4.1.2 for a general discussion on low entropy key material for these devices.

The DEK creation algorithm for INIC-1607E is similar to JMS538S but is a bit more complex. See Algorithm 7 for details.

7.1.3 Key material source 1: Host computer

As the host computer is the same for all WD MP devices, the situation is the same as for JMS538S, when the $erase$ VSC has been used. Nevertheless we were not able to verify that the factory DEK_F for INIC-1607E might get similar factory key material as JMS538S. This remain future work. See Section 4.1.3 for details.

7.1.4 Key material source 2: on-board RNG

As we can see from Algorithm 7, the two sources of on-device key material are $HWRNG32()$ and $GetTickCnt()$.

Algorithm 7 DEK generation on INIC-1607E

function GENERATEDEK($HostSuppliedBLOB$)

// Key material bytes provided by host computer
$KeyBytesHost = HostSuppliedBLOB[8 : 8 + KeyLength]$
$nKL = KeyLength$

// Host computer decides if on-board RNG generated key material should be mixed in
$bMixKeyBytesHost = HostSuppliedBLOB[3]$

if $bMixKeyBytesHost == 0x01$ **then**
// Mix key material from host computer with My Passport on-board RNG
MEMCPY$(DEK, KeyBytesHost, KeyLength)$
$nRNG =$ HWRNG32$() >> 8) \ \& \ 0xfff$
$nTick =$ GETTICKCNT$() \ \& \ 0xfff$
$DEK[nKL-1] = DEK[nKL-1] \ \oplus \ nTick \ \& \ 0xfff \ \oplus \ nRNG$
for $i = 0; i < nKL - 1; i++$ **do**
// RNG values always 15 bit
$nRNG =$ HWRNG32$() \ \& \ 0x7fff$
// Tick values always 15 bit - bit 0 always set
$nTick =$ GETTICKCNT$()$
$nTickHIGH = (nTick >> 8) \ \& \ 0xfff$
$nTickLOW = nTick \ \& \ 0xfff$
$DEK[nKL-i] = DEK[nKL-i] \ \oplus \ nTickHIGH; \oplus nRNG \ \& \ 0xfff$
$DEK[nKL-i-1] = DEK[nKL-i-1] \oplus nTickLOW; \oplus (nRNG >> 8) \ \& \ 0xfff$
end for
else
for $i = 0; i < KeyLength; i++$ **do**
// Use host supplied key material as raw DEK key
$DEK[i] = KeyBytesHost[i]$
end for
end if
return DEK
end function

The RNG is seeded early in the firmware power-up sequence. The seed is two 1-byte CRC values xor'ed with a 15-bit $GetTickCnt()$ value to make a 16-bit seed.

The $GetTickCnt()$ function will return opcode ticks. It will for some reason always have bit 0 set. The increased tick count value will be predictable for a known code block or loop. We can simply measure a code block's tick delta or calculate it based on the opcodes executed in the block. We need this to implement an attack on this chip type.

Moving back to Algorithm 7, the inner for loop has a call to $GetTickCnt()$ for each iteration. So we need to measure how many opcodes are executed in each iteration of the loop. This turned out to vary within a narrow interval, and it turns out that different calls to $HWRNG32()$ adds different tick deltas, based on the state of the RNG. The reason is that the $HWRNG32()$ is bit shifting based on bits set in current state, which of course will vary depending on the actual state value. But we can overcome this very easy, since we need to know the state of the RNG when brute forcing we also know how big the tick delta will be. So an attacker can reimplement the $HWRNG32()$ function in software which will return both the next 32-bit RNG value together with the estimated tick delta based on current state.

In firmware power-on, the state of $HWRNG32()$ is seeded by a 32-bit value, with zeros in the lower 16-bits. So every seed is on the form $0xffff0000$. Further, it turns out, since Algorithm 7 only uses the lower 15-bits of each $HWRNG32()$ call, we can reduce complexity for the seed. In fact, of the 32-bit seed with only bits 16-31 used, bits 24-29 are insignificant to every generated sequence of values from $HWRNG32()$ & $0x7fff$. This means that the seed must be on the form $0xc0ff0000$. So the 16-bit seed alone can be brute forced with a complexity of 2^{10}.

Currently, at the time of writing, we have not found a way to reduce the complexity of the 15-bit value of $GetTickCnt()$. *This currently leaves us with a bruteforce attack to predict any on-device DEK key material from INIC-1607E with a complexity of* $2^{10} * 2^{15} = 2^{25}$.

This has been implemented as a PoC and multiple tests verify the complexity.

7.1.5 The attack on the $erase$ VSC generated DEK

The situation for INIC-1607E is similar as for JMS538S. See section 4.1.5 for details.

The complexity for attacking this chip is much higher because of the more complex on-device RNG generation. If we assume a Windows host, giving a host provided key material complexity of 2^{32}, we will get a total complexity of 2^{57}. This is still within reach given suitable hardware to run this type of complexity.

