Lost your “secure” HDD PIN? We can help!

Julien Lenoir and Raphaël Rigo
2016-10-21

Abstract
USB HDD enclosures with encryption and pin pads are convenient and (supposedly) secure. In this paper we present our analysis of several models, both at the design level and the implementation level. We show that most of them have serious design flaws, some are totally broken and one even has a backdoor. After taking a step back and reflecting on the ecosystem, we propose a better design.

Contents

1 Introduction ............................................ 2
   1.1 Context ........................................ 2
      1.1.1 Drive types ................................ 2
      1.1.2 Attacker model ................................ 2
   1.2 Architectures .................................... 3
      1.2.1 Usability .................................. 3
      1.2.2 Hardware design .............................. 3
      1.2.3 Auditors’ goal ............................... 4

2 Practical examples .................................... 4
   2.1 Zalman ZM-SHE500 ................................. 4
      2.1.1 Hardware .................................. 4
      2.1.2 First steps: basic crypto checks .......... 5
      2.1.3 The update, the updater and the backdoor ....... 6
      2.1.4 Secrets: PIN ................................ 7
      2.1.5 Secrets: master key ......................... 10
      2.1.6 Secrets: AES-XTS keys ..................... 11
      2.1.7 Attack 1: Instantaneous PIN recovery ........ 12
      2.1.8 Attack 2: Offline hard drive decryption ...... 12
      2.2 Zalman ZM-VE500 ................................. 13
         2.2.1 Hardware .................................. 13
         2.2.2 First steps: basic crypto checks .......... 14
         2.2.3 Firmware analysis .......................... 15
         2.2.4 “Security” scheme .......................... 16
         2.2.5 LCD debugging ............................. 17
         2.2.6 AES patching ................................ 18
         2.2.7 Bruteforcing ................................ 18
         2.2.8 Conclusion .................................. 19
      2.3 Zalman ZM-VE400 ................................. 19
1 Introduction

The goal of this paper is to give an overview of how “secure” external hard drive enclosures are designed today and how their security can be assessed, mostly with a software approach, as a complement to [CR15] which covered hardware aspects.

1.1 Context

1.1.1 Drive types

External hard drives are now all connected through USB. Products are sold both with and without included hard drives and most of them offer some kind of security features to optionally protect data stored on the drive: password protection, encryption, etc.

Usually, those protections rely on software components running on the computer to “unlock” the drive. This paper will not consider such drives, which have been studied extensively by [Dom16] or [AKm15] but will focus on self-contained products, unlocked using a physical keyboard.

Models are basically split in three categories:

- cheap (< 100 USD): Zalman products, LOCKDOWN, Corsair Padlock, Netac, etc.
- mid-range (> 100 USD): Apricorn, Lenovo, DataLocker/IronKey.
- certified for classified material storage: Globull, LOK-IT.

This study will deal mostly with cheap drives but the discussion will also cover some mid-range designs.

1.1.2 Attacker model

Designs using a physical keyboard face interesting challenges to provide good security:

- their computing power is limited,
- the keyboard interface does not (practically) allow for long and complex passwords.

Regarding the threats faced by the hard drives, the obvious goal for the attacker is to access the user data in the clear, without knowledge of the PIN.

In practice, we will consider the following attacker models, of increasing difficulty:

- access to disconnected drive with enclosure, destructive actions allowed.
- access to disconnected drive with enclosure, case opening.
- access to disconnected drive with enclosure, no case opening.
• access to drive without enclosure.

Of course, in this study, we consider all the attack models at once but this classification is useful to rate the impact of the final attack.

"Evil maid" or "BadUSB" attacks are also possible, but will not be considered.

1.2 Architectures

1.2.1 Usability

All drives are unlocked using a PIN, some of them support several PINs. Of course, the user must be able to change his PIN without reencrypting the entire drive. The drive should also tell the user when a wrong PIN is entered. This implies that:

• the data encryption key is set once and for all (except reset)
• some means of verifying the PIN is present

i.e. two different types of secrets.

