RAID Reconstruction
And the search for the Aardvark

Dr Michael Cohen

This talk does not represent my Employer

April 2005
What is RAID?

- **RAID 0: Striping**
  - Improves performance due to parallel disk access
  - No redundancy – Lose a single disk=lose data
  - Capacity of array = $n \times$ capacity of each disk

RAID-0 Configuration

<table>
<thead>
<tr>
<th>Physical Block</th>
<th>Disk 1</th>
<th>Disk 2</th>
<th>Disk 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
• **RAID 1: Mirroring**
  - Improves read speed – not write speed
  - Full redundancy – May lose any number of disks but one
  - Capacity of array = capacity of one disk
• **RAID 5: Striping with parity**
  - Strikes a balance between redundancy, speed and capacity
  - Data is stripped across all disks and includes a parity block:
    - \( A \ xor \ B \ xor \ C = 0 \)
  - Can lose one disk from the array and recover the data.

![RAID-5 Configuration](image-url)
Why would we care?

• **Image acquisition:**
  - RAID arrays are common in server class machines
  - Traditional techniques involve booting the original hardware and imaging over network/USB
    - This is typically slow
    - Can’t use imaging hardware like Logicube etc.

• **Data recovery:**
  - Some controllers can mark disks as bad when they detect a fault with the disk:
    - Often the fault is intermittent.
    - A drive may be repairable.
Sounds easy?

- We can just reassemble the striping pattern as in the previous figure?
  - Unfortunately there is no standard striping pattern – Each controller has its own pattern
  - Often the order of the disks in the array is nothing like the labeling on the disk
  - Sometimes we are missing one drive – we should be able to rebuild the array without it.
Definitions:

• **Block**
  - Logical Block
  - Physical Block

• **Array period**
  - Physical array period
  - Logical array period
  - Slot

• **Striping Map**
How do we reassemble the array?

- **We need to establish the block size**
  - Look through the data for obviously disjoint dis-continuities

<table>
<thead>
<tr>
<th>Disk1</th>
<th>Disk2</th>
<th>Disk3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1dfe0 ..S(....Ns....wh</td>
<td>s.S(.hre(.......</td>
<td>s....hre....re</td>
</tr>
<tr>
<td>0x1dff0 ite(....(....s..</td>
<td>...Kes..(.pos.</td>
<td>p</td>
</tr>
<tr>
<td>0x1e000 ..self(....(....</td>
<td>.as.possible.so.</td>
<td>a.E..[siblM.so.</td>
</tr>
<tr>
<td>0x1e010 s5.../home/mic/p</td>
<td>it.may.be.instal</td>
<td>A.maVH..EF...N.</td>
</tr>
<tr>
<td>0x1e020 yflag_website/py</td>
<td>led.with.as.few.</td>
<td>A.6..B..T.J.Y</td>
</tr>
<tr>
<td></td>
<td>.....</td>
<td></td>
</tr>
<tr>
<td>0x1efe0 pts.to.include.a</td>
<td>ill.then.be.inde</td>
<td></td>
</tr>
<tr>
<td>0x1eff0 s.many.libraries</td>
<td>xed.during.scann</td>
<td></td>
</tr>
<tr>
<td>0x1f000 ...Gedz....HI...</td>
<td>ing...Usage:..load</td>
<td></td>
</tr>
<tr>
<td>0x1f010 K6....G....[.$.T</td>
<td>d_dictionary.py.</td>
<td></td>
</tr>
<tr>
<td>0x1f020 46....SA.Ls.I..]</td>
<td>[Dictionary.file</td>
<td></td>
</tr>
<tr>
<td></td>
<td>orials&lt;/a&gt;..&lt;/td&gt;</td>
<td></td>
</tr>
</tbody>
</table>
Demo
Establish the RAID map

• Common pitfalls to avoid:
  – Block that contain zeros will copy the parity block – so it will be difficult to tell them apart.
  – Regions of the disk containing random data can be indistinguishable from parity and are best avoided.

• Bonus:
  – Finding a dictionary makes life really easy because we are able to follow the block order alphabetically:
    • Search for Aardvark
  – For example /usr/share/dict/american-english is 900kb, with a 4k block size this will cover over contiguous blocks.
# Build the RAID Map

<table>
<thead>
<tr>
<th>Block</th>
<th>Disk 1</th>
<th>Disk 2</th>
<th>Disk 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>30:</td>
<td>0</td>
<td>1</td>
<td>P</td>
</tr>
<tr>
<td>31:</td>
<td>P</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>32:</td>
<td>5</td>
<td>P</td>
<td>4</td>
</tr>
<tr>
<td>33:</td>
<td>6</td>
<td>7</td>
<td>P</td>
</tr>
<tr>
<td>34:</td>
<td>P</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Determine the period

- **The period is the number of blocks before the pattern starts repeating**
  - It's usually easy to determine the period by looking at the parity.
  - In the previous case, the period was 3 so the slot map was:

<table>
<thead>
<tr>
<th>Block</th>
<th>Disk 1</th>
<th>Disk 2</th>
<th>Disk 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>30:</td>
<td>0</td>
<td>1</td>
<td>P</td>
</tr>
<tr>
<td>31:</td>
<td>P</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>32:</td>
<td>5</td>
<td>P</td>
<td>4</td>
</tr>
</tbody>
</table>
Recovering the partition table