The full attack has not been verified on actual DEKs due to the high complexity.

7.1.6 The attack on the factory set DEKs

Again the situation is similar to JMS538S. Please refer to Section 4.1.6 for details. For the factory DEK, assuming the factory key material is similar to the verified JMS538S, we should take advantage of the $time()$ seed and the printed HDD date.

We expect the DEK creation $time()$ to be shortly after the printed HDD date. So an attack on the factory

DEK should start $time()$ brute force at printed HDD date and move forward. This should dramatically reduce the complexity. But still, a worst case scenario would be to try all possible valid $time()$ values from 2007 to today. This would leave an attacker with the complexity of $2^{28} * 2^{25} = 2^{53}$.

This full attack has been implemented, but has not been verified to work for a factory generated DEK, due to the high complexity.

7.2 INIC-3608 - My Passport Ultra and Slim - Revealing a backdoor

The INIC-3608 microprocessor is based on an ARC 600 CPU to bridge USB to SATA once again. This chip does not have a hardware accelerated AES engine. At first this seemed rather suspicious, as the package of the device advertises with hardware based encryption. If done in software there is no chance with an observed clock speed below 100 Mhz to encrypt at USB3.0 speed. Nevertheless as advertised, the MP-Slim and MP-Ultra do hardware accelerated AES encryption, but not on the bridge. It turns out, the HDD controller is doing the en- and decryption of user data. The USB to SATA is performing the user authentication and supports the standard VSCs from WD.

We tried to connect the SATA HDD directly over the PCB [4] for accessing a raw sector, but did not succeed. The data access is restricted by a set ATA-password in the SATA-HDD. The bridge is therefore doing user authentication by setting an ATA password, once the user generates his password. If the user does not supply a password, the user data is inaccessible when connecting directly to the SATA ports.

Finally, we were able to bypass the ATA password with commercial tools. Nevertheless, this is not an off-the-self solution offered, so we worked our way through to the AES protection. We located the location of the ATA password and some (unknown) connection to the AES password in different SAs from the internal 2.5" SATA HDD. After resetting the ATA password, we had complete access to the decrypted user data, as the SATA chip decrypted on-the-fly. Regardless of the user-password, KEK or DEK. By resetting the AES key and ATA password, we verified that the user data is indeed encrypted.

So far we have shown user data access for any case. Since this method is rather complex to perform with soldering SATA ports and getting SA access by commercial tools. We looked deeper into the authentication scheme for the INIC-3608. If we are able to let the bridge think we are the user, we can let it do the ATA password removal and just use the device like intended. As we already mentioned, the INIC is only responsible for user authentication. With no hardware encryption in place, a safe solution can be rather problematic to accomplish.

We moved on and dumped the U14 EEPROM by chip-off. We dumped the whole EEPROM for two different user-passwords. After diffing two sets of user passwords, we had only one location within the whole EEPROM that were different. As we set the user-password, we are able to calculate the KEK like described in Section 2.4 and simply search for it.

It turns out that the KEK is stored plain within the U14 EEPROM.

Simply by de-soldering the EEPROM, dumping it and re-soldering it back, we have the needed plain KEK for user authentication. From here on we get data access with a simple $unlock$ VSC.

The next natural step now would have been to try to create a patched firmware, upload it to the locked disk and read out the KEK from EEPROM, similar to the eDEK dumping hack for INIC-1607E (see Section 3.1). This would save an attacker the hassle of de-soldering and re-soldering.

However, there's a much simpler way of retrieving the KEK from a locked INIC-3608 disk.

It turns out that the KEK can be retrieved from a locked INIC-3608 device with a single VSC.

This means that an attacker only needs to send two VSCs to unlock any INIC-3608 disk; one VSC to retrieve the KEK and a second VSC, $unlock$, to authenticate to the disk.

Unlocking the disk using the backdoor is in theory a quicker way compared to unlocking with a know password. One simply doesn't need to iterate SHA256 1000 times.

The VSC command to read out the KEK directly from INIC-3608 EEPROM is not included, to comply with the responsible disclosure model [11].

A list of different VID:PIDs that might be using the INIC-3608 chip, are listed in Table 6.