1.2.2 Hardware design

Most drives use the same architecture, based on the following components:

• a commercially available USB to SATA bridge, with integrated AES encryption
• a microcontroller to handle PIN input and (optionally) LCD
• a SPI flash memory to store the controller’s firmware

Two main variants exist:

• with the microcontroller integrated in the USB-SATA bridge (figure 1)
• with an external microcontroller, with internal flash memory too and optional external SPI flash (figure 2)

"Secure" variants for mid-range models may use secure external microcontrollers, for example:

• IronKey, even though they do not have keyboard-based models, uses a custom ASIC, plus physical security measures such as epoxy.
• Apricorn, in their Fortress model, uses epoxy and a FIPS validated microcontroller.
1.2.3 Auditors’ goal

Audited drives use a USB-SATA bridge chip implementing standard encryption algorithms, namely AES. Moreover those chips are often FIPS validated, which means the crypto implementation should be correct\(^1\) and thus that flaws probably reside in invalid key management or insecure storage.

Auditors have to answer the following questions regarding crypto keys:

- How are they generated?
- Is their entropy high enough?
- Where are they stored?
- Do their storage depend on user’s secret (PIN)?

This means some basic testing should be done before any advanced analysis: testing different PINs, checking for actual encryption, testing an encrypted drive in several enclosures, etc. This has been covered in [CR15].

2 Practical examples

2.1 Zalman ZM-SHE500

2.1.1 Hardware

First, we open the enclosure to look at the PCB to identify the various components, as shown on figure 3.

We find that the drive uses the simple architecture with essentially 2 chips:
- a MediaLogic MLDU03 USB-SATA bridge (product brief)
- a Winbond W25Q16CL SPI flash (datasheet)

Unfortunately, the data sheet for the MediaLogic chip is not available. And no information is given about the CPU core it uses. However, we have found that it uses a VE850 as MediaLogic offers consulting services on this unusual architecture.

We also discover that the PCB is marked with another company name (hidden by the LCD) and model: SKYDIGITAL LockDown. A quick search reveals the original product.

\(^1\)But not necessarily resistant to side channel attacks.
2.1.2 First steps: basic crypto checks

Once a hard drive is inserted in the SHE500 enclosure, it as to be “associated” with it. End user is prompted for a PIN during association. A SHE500 can be associated with up to 50 different hard drives. Once the drive is “associated”, it is ready for use and a master key is displayed to the user (see figure 4).

According to user manual, this master key is for data rescue purposes. If the PIN is lost, Zalman is then able to recover the user’s data provided that the HDD is working. This hints that the secrets are not stored on the drive itself but in the enclosure, which is confirmed by tests.

Also, basic crypto checks show that data seems to be actually encrypted and that the encryption key is not directly derived from the PIN. However, the
master key is suspiciously short, with only 64 bits, where AES-256-XTS needs 512 bits of key... We will come back later on this subject.

2.1.3 The update, the updater and the backdoor

Firmware and upgrade utility are available for download on Zalman’s website. Which means we probably will not have to do complex hardware analysis.

The security analysis was performed using:

- an enclosure
- a firmware update
- the firmware updater

The SHE500 updater is a Windows program which communicates with the enclosure using SCSI commands encapsulated in USB packets. FirmwareWriter.exe is a wrapper on fwdu03_sdk.exe, the binary responsible for low-level communication with the device:

Usage: fwdu03 [option...] image-filename

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/INFO</td>
<td>Chip Info.</td>
</tr>
<tr>
<td>/D=n</td>
<td>Device Index(n=0..9)</td>
</tr>
<tr>
<td>/LIST</td>
<td>Device List</td>
</tr>
<tr>
<td>/SNTXT</td>
<td>Use &quot;SN.TXT&quot; file for Serial Number assignments.</td>
</tr>
<tr>
<td>/SNCMDLINE</td>
<td>Use Command Line &quot;xxxxxx&quot; for Serial Number assignments.</td>
</tr>
<tr>
<td>/UPDATE</td>
<td>F/W Update(Write Only F/W in Image File).</td>
</tr>
<tr>
<td>/BINIMG</td>
<td>Write Binary Image File to SRQM.</td>
</tr>
<tr>
<td></td>
<td>&quot;xxxxxx&quot;h is start address.</td>
</tr>
<tr>
<td></td>
<td>&quot;yyyyyy&quot;h is writing size.</td>
</tr>
<tr>
<td></td>
<td>Oh/non-designation : writing size = file size</td>
</tr>
</tbody>
</table>

But fwdu03_sdk.exe also has hidden commands that allow the user to perform a full dump of device memory, even if the PIN protection is enabled and the drive locked:

- /MEMDUMPALL command: dumps microcontroller’s RAM
- /ROMDUMPALL command: dumps microcontroller’s flash

Output of /MEMDUMPALL command:

*** MEM ALL DUMP (adr = 03FF0000h len = 00004000h) ***

Output of /ROMDUMPALL command:

*** FLASHROM ALL DUMP (adr = 00000000h len = 00200000h) ***
These functionalities are in fact implemented using a debugging feature/backdoor in the MediaLogic chip that allows reading and writing arbitrary parts of RAM or flash, using SCSI with vendor-specific control code over USB:

```python
def DumpMemory(dev, dump_addr, dump_size):
    # sg_raw -r 1k /dev/sg3 fd 09 03 ff 15 a4 00 02 00 00
    # AAAAAAAAAA = address
    # LL LL = len
    dump_size = dump_size & 0xFFFF
    cmd = struct.pack(">BBI", 0xFD, 0x09, dump_addr) +
          struct.pack("<I", dump_size)
    output = py_sg.read(dev, cmd, dump_size)
    return output
```

By coding our own memory dumper, we were able to perform a full dump of the device memory, bigger than the one produced by `fwdu03_sdk.exe`. In particular, several parts of memory contained code that is not present in the firmware update file, which was very helpful to reverse engineer as we were able to map the code at correct locations and build cross references to actual data and initialized variables.

As a side note, we also discovered the following string in the dump:

```
04003E50 52 65 6E 65 73 61 73 20 75 50 44 37 32 30 32 33 Renesas uPD72023
04003E60 30 20 20 20 20 20 20 20 30 32 30 00 00 00 00 00 200....
```

The MLDU03 is actually a rebranded Renesas μPD720230 bridge! But the data sheet is also not available, so it does not change anything.

### 2.1.4 Secrets: PIN

We now have all the elements needed to assess the security of the secrets. In particular, by diffing the memory before/after associating a drive, as described previously, we discovered how the enclosure stores secrets.

It writes structures to its flash, one for every HDD the enclosure is associated with. These structures are located at offset 0x30000. They are composed of a 10 bytes magic `SKYDIGITAL`, 0x3F6 bytes pseudo-random buffer and a 0x90 bytes long encoded buffer.

```
00030000 53 4B 59 44 49 47 49 54 41 4C E6 EE F6 FE 07 11 SKYDIGITAL.....
00030010 19 21 29 31 39 41 49 53 5B 63 6B 63 6B 73 7B 83 8B 95 .!)19AIS[cks...
```
As you can see, the patterns are evident and indicate that the “crypto” is probably weak.

By searching for SKYDIGITAL magic in the firmware we were able to locate the function responsible for decoding the structure (see figure 5).

The algorithm to decode the structure is quite simple but non-standard. It uses a hard-coded vector and a random buffer to generate a key, which is later used to decode (using xor) the relevant part of the structure at offset 0x400.