Table 6: *My Passport*/*My Book* models utilizing INIC-3608 or JMS569

Product Name	VID:PID	Chip model
My Passport Essential	1058:07a8	INIC-3608[11]
My Passport Essential	1058:07aa	INIC-3608[12]
My Passport Essential	1058:07ac	INIC-3608
My Passport	1058:07ae	INIC-3608
My Passport Ultra	1058:0810	INIC-3608
My Passport Ultra	1058:0812	INIC-3608[12]
My Passport Slim	1058:0814	INIC-3608
My Passport Slim	1058:0815	INIC-3608[12]
My Passport Air	1058:0816	INIC-3608
Elements 2.5 SE	1058:10a2	INIC-3608[12]
Elements 2.5	1058:10a8	INIC-3608[12]
My Book	1058:11a0	INIC-3608
My Book	1058:11a2	INIC-3608[12]
My Passport Essential	1058:07b8	JMS569
My Passport Essential	1058:07ba	JMS569[12]
My Passport Ultra	1058:0820	JMS569[11]
My Passport Ultra	1058:0822	JMS569[12]
My Passport Slim	1058:0824	JMS569
My Passport Slim	1058:0825	JMS569[12]
My Passport Air	1058:0826	JMS569
My Passport Ultra	1058:0827	JMS569
My Passport Ultra	1058:0828	JMS569[12]
My Passport Ultra for MAC	1058:0829	JMS569
My Passport Ultra Metal	1058:082a	JMS569
My Passport Ultra Metal	1058:082b	JMS569[12]
Elements 2.5	1058:10b8	JMS569[12]
Essentials Desktop	1058:10bc	JMS569[12]
Elements 2.5	1058:10bd	JMS569[12]
Elements 2.5	1058:10be	JMS569[12]
My Book	1058:1220	JMS569
My Book	1058:1222	JMS569[12]
My Book for MAC	1058:1224	JMS569

[11] Tested

[12] Might not utilize encryption

8 Conclusion

In this paper we have analyzed different models from the external HDD *My Passport* series made by Western Digital. Overall we analyzed 6 different hardware models spread and well-distributed in the global market. We show the security concept intended from WD and present vulnerabilities on different hardware models. These findings range from easy eDEK leakages to perform off-line password brute-force to complete backdoors and plain KEK storage, resulting in complete security bypass. We were able to extract the bridge's firmware and eDEK in every hardware model with ease by software and/or chip-off. This leaked the firmware of the different chips for further security weaknesses and allows off-device password brute-force.

The KEK generation of the host software lacks the usage of a proper salt, even though the different hardware models support it. This allows precomputed hash-tables of dictionaries and brute-force attempts. This circumvents even the costly Key Derivation Function (KDF) of $1000\times$ SHA256 executions. The default static AES KEK, if the user does not set a password, is known and stored plain in the firmware EEPROM.

Another attack vector on every WD we analyzed, is the update mechanism of WD. The firmware update of the bridges and the emulated CD-ROM is done by undocumented vendor-specific SCSI commands, that is executable post-authentication. The firmware and virtual CD are not digital signed or cryptographically secured from tampering and modification. Both *evil maid* and *badUSB* attack scenarios are possible ([1],[10]). In addition, the VCD executable requires administrative privileges on Windows OS, so a modified VCD has full access to any host computer starting executables from it. The firmware can be modified to log the user-password or spread malware. The preinstalled VCD software that runs on hosts was found to generate weak key material with a worst case complexity of only 2^{32}. This can normally be compensated from the different hardware models with a sufficient RNG generator.

The RNG generator of the JMS538S has been shown to be predictable, as it returns 255 different bytes in a specific order, like a LFSR. The LFSR is constant over multiple JMS538S devices. Combined with the unpatched and vulnerable VCD software, a generated DEK after an $erase$ can be brute-forced with a complexity of 2^{40} . This is performed within hours on a modern computer and can even be precomputed to a simple look-up. This is completely independent from any user password set.

Further analysis showed that the factory set DEK on JMS538S devices are attacked with a simple table lookup attack. This can potentially affect many devices as the authors assume most users will never change this factory set DEK. The only way for users to fix this is to move all data from the device, erase the device with patched Western Digital software and move data back to the device.

Another hardware model uses a SYMWAVE, SW6316, USB to SATA bridge. This papers reveals a backdoor for instant KEK and DEK decryption. The KEK is encrypted with a static AES key, stored in the firmware. This obviously bypasses the whole security of an AES encryption, as an attacker can authenticate as the real user. The vulnerability is completely independent of the chosen user password. With the knowledge of the KEK an attacker could try to guess the user password off-device by running the KDF for every key guess, or look it up in a precomputed hash table since the salt is constant. With easy access to the `SYMW` encrypted DEK blob through HDD recovery tools, this opens the whole device with a simple HDD sector read and one AES decryption.

The paper also reveals a backdoor in another bridge, the PLX OXUF943SE. This bridge stores the previous encrypted DEK blob, `SInE`, when the user sets a (new) password. If this is the first and only time the user sets a password, there exists an encrypted DEK on disk that's protected only with the

default, and known, KEK. This ”PI” KEK is found in the firmware.

The weakest hardware model in terms of security is the INIC-3608 bridge. The chip does not support hardware accelerated AES encryption, and it's main purpose is the bridging and user authentication. The authentication is done by a simple memory compare matching with a stored *plain* KEK in the U14 EEPROM. One single command sent to the device will reveal the KEK, even if the disk is in a locked state. After retrieving the KEK, we can just authenticate as the user with the standardized $unlock$ VSC. Nevertheless this does not reveal the user-password. We were also able to bypass the ATA password set by the USB bridge with commercial tools, allowing data-access.