This algorithm is equivalent to the following C pseudo-code:

```c
uint8_t vector = {
    0xAF, 0x68, 0x11, 0x18, 0x70, 0x48, 0xC1, 0x6C,
    0xF7, 0x1B, 0xA2, 0xE1, 0x50, 0x46, 0x5C, 0xCE
};
uint8_t key[16] = {0};
uint8_t *random_buf = flash_read(0x30200, 0x200);
for (i=0; i<16; i++) {
    key[i] = random_buf[vector[i]];
}
uint8_t *secret_struct = flash_read(0x30400, 0x200);
xor_buffers(secret_struct, key, 0x200);
```

After decoding, the structure appears in clear form:

```
00000000  53 31 30 55 4A 44 30 50 38 32 36 37 31 35 20 20 S10UJDOP826715
00000010  20 20 20 20 53 41 4D 53 55 54 4E 47 20 48 4D 31 36  SAMSUNG HM16
00000020  30 48 49 20 20 20 20 20 20 20 20 20 20 20 20 20 20  OHI
00000030  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20  E7 CF 6A 00  ....
00000040  59 7D 7F 9B 53 31 30 55 4A 44 30 50 38 32 36 37  Y}..S10UJDOP8267
00000050  31 35 20 20 20 20 20 20 20 20 53 41 4D 53 55 4E 47 20 15  SAMSUNG
```
Figure 5: Decoding structure in firmware

Obviously, the structure contains the model name and the serial number of the associated hard drive. Less obvious, but nevertheless very important, it also contains the user’s PIN and AES-XTS keys, which we will cover later.

The code responsible for checking the PIN is not in the firmware file, we found it in the RAM at high addresses (over 0x04000000).

The current masked PIN is stored at offset 0x80 of the struct (9BE4EDC9 in our example). Note that it may be different from the PIN used to setup the drive, as the user is able to change PIN at anytime. The PIN is not stored directly but encoded using this algorithm, taking the PIN as a 32-bit integer as input:

```python
def reveal_pin(input_pin, serial_number, hw_model):
    var_1AEAD = [0x87, 0xF4, 0x5E, 0xCD]
    for i in range(4):
        c = input_pin[i]
        c = c ^ var_1AEAD[i]
        input_pin[i] = c

    for j in range(5):
        x = serial_number[4*j+1]
```

00000060 48 4D 31 36 30 48 49 20 20 20 20 20 20 20 20 20 HM160HI
00000070 20 20 20 20 20 20 20 20 20 20 20 20 E7 CF 6A 00 ....
00000080 C9 ED E4 9B 00 03 00
The PIN is stored xored with a constant dword (0xCD5EF487), drive model name and drive serial number, in our case S10UJD0P826715 and SAMSUNG HM160HI. The reveal_pin converts the input 0x9BE4EDC9 into a decimal number 33440000. When user inputs PIN, digits are stored in memory and the converted to a 32 bits number, PIN digit 0 is encoded as 1, PIN digit 1 is encoded as 2 and so on. This means decimal number 33440000 is the representation of a four digit PIN: 2233. This also means PIN collisions are possible. The following script does the final computation and outputs all possible PIN collisions:

```python
def compute_pins(dword, s1, s2):
    s = struct.pack("<I", dword)
    input = [ord(s[0]), ord(s[1]), ord(s[2]), ord(s[3])]
    output = reveal_pin(input, s1, s2)

    s = output[3] << 24
    s += output[2] << 16
    s += output[1] << 8
    s += output[0]

    s = "%08d" % s

    pins = []
    n = int(s)
    for l in range(4, 9):
        m = 10**(8-l)
        if n % m == 0:
            a = int("1"*l)*m
            if 0 <= n-a < 100000000:
                pin = (n-a)/m
                pins.append(("%0%ii" % l) % pin)
    return pins
```

In our example, 0x9BE4EDC9 has five PIN collisions 2233, 22329, 223289, 2232889 and 22328889.

2.1.5 Secrets: master key
We also located code responsible for computing the master key.

```python
hardcoded_value = [0x9E, 0x87, 0xAB, 0xC4,
```
def print_master_key(vector):
    c0 = vector[0] ^ vector[4] ^ hardcoded_value[0]
    print "%02X"*8 % (c0, c1, c2, c3, c4, c5, c6, c7)

The master key is the result of encoding two dwords: 0x6ACFE7 and 0x9B7F7D59. We are not sure how the first is generated but we think it is time dependent as we noticed it does not change much from one drive to another.

The second dword (0x9B7F7D59) is the masked PIN entered by the user when associating the drive (in our case 1234).

Careful readers are probably wondering how this master key can help recover HDD data since its size is very small: only 64 bits. This leads us to the final, and probably most important, data stored in the structure: AES-XTS keys used to encrypt the whole drive.

2.1.6 Secrets: AES-XTS keys

The AES keys are indeed stored in the structure. However we did not realize they were there until we found the piece of code copying the keys in the MMIO of the crypto chip (see figure 6).

Figure 6: Copy of AES keys to chip MMIO

The device uses AES in XTS mode, thus working with two 256-bit keys. Figure 7 shows the two 256-bit keys:

As surprising as it can be, those two keys are composed of:
- dword 0x6ACFE7, time-dependent
- dword 0x9B7F7D59, first user PIN, masked
- HDD model name
2.1.7 Attack 1: Instantaneous PIN recovery

The first attack scenario is the instantaneous PIN recovery. Physical access to the enclosure is required. Using the MediaLogic backdoor one can read the secret structure in the device’s RAM. In our firmware build the masked PIN is stored in RAM at address $0x3FF1624$. It is then straightforward to recover the PIN using the algorithm presented in previous section.

As a result, an attacker is able to: recover the user’s PIN, unlock the drive and access user’s data instantly.

2.1.8 Attack 2: Offline hard drive decryption

With physical access to an encrypted hard drive, even without the enclosure, it is possible to conduct a brute force attack on the PIN to recover the encryption keys.

Indeed, as the model name and serial number are available, the only missing parts to recover the encryption keys are the 64 bits of the first XTS key, in our example the two dwords $0x6ACFE7$ and $0x9B7F7D59$. As the first dword $0x6ACFE7$ does not vary much from one drive to another, we empirically reduced entropy to 16 bits. The other dword is the the first user PIN, encoded.

The SHE500 allows PINs between 4 and 8 digits. So the total entropy is about 42 bits. We used the first sector as input data and as stop criteria:

- $0x55AA$ marker
- a high number of zeros in the decrypted block, which is the case for usual boot sectors.

In practice we can test all PINs for a given first DWORD in about 2s on a desktop Core i7, which translates to 36 hours maximum. But this could probably be done much faster by reducing the possible values for the first dword.

The bruteforce is also helped by the fact that $Key2$ is constant: one can precompute the xor masks to apply and only use AES on the first key.
2.2 Zalman ZM-VE500

2.2.1 Hardware

First, as on the SHE500, we open the enclosure to look at the PCB to identify the various components, as shown on figure 8.

![Figure 8: VE500 PCB front and back](image)

Again, we find that the drive uses the simple architecture with essentially 3 chips:

- a Initio INIC-3607E USB-SATA bridge (no public info)
- an unmarked capacitive keyboard controller
- a Pm25LQ032 SPI flash [data sheet](data_sheet)

The data sheet is not available for the INIC controller, but public info shows that it uses an ARCompact CPU. Also, this time the PCB is branded with the real model name, so it is not just a case rebranding.
### 2.2.2 First steps: basic crypto checks

We first start with the usual basic testing:

- checking for actual encryption,
- checking for change in disk size or behavior,
- configuring the same PIN several times and verifying the encryption is different.

The results are interesting: encryption seems OK but while the drive does not show any change in its advertised size, reading the last sectors does not return any data.

Reading them directly with a normal USB-SATA bridge gives us the following (with high entropy sectors edited for brevity):