References

References

[1] Karsten Nohl, Sascha Krißler, Jakob Lell: BadUSB - On accessories that turn evil, https://srlabs.de/blog/wp-content/uploads/2014/07/SRLabs-BadUSB-BlackHat-v1.pdf, 2014

[2] SCSI command, http://en.wikipedia.org/wiki/SCSI_command

[3] SCSI Operation Codes, http://www.t10.org/lists/op-num.htm

[4] Tidosho, Recovering data from WD Elements drive when the USB connector is broken, http://tidelog.kitamuracomputers.net/2013/05/31/recovering-data-from-wd-elements-drive-when-the-usb-connector-is-broken/, 2013

[5] PC-3000 UDMA, http://www.acelaboratory.com/pc3000.udma.php

[6] J. Schaad, R. Housley: Advanced Encryption Standard (AES) Key Wrap Algorithm, https://www.ietf.org/rfc/rfc3394.txt, 2002

[7] WD Security for Windows Application Release Notes, http://support.wdc.com/download/notes/WD_Security_for_Windows_Release_Notes_1.1.1.3.pdf?r=833

[8] SCSI CDB, http://en.wikipedia.org/wiki/SCSI_CDB

[9] The Linux sg3_utils package, http://sg.danny.cz/sg/sg3_utils.html

[10] Evil Maid goes after TrueCrypt!, http://theinvisiblethings.blogspot.com/2009/10/evil-maid-goes-after-truecrypt.html

[11] Coders Rights Project Vulnerability Reporting FAQ, https://www.eff.org/issues/coders/vulnerability-reporting-faq

[12] Stuxnet, http://en.wikipedia.org/wiki/Stuxnet

[13] ATA Security feature Set Clarifications, http://www.t13.org/documents/UploadedDocuments/docs2006/e05179r4-ACS-SecurityClarifications.pdf

A JMS538S - DEK generation LFSR sequence

The 255 byte RNG sequence extracted from the on-board LFSR used on `VID:PID` models `1058:0740` and `1058:0748`, used in JMicron JM538S DEK generation function in firmware.

The full 255 sequence is not included to comply the the responsible disclosure model [11].

```
unsigned char aRandSeq[255] = {
    0x8A, 0x5C, 0x6A, 0xDD, 0x1F, 0xEA, 0x6E, 0xE2,
    0x10, 0xFC, 0x3C, 0x58, 0x55, 0xD2, 0x09, 0xB8,
    0xD4, 0xA7, 0x3E, 0xC9, 0xDC, 0xD9, 0x20, 0xE5,
    0x78, 0xB0, 0xAA, 0xB9, 0x12, 0x6D, 0xB5, 0x53,
    0x7C, 0x8F, 0xA5, 0xAF, 0x40, 0xD7, 0xF0, 0x7D,
    0x49, 0x6F, 0x24, 0xDA, 0x77, 0xA6, 0xF8, 0x03,
    0x57, 0x43, 0x80, 0xB3, 0xFD, 0xFA, 0x92, 0xDE,
    0x48, 0xA9, 0xEE, 0x51, 0xED, 0x06, 0xAE, 0x86,
    0x1D, 0x7B, 0xE7, 0xE9, 0x39, 0xA1, 0x90, 0x4F,
    0xC1, 0xA2, 0xC7, 0x0C, 0x41, 0x11, 0x3A, 0xF6,
    0xD3, 0xCF, 0x72, 0x5F, 0x3D, 0x9E, 0x9F, 0x59,
    0x93, 0x18, 0x82, 0x22, 0x74, 0xF1, 0xBB, 0x83,
    0xE4, 0xBE, 0x7A, 0x21, 0x23, 0xB2, 0x3B, 0x30,
    0x19, 0x44, 0xE8, 0xFF, 0x6B, 0x1B, 0xD5, 0x61,
    0xF4, 0x42, 0x46, 0x79, 0x76, 0x60, 0x32, 0x88,
    0xCD, 0xE3, 0xD6, 0x36, 0xB7, 0xC2, 0xF5, 0x84,
    .., ..,
};
```

B Encrypted and decrypted DEK blob examples

B.1 JMS538S

Protected with password ”abc123” (see Algorithm 1).