<table>
<thead>
<tr>
<th>Sector</th>
<th>Hexadecimal</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>01e0100</td>
<td>2a bd 1d 45 87 90 53 84 dd f7 a2 81 5f 2e 38 48</td>
<td>...E..S.....-8H</td>
</tr>
<tr>
<td>01e0101</td>
<td>9f 19 5a 5c 20 2b 53 dc 9e 8e f5 1f ca c5 4a</td>
<td>...Z+S.....J</td>
</tr>
<tr>
<td>01e0102</td>
<td>43 b5 90 49 67 f2 c5 bh 90 66 9b 2c c8 2a 32</td>
<td>C..Io..C..=f.</td>
</tr>
<tr>
<td>01e0103</td>
<td>ab b2 6f 2a 5f 34 5b bb af 5e 4a 4a 44 66 e8 21</td>
<td>..._*4[^JJ.f.!</td>
</tr>
<tr>
<td>01e0104</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>........................</td>
</tr>
<tr>
<td>01e0200</td>
<td>20 49 e4 49 3a 00 00 00 f3 ac 93 a1 28 00 00 00</td>
<td>INI:......(....</td>
</tr>
<tr>
<td>01e0201</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>........................</td>
</tr>
<tr>
<td>01e0202</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>........................</td>
</tr>
<tr>
<td>01e0203</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>........................</td>
</tr>
<tr>
<td>01e0204</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>........................</td>
</tr>
<tr>
<td>01e0205</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>........................</td>
</tr>
</tbody>
</table>

We have two similar blocks composed of two parts:

- one high entropy sector of 512 bytes,
- one very low entropy part with a "INI" magic at the beginning.

We make the hypothesis that everything needed for encryption is stored in the encrypted blobs, which is easily verified by testing the drive in another enclosure and entering the correct PIN: everything works.

Our goal from now on will be to identify how the blobs are generated and encrypted so we can mount an offline attack to bruteforce the PIN code.
2.2.3 Firmware analysis

Fortunately for us, the firmware upgrade utility is available on the web site and seems to contain an unencrypted firmware.

The microcontroller is an ARCompact which is thankfully supported by IDA. As the firmware file starts with 0x4000 bytes of 0xFF, we load the file from this offset in IDA, hoping that the base loading address will not be too difficult to identify. The result can been seen in figure 9.

![Figure 9: ARCompact firmware loaded in IDA](image)

Luckily, the base address seems to be 0 as most cross references are identified. Also, the start address at 0x400 looks promising (figure 10). The gp based addressing is not handled by IDA so a small Python script was developed to resolve all globals.

![Figure 10: VE500 startup code](image)

Now that we have (almost) everything we need, we can start reversing the firmware, looking for:

- crypto chip init (but we do not have data sheets)
- disk I/O (no data sheet...)

15
• PIN input
• menu entries

Of course since the crypto is handled in hardware, looking for known constants gives no results. So the easiest way in is through strings displayed on the LCD. In particular, the function handling the menu display may help discovering the role of some globals and functions, but its CFG is not pretty (figure 11).

![CFG of VE500 menu handling function](image)

After a lot of reversing work, particularly around the “Wrong PWD” strings, several things were identified:

• crypto processor MMIO addresses,
• two weird AES keys (\(\pi\)), figure 12
• a comparison of a (seemingly) decrypted block with a “INI” magic

```assembly
Pi_key_256_bits:.byte 3,0x14,0x15,0x92,0x65,0x35,0x89,0x79# 0
# DATA XREF: memcpy_Pi_key
.byte 0x32,0x38,0x46,0x26,0x43,0x38,0x32,0x79# 8
.byte 0xFC,0xE8,0x8A,0x6D,0x9A,0xCA,0x76,0x85# 0x:
.byte 0xCD,0xC7,0x89,0xD9,0xBC,0xC7,0xCD,0x85# 0x:
Pi_key_128_bits:.byte 3,0x14,0x15,0x92,0x65,0x35,0x89,0x79# 0
# DATA XREF: memcpy_Pi_key
.byte 0x2B,0x99,0x2D,0x9F,0xA2,0x32,0x49,0xD5# 8
```

Figure 12: Hardcoded AES keys

### 2.2.4 “Security” scheme

So, in the end, the PIN verification scheme can be summarized as:

1. get PIN in 8 byte array, 0 padded
2. `memcpy(aeskey, pin, 8)`: copy 8 bytes of PIN, zero-padded, at the beginning of the key

3. configure HDD crypto engine with AES-256-XTS with:
   - AES π key partially overwritten with PIN as key 1
   - “0” (yes, 256 bits of zeros) key as key 2
   - sector number as tweak

4. read “secret” block through crypto engine

5. check for magic “INI”

So, in theory, writing a bruteforcer is trivial: just read the secret block and try all PINs until the correct one is found. But of course, in practice, our bruteforcer did not stop.

This meant that we had to dig deeper to identify what was wrong: our hypothesis or a particular implementation detail?

In particular, we located the disk access function, using the constants from the ATA command set. It helped us understand how the firmware accesses encrypted sectors at the end of hard drive, using ATA “READ DMA EXT” and “WRITE DMA EXT” commands (figure 13).

![Figure 13: Firmware builds DMA read/write ATA command](image)

We also reversed most of the global states variable of the menu to ensure we were not missing any interesting code path and soon reached the conclusion that a static only approach was not enough.

2.2.5 LCD debugging

So, how can we debug and try to find what’s wrong? The easiest way we found was to:

- patch the firmware to display memory on the LCD
- patch the firmware to change AES options and keys

As we were not 100% sure of some things, we wanted to have a means of displaying memory, but since the chip is very small and we had no data sheet, the only easy thing was to use the `DisplayStr` function to display arbitrary memory.
Figure 14: Example of memory dumping using the LCD

(figure 14). As the firmware is not signed and only checked by a CRC16, we could reflash arbitrary code, trying not to brick anything.

Using it, we verified that we were using the right keys and the right tweak for the AES function.

2.2.6 AES patching

The other things we did were to modify the AES parameters in the firmware. Thus trying to understand exactly how they are used by the crypto chip, using a combination of:

• fixing the XTS tweak to 0 to avoid any endianness issue,
• switching to AES-ECB mode to use only one key,
• set the AES keys to known values and compare output to AES test vectors:
  – all zeros
  – only one byte to 0x01, in various positions
  – 01 02 03 04 00 ... , to test for endianness

After a lot of disk swapping between the VE500 and a normal bridge, we discovered that although the key is passed “normally” to the crypto engine, the actual key used for encryption is:

• byteswapped: 01 02 03 04 05 06 07 08 becomes 08 07 06 05...
• key 1 and key 2 are swapped

The tweak is the actual sector number, in little endian.

2.2.7 Bruteforcing

Implementing the bruteforcer is rather trivial, but interestingly, since the padding of the PIN used in the key is made with zeros and as the digit “0” is encoded as 0x00, all PINs less than 4 chars can be padded with 0 and still be considered as valid.

Performance: an OpenSSL based bruteforcer, parallelized with OpenMP and running on a desktop Intel i7 bruteforces all PIN candidates in about 6s.

As Key2 is only zeros, we can apply the same optimization we used on the SHE500 and only bruteforce Key1.

Firmware 2.0 When firmware 2.0 was released, we tried our bruteforcer but it did not work anymore. After a bit of reversing, we found that the PIN is now padded with 0xFD instead of 0x00 to avoid PIN collisions.
2.2.8 Conclusion

Once again the design itself is broken from the start:

- storing the secrets on the disk allows for immediate access (with open case) to the attacker
- the absence of real key derivation from the PIN allows for trivial brute-forcing
- 0-padding in version 1 of the firmware implies PIN collisions

So against our attacker model, the drive is broken against an attacker who can access the enclosure and open it.

We did not fully understand how the actual data encryption key are generated but the “secret” block, once decrypted, contains high entropy material which could be used as AES keys.

2.3 Zalman ZM-VE400

The VE400 was initially studied in [CR15] using electronic methods as the firmware is encrypted. Further studies of the Fujitsu MB86C311 were done on another drive in [OF16].

While it was not broken initially, it is interesting to compare its design to the other drives covered: as with the VE500, the secrets are stored on the drive, and a bruteforce attack would most probably be feasible as it also limits the PIN to 8 digits.

However, two independant factors have rendered the complete attack way more complex compared to the VE500:

- the bridge’s firmware encryption,
- the PIC32MX protection bits.