Encrypted DEK blob, `WDv1`, with header

```
00000000  57 44 76 31 69 98 00 00  00 e8 41 25 00 00 00 00  |WDv1i.....A%....|
00000010  00 00 00 00 00 00 f0 00  00 00 00 00 00 00 00 00  |................|
00000020  01 00 00 00 00 00 46 50  00 00 00 00 00 00 00 00  |......FP........|
00000030  00 02 ff 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  20 00 60 9c 00 00 01 63  00 00 00 00 57 44 76 31  | .`....c....WDv1|
00000060  ee ef 05 72 7c 06 6d 77  0c ad 3c f8 37 13 0a 7c  |...r|.mw..<.7..||
00000070  2a 9a 0c d8 0b c8 ef 80  a3 97 6b 32 b8 2d 68 0f  |*.........k2.-h.|
00000080  4b 24 fb d8 b6 fa a9 01  08 67 7c 4d a0 44 dd a2  |K$.......g|M.D..|
00000090  6d 3c 08 eb 62 67 0b 4a  fe ff 03 53 94 28 fa ac  |m<..bg.J...S.(..|
000000a0  5c fc bc 3c f3 1a c7 96  4b b1 87 8d 83 38 91 2f  |\..<....K....8./|
000000b0  13 60 26 07 62 a0 10 75  c6 94 0c 45 05 43 fe 83  |.`&.b..u...E.C..|
000000c0  9d 6c ed 94 f4 27 29 44  e8 a4 c0 34 49 00 04 9f  |.l...')D...4I...|
000000d0  9a 40 1d aa 2e 02 e7 62  09 35 02 ca 6e 80 e9 1f  |.@.....b.5..n...|
000000e0  87 7c 7a 2e 7a b4 99 5f  81 83 f7 41 f4 8e 51 d5  |.|z.z.._...A..Q.|
000000f0  6e 79 3d 49 f5 d0 2e b1  82 d0 6e 06 3e a2 fc 40  |ny=I......n.>..@|
```

Decrypted DEK blob, `WDv1`, with header

```
00000000  57 44 76 31 69 98 00 00  00 e8 41 25 00 00 00 00  |WDv1i.....A%....|
00000010  00 00 00 00 00 00 f0 00  00 00 00 00 00 00 00 00  |................|
00000020  01 00 00 00 00 00 46 50  00 00 00 00 00 00 00 00  |......FP........|
00000030  00 02 ff 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  20 00 60 9c 00 00 01 63  00 00 00 00 57 44 76 31  | .`....c....WDv1|
00000060  1f 00 00 00 f1 00 00 00  eb 00 00 00 d4 00 00 00  |................|
00000070  44 00 00 00 63 00 00 00  12 00 00 00 60 00 00 00  |D...c.......`...|
00000080  a3 00 00 00 49 00 00 00  f2 00 00 00 5a 00 00 00  |....I.......Z...|
00000090  44 45 4b 31 de 13 00 00  48 56 79 a7 74 f1 bb 83  |DEK1....HVy.t...|
000000a0  e4 be 7a 21 23 b2 3b 30  19 44 e8 ff 0c b4 87 40  |..z!#.;0.D.....@|
000000b0  6b 1b d5 61 f4 42 46 79  76 60 32 88 cd e3 d6 36  |k..a.BFyv`2....6|
000000c0  bb 1e 2f de 19 00 36 00  05 00 34 00 bf 00 2b 00  |../...6...4...+.|
000000d0  5a 00 0f 00 5c 00 dc 00  7d 00 ee 00 11 00 e4 00  |Z...\...}.......|
000000e0  79 00 87 00 42 b8 33 c7  20 00 00 00 00 00 00 00  |y...B.3. .......|
000000f0  13 00 00 00 0f 00 00 00  4f 00 00 00 8d 00 00 00  |........O.......|
```

Decoded `DEK1` blob

```
Magic                0x00: "DEK1"
CRC                  0x04: de13
Unknown              0x06: 0000
random1              0x08: 485679a7
key 0x3ee2 128 bit   0x0c: 74f1bb83e4be7a2123b23b301944e8ff
random2              0x1c: 0cb48740
key 0x3ef2 128 bit   0x20: 6b1bd561f442467976603288cde3d636
random3              0x30: bb1e2fde
key 0x3f02 256 bit   0x34: 1900360005003400bf002b005a000f005c00dc007d00ee001100e40079008700
random4              0x54: 42b833c7
key size (byte)      0x58: 20 => 256 bits
Unknown              0x59: 00000000000000
```

B.2 SYMW6316

Protected with password ”abc123” (see Algorithm 1).

Encrypted DEK blob, `SYMW`, with header

```
00000000  53 59 4d 57 f8 01 a6 0a  00 00 00 00 00 00 00 02  |SYMW............|
00000010  ed 16 1c d9 23 75 05 f9  68 d5 e4 0a 79 07 52 34  |....#u..h...y.R4|
00000020  95 8c 1a 21 14 26 77 72  d3 e1 a5 c5 61 42 b1 b7  |...!.&wr....aB..|
00000030  37 39 17 d9 96 34 e1 e2  c5 a5 35 52 56 2c 52 a2  |79...4....5RV,R.|
00000040  a8 bf e3 5e b9 fe 7a 9a  dd ec ac 4b 2b 89 8e 0e  |...^..z....K+...|
00000050  c2 7b 90 f0 86 fc d4 f1  fc bf a1 a2 5f 81 e1 be  |.{..........._...|
00000060  a4 1f 55 75 b1 c3 cc af  69 ab c5 54 61 e3 9a f8  |..Uu....i..Ta...|
00000070  0d 86 79 7e c6 e8 ca a1  3f 5a 50 75 0a e5 9b e5  |..y~....?ZPu....|
00000080  ab 45 b0 d7 7d c4 f6 ee  00 00 00 00 00 00 00 00  |.E..}...........|
```

Unwrapped (RFC 3394 w/hardcoded key) and decoded `SYMW` blob

```
[*] wrapped   PW HASH/KEK: a41f5575b1c3ccaf69abc55461e39af80d86797ec6e8caa13f5a50750ae59be5
                           ab45b0d77dc4f6ee
[*] unwrapped PW HASH/KEK: 8244bc089c4aab5e53aaec57ae9019a73f3ca06ede807a705bbba710cf7c3ac8

[*] wrapped   DEK part1  : ed161cd9237505f968d5e40a79075234958c1a2114267772d3e1a5c56142b1b7
                           373917d99634e1e2
[*] unwrapped DEK part1  : 9854236e0f4ea90950a831ed1c13643437311c0d316284f537310359306251ab

[*] wrapped   DEK part2  : c5a53552562c52a2a8bfe35eb9fe7a9addecac4b2b898e0ec27b90f086fcd4f1
                           fcbfa1a25f81e1be
[*] unwrapped DEK part2  : 0000000000000000000000000000000000000000000000000000000000000000
```

B.3 OXUF943SE

Protected with password ”abc123” (see Algorithm 1).

Encrypted DEK blob x 2, `SInE`, with header

```
00000000  53 49 6e 45 01 00 00 00  04 00 64 01 01 85 84 00  |SInE......d.....|
00000010  01 00 00 00 dc 22 c2 ed  f2 a5 7f 73 23 cf 58 28  |.....".....s#.X(|
00000020  4d 6f 4d 6f b5 fb 1a d1  9f f9 2a 72 70 51 93 b8  |MoMo......*rpQ..|
00000030  4b 74 2c 6a 67 19 3f 4c  c1 f9 57 6f ab e6 07 e5  |Kt,jg.?L..Wo....|
00000040  db e0 49 3c ad 00 89 b3  0d cb ef a1 e7 c5 75 9a  |..I<..........u.|
00000050  e2 db 1f 5f ff ff ff ff  ff ff ff ff ff ff ff ff  |..._............|
00000060  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
*
00000090  ff ff ff ff 01 00 00 25  88 6d 70 74 00 00 00 00  |.......%.mpt....|
000000a0  44 07 00 80 44 07 00 80  44 07 00 80 53 49 6e 45  |D...D...D...SInE|
000000b0  01 00 00 00 03 00 64 01  92 92 08 01 01 00 00 00  |......d.........|
000000c0  5a c1 e6 bc 38 ae 5d 07  c6 f6 43 5b ae 18 b9 12  |Z...8.]...C[....|
000000d0  94 a4 df 4a d5 3d 37 7d  a2 8d 26 06 83 02 3e 8d  |...J.=7}..&...>.|
000000e0  35 53 fa 24 52 5e e5 dc  a7 62 c5 d2 d5 c0 a9 8d  |5S.$R^...b......|
000000f0  1a d8 85 74 11 9e 57 52  b0 83 06 0d 8f cd 11 f3  |...t..WR........|
00000100  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
```

Decrypted and decoded `SInE` blobs

```
[*] Using abc123 as password for key blob
[*] SInE cipher mode (AES mode): 0x20
[*] Encrypted DEK blob: dc22c2edf2a57f7323cf58284d6f4d6fb5fb1ad19ff92a72705193b84b742c6a
5d 2e 4c 48 07 c1 10 8e ad 84 d2 bf 4b 5a 63 b9   | ].LH........KZc. |
be 24 2f 15 d0 06 f3 64 9c eb 35 bc 56 87 61 ca   | .$/....d..5.V.a. |
2d cf 9f b4 00 00 00 00 00 00 00 00 00 00 00 00   | -............... |
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   | ................ |
5a a6 46 69 c7 99 29 07 c5 85 11 fc 7f 88 b8 ef   | Z.Fi..)......... |
5a a6 46 69 c7 99 29 07 c5 85 11 fc 7f 88 b8 ef   | Z.Fi..)......... |
5a a6 46 69 c7 99 29 07 c5 85 11 fc 7f 88 b8 ef   | Z.Fi..)......... |
5a a6 46 69 c7 99 29 07 c5 85 11 fc 7f 88 b8 ef   | Z.Fi..)......... |
[*] This is the decrypted DEK of size 32 bytes (256 bit key):
[*] 5d2e4c4807c1108ead84d2bf4b5a63b9be242f15d006f3649ceb35bc568761ca

[*] No password given ==> using default passwork key from firmware
[*] SInE cipher mode (AES mode): 0x20
[*] Encrypted DEK blob: 5ac1e6bc38ae5d07c6f6435bae18b91294a4df4ad53d377da28d260683023e8d
5d 2e 4c 48 07 c1 10 8e ad 84 d2 bf 4b 5a 63 b9   | ].LH........KZc. |
be 24 2f 15 d0 06 f3 64 9c eb 35 bc 56 87 61 ca   | .$/....d..5.V.a. |
2d cf 9f b4 00 00 00 00 00 00 00 00 00 00 00 00   | -............... |
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   | ................ |
2b e4 8a 7f 6f 94 a7 9c f3 fa 25 51 c8 f9 a9 31   | +...o.....%Q...1 |
2b e4 8a 7f 6f 94 a7 9c f3 fa 25 51 c8 f9 a9 31   | +...o.....%Q...1 |
2b e4 8a 7f 6f 94 a7 9c f3 fa 25 51 c8 f9 a9 31   | +...o.....%Q...1 |
2b e4 8a 7f 6f 94 a7 9c f3 fa 25 51 c8 f9 a9 31   | +...o.....%Q...1 |
[*] Here U go: This is the decrypted DEK of size 32 bytes (256 bit key):
[*] 5d2e4c4807c1108ead84d2bf4b5a63b9be242f15d006f3649ceb35bc568761ca
```