As recovering the code is difficult, we could not reverse engineer the details of the PIN verification to implement either a software bruteforcer or a fake keyboard.

3 Going further

3.1 Suppliers

3.1.1 AES core

During our investigation, we have found troubling similarities between different designs, in particular in the work of [AKm15]. Several chip manufacturers, apparently completely unrelated, have exactly similar things:

- same AES π key
- same AES mode constants

[AKm15] show that:

- JMicron chips (JMS538S), used by Western Digital mainly
- Initio chips (1607E, 3607E), used by WD, Lenovo, Apricorn, Zalman, etc.
- PLX chips (OXUF943SE), used by WD

include the same keys.
The AES constants can also be found in the “WD security” application for Mac, in some C headers file:

```c
enum {
    NO_CIPHER = 0x00,
    AES_128_ECB = 0x10,
    AES_128_CBC = 0x12,
    AES_128_XTS = 0x18,
    AES_256_ECB = 0x20,
    AES_256_CBC = 0x22,
    AES_256_XTS = 0x28,
    FULL_DISK_ENCRYPTION = 0x30
};
```

Our hypothesis here is that a single AES-XTS “IP” (i.e. Verilog or VHDL code) was used by all manufacturers and that it comes with example code that used those AES keys.

This would imply that a single core is used by the major providers on the market, which would all be vulnerable if the core itself was flawed.

**Certifications, FIPS** Another interesting point is that many chips are “FIPS certified”, but (most of the times) it only means that their core correctly implements AES and it gives no information about the actual security.

### 3.1.2 PCB

It is also interesting to note that the brand visible on the outside does not mean the actual PCB was designed by the company.

Several rebrandings we know of:

- at least some iStorage products: manufactured by Apricorn
- Lenovo ThinkPad Secure Hard Drive: manufactured by Apricorn
- Zalman ZM-VE400: manufactured by iODD
- Zalman ZM-SHE500: manufactured by Sky Digital
- Zalman ZM-SHE350: manufactured by Sky Digital

So, actually knowing what you are buying is not easy.

### 3.2 A “secure” design

Some manufacturers want their users to be able to replace the enclosure and still be able to access their data. In that case, two options are possible:

---

2Which also include a lot of SCSI commands and definitions.
• store the secrets on the enclosure and display a recovery key to the user
• store the secrets on the drive.

**User-friendlyness : secrets on disk** With secrets on the drive, no truly secure design can be done as, once the reverse engineering work is done, brute-forcing the PIN will only necessitate access to the drive.

The only thing is to try to slow the hash as much as possibly by hashing the PIN incrementally, using the fact that the user takes time between each digit and of course by allowing long PIN lengths.

**Storing secrets on the enclosure** As without *hardware help* we are essentially screwed (only long PIN could be secure), the basic idea is to make it harder for the attacker to access the secrets by either:

• storing them on a component that cannot be read programmatically,
• storing (even on the drive) them encrypted with a random key stored in a component.

For cheap models, this can be achieved by using an external microcontroller with internal storage protected against external access such as PIC and AVR. But such chips can be read for a few thousands of USD.

However, the best design is to use a secure component with an integrated crypto engine with a fuse programmable key:

• provision the micro-controller with a random AES key (fuse blowing)
• encrypt the PIN’s hash with the micro-controller keyed AES-engine

Add some brute-force counter measures and firmware signature.

Thus an attacker wishing to bruteforce a long PIN has to recover the secret AES key from the fuses using destructive and expensive hardware attacks.

### 4 Conclusion

This overview of several drive enclosures leads to a depressing conclusion:

• (almost) no enclosure can be trusted without an independent audit.

As the cost requirements of the cheaper drives probably prevent such audits, it would be better for the manufacturers to:

• hire cryptographers !
• publish their crypto design
• allow for PIN longer than 8 digits (!)

Even though with cheap devices like those we cannot expect a very high security level, it can be improved with very low investment. We were also very surprised to see such bad coding practices and lack of crypto knowledge.
References