B.4 INIC-1607E

Protected with password ”abc123” (see Algorithm 1).

Encrypted DEK blob, `WDx01x14`, with header

```
00000000  57 44 01 14 00 00 00 00  00 00 00 00 00 00 00 00  |WD..............|
00000010  00 00 00 00 1d 07 68 00  00 00 00 00 1d 07 68 00  |......h.......h.|
00000020  00 00 00 00 00 14 e0 00  20 00 00 00 00 00 00 00  |........ .......|
```

```
00000030  00 00 00 00 00 00 00 00  00 00 00 00 57 44 01 14  |............WD..|
00000040  32 92 ed 81 13 26 9e 98  df 1b a4 87 ef c6 37 3c  |2....&........7<|
00000050  18 e8 bb 69 98 0e 5b 22  1a 8d 1f 4f 82 8d 7b af  |...i..["...O..{.|
00000060  cf 39 c6 c5 d9 bd 9f c2  3e 6e 6e d5 d2 01 49 cd  |.9......>nn...I.|
00000070  f7 b2 33 c3 f1 fd 9e e7  e6 9b ba ad ae 1a ff 87  |..3.............|
00000080  c9 95 bd 5c 38 c0 21 22  e1 09 7e 9e d8 51 81 45  |...\8.!"..~..Q.E|
00000090  58 a4 dc ce 73 92 81 c5  67 06 f9 5b 25 30 dc 31  |X...s...g..[%0.1|
000000a0  82 62 b6 2c 86 27 a9 d6  14 a8 e6 41 b5 18 24 19  |.b.,.'.....A..$.|
000000b0  d2 9d 40 0a bd b2 b7 23  99 7d d7 d1 f3 dd a9 b0  |..@....#.}......|
000000c0  6d 33 67 b5 e0 ae 8e cb  dd 51 fb bf fc 27 74 12  |m3g......Q...'t.|
000000d0  e9 f5 b8 8d 94 c1 d9 be  ed 83 0a f5 aa d4 a3 0d  |................|
000000e0  72 36 4e ae 85 b4 b8 7a  23 7c c1 aa 20 c4 d7 3c  |r6N....z#|.. ..<|
000000f0  02 65 4a 45 b3 bd 27 e8  41 b9 a7 c1 f0 c2 2f 0b  |.eJE..'.A...../.|
00000100  54 6c 4a c5 d9 02 f3 25  bc b9 35 2c 32 5b 3b 18  |TlJ....%..5,2[;.|
00000110  b7 31 8c e4 e6 a9 88 b4  21 d9 98 06 3f 29 2f 69  |.1......!...?)/i|
00000120  8c e5 e1 ee c4 be f0 b3  8b fb 46 ee 16 1d e2 08  |..........F.....|
00000130  ef 18 a8 c1 1e a5 ec bc  96 bd a7 2d bf 39 2f 28  |...........-.9/(|
00000140  6b 69 be 08 ab 97 c8 97  27 2f 26 e8 65 f2 04 5a  |ki......'/&.e..Z|
00000150  24 f0 79 e0 62 1f be b4  3a f0 54 ab 8f 55 89 1b  |$.y.b...:.T..U..|
00000160  f1 6f b2 b0 ce 0e 3d ea  7b 74 19 e2 22 43 13 2c  |.o....=.{t.."C.,|
00000170  fc 14 54 79 f6 1d 3c 9d  b7 83 22 7e da 9c c5 aa  |..Ty..<..."~....|
00000180  94 91 cd 4a b8 4b 3b e7  81 39 13 21 ad e0 c8 64  |...J.K;..9.!...d|
00000190  b3 59 44 8c e7 dd 47 a3  5a 10 01 dc 45 8b 1e b1  |.YD...G.Z...E...|
000001a0  bd 38 e1 11 92 e3 58 9e  ac 4f d6 31 73 b6 fe 7d  |.8....X..O.1s..}|
000001b0  ef b2 a0 66 79 24 c9 4f  b0 b2 a9 73 2f 02 f7 ea  |...fy$.O...s/...|
000001c0  33 9e ba b5 fe 5e 9c 97  34 df c9 17 82 14 de 0f  |3....^..4.......|
000001d0  a0 c1 f0 fe d1 69 04 fd  d3 3e 06 90 84 a7 c9 d5  |.....i...>......|
000001e0  78 b7 2b 61 c2 df 71 77  63 7e a9 2f c5 5e 46 7e  |x.+a..qwc~./.^F~|
000001f0  38 dc 46 c6 06 6b 56 d9  e7 41 20 0b 16 44 cf c4  |8.F..kV..A ..D..|
```

Decrypted and decoded `WDx01x14` blob

```
[*] Using abc123 as password for key blob
00000000  57 44 01 14 00 00 00 00 00 00 00 00 00 00 00 00   | WD.............. |
00000010  00 00 00 00 1d 07 68 00 00 00 00 00 1d 07 68 00   | ......h.......h. |
00000020  00 00 00 00 14 e0 00 20 00 00 00 00 00 00 00 00   | ........ ........ |
00000030  00 00 00 00 00 00 00 00 00 00 00 00 57 44 01 14   | ............WD.. |
00000040  00 00 6c 2d 00 00 00 00 00 00 00 00 00 00 00 00   | ..l-............ |
00000050  00 00 f6 f5 00 00 00 00 00 00 00 00 00 00 00 00   | ................ |
00000060  00 00 95 d9 00 00 00 00 00 00 00 00 00 00 00 00   | ................ |
00000070  00 00 aa ce 00 00 00 00 00 00 00 00 00 00 00 00   | ................ |
00000080  00 00 1e 8c 00 00 00 00 00 00 00 00 00 00 00 00   | ................ |
00000090  00 00 10 ad 00 00 00 00 00 00 00 00 00 00 00 00   | ................ |
000000a0  00 00 67 29 00 00 00 00 00 00 00 00 00 00 00 00   | ..g)............ |
000000b0  00 00 55 5d 00 00 00 00 00 00 00 00 00 00 00 00   | ..U]............ |
000000c0  00 00 4f 04 00 00 00 00 00 00 00 00 00 00 00 00   | ..O............. |
000000d0  00 00 82 01 00 00 00 00 00 00 00 00 00 00 00 00   | ................ |
000000e0  00 00 57 45 00 00 00 00 00 00 00 00 00 00 00 00   | ..WE............ |
000000f0  00 00 57 66 00 00 00 00 00 00 00 00 00 00 00 00   | ..Wf............ |
00000100  00 00 e8 93 00 00 00 00 00 00 00 00 00 00 00 00   | ................ |
00000110  00 00 4e ec 00 00 00 00 00 00 00 00 00 00 00 00   | ..N............. |
00000120  00 00 8c b4 00 00 00 00 00 00 00 00 00 00 00 00   | ................ |
00000130  00 00 c1 9e 00 00 00 00 00 00 00 00 00 00 00 00   | ................ |
00000140  00 00 cf f9 00 00 00 00 00 00 00 00 00 00 00 00   | ................ |
00000150  00 00 0f e9 00 00 00 00 00 00 00 00 00 00 00 00   | ................ |
00000160  00 00 72 50 00 00 00 00 00 00 00 00 00 00 00 00   | ..rP............ |
00000170  00 00 f4 37 00 00 00 00 00 00 00 00 00 00 00 00   | ...7............ |
00000180  00 00 05 54 00 00 00 00 00 00 00 00 00 00 00 00   | ...T............ |
00000190  27 5d ba 35 bf e0 2f f9 00 00 00 20 4b d0 0e 82   | ']..5../.... K... |
000001a0  8b 61 60 ef 24 ec 55 b7 38 99 8c 68 93 a5 b5 61   | .a`.$.U.8..h...a |
000001b0  a4 94 bf 92 03 3b 80 bf eb 2e 5d 48 00 00 27 5a   | .....;....]H..'Z |
000001c0  00 00 b9 bf 00 00 00 00 00 00 00 00 00 00 00 00   | ................ |
000001d0  00 00 14 d4 00 00 00 00 00 00 00 00 00 00 00 00   | ................ |
000001e0  00 00 f1 9b 00 00 00 00 00 00 00 00 00 00 00 00   | ................ |
000001f0  00 00 22 de 00 00 00 00 00 00 00 00 00 00 00 00   | ..".............. |
[*] Raw DEK (32 bytes): 820ed04bef60618bb755ec24688c993861b5a59392bf94a4bf803b03485d2eeb
```