- **Determine the start of the partition in the logical image:**
  - Usually there is a partition table at the start of the logical image:

  ```
  ~/pyflag$ ./bin/mmls -t dos d1.dd
  DOS Partition Table
  Units are in 512-byte sectors
  
<table>
<thead>
<tr>
<th>Slot</th>
<th>Start</th>
<th>End</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:</td>
<td>0000000000</td>
<td>0000000000</td>
<td>0000000001</td>
<td>Primary Table (#0)</td>
</tr>
<tr>
<td>01:</td>
<td>0000000001</td>
<td>00000000062</td>
<td>00000000062</td>
<td>Unallocated</td>
</tr>
<tr>
<td>02:</td>
<td>00:00</td>
<td>00000009829</td>
<td>0000009767</td>
<td>Linux (0x83)</td>
</tr>
<tr>
<td>03:</td>
<td>00:01</td>
<td>00000020479</td>
<td>0000010650</td>
<td>Win95 FAT32 (0x0C)</td>
</tr>
</tbody>
</table>
  ```
PyFlag Hooking infrastructure

• **PyFlag is a GPL forensic software**
  - There are many types of forensic images, in lots of different formats:
    • dd format (disk/partition)
    • Encase format
    • RAID Format
  - There are many different types of forensic tools
    • Sleuthkit
    • PyFlag
    • Lookback mounting through the kernel
    • md5sum, file, hexedit
    • Exgrep, foreman, strings
Putting them together

• **We need to use Forensic Tools on Forensic Images**
  - It would be best if we can use standard forensic tools on all kinds of images
  - We do not want to change the tools:
    - Although many tools are Open Source, it would be a nightmare to maintain patches against all of them.
  - We wish to enhance the functionality of arbitrary tools without touching their source code? Is this possible?
Hooking IO For fun and profit

- We can do this by using a technique called library hooking:
  - Consider a program:

```c
main() {
    fd=open("somefile",O_RDONLY);
    read(fd,buffer,SIZE);
    close(fd);
}
```
How does this work?

Normal Operation

Program → C Library → Kernel

open() open syscall

Hooked program

Program → Hooker → C Library → Kernel

open() Driver calls open()
How does this work?

- The hooker library intercepts standard library calls related to IO (open, read, write etc)
- When the application wants to issue an IO function, the hooker uses an appropriate driver to access the raw image depending on its format
- The Hooker returns the data to the program as if the image was a simple linear block of data.
- Parameters to the hooker driver are passed in through environment variables.
- An iowrapper is responsible for arranging the hooker and its environment variables.
- The hooker can use a number of drivers (We call them subsystems).
Example:

- Let us see the hexdump of an Encase image:

```
~/pyflag$ ./bin/iowrapper -i ewf -filename=test.e01 -- hexdump -C foo | head
00000000 45 56 46 09 0d 0a ff 00 01 01 00 00 00 68 65 61  |EVF...y........|
00000010 64 65 72 00 00 00 00 00 00 00 00 00 00 00 b2 00 00  |der...............|
00000020 00 00 00 00 00 a5 00 00 00 00 00 00 00 00 80 00 10  |......¥...........|
```

```
~/pyflag$ hexdump -C test.e01 | head
00000000 45 56 46 09 0d 0a ff 00 01 01 00 00 00 68 65 61  |EVF...y........|
00000010 64 65 72 00 00 00 00 00 00 00 00 00 00 00 b2 00 00  |der...............|
00000020 00 00 00 00 00 a5 00 00 00 00 00 00 00 00 80 00 10  |......¥...........|
```

```
~/pyflag$ ./bin/iowrapper -i ewf -filename=test.e01 -- hexdump -C foo | head
00000000 eb 3c 90 4d 53 44 4f 53 35 2e 30 00 02 01 00 00 68 65 61  |ë<.MSDOS5.0......|
00000010 02 e0 00 40 f0 09 00 12 00 02 00 00 00 00 00 00 b2 00 00  |.à.@.@...........|
00000020 00 00 00 00 00 29 fc 02 29 08 4e 4f 20 4e 41  |.......ü).NO NA|
00000030 4d 45 20 20 20 20 20 20 46 41 54 31 32 20 20 20 33 c9  |ME FAT12 3É|
```
Example

• Let us convert an Encase image to a dd image:
  - The -f ensure we only hook files called foo – this allows dd to create the output file.
  - blocksize is 64k to ensure faster performance.

```bash
~/pyflag$ ./bin/iowrapper -i ewf -filename=test.e01 -f foo -- dd if=foo of=out.dd bs=64k
```
Remote Subsystem

- We can trick local programs to read disk images directly off remote systems:
  - Program being hooked receives data across the network from a remote servlet.
  - The servlet provides direct access to the raw disk on the remote machine
Remote Subsystem

• **Advantages:**
  - We do not need to copy the entire image prior to analysis – we can analyse the remote image un-intrusively in place.
  - Since the local program reads the filesystem structures directly, rather than relying on remote system's kernel system calls – we are immuned from kernel level rootkits on the remote system hiding files.

• **Disadvantages:**
  - Slow operation due to network latency.
  - Possibilities of race conditions since we are operating on a live system.
Example

```bash
~/pyflag$ ./bin/iowrapper -i remote -host 127.0.0.1 -user root -server_path /bin/remote_server -device /dev/hdc -- mmls foo
```

DOS Partition Table
Sector: 0
Units are in 512-byte sectors

<table>
<thead>
<tr>
<th>Slot</th>
<th>Start</th>
<th>End</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:</td>
<td>----</td>
<td>000000000000</td>
<td>000000000000</td>
<td>000000000001 Primary Table (#0)</td>
</tr>
<tr>
<td>01:</td>
<td>----</td>
<td>000000000001</td>
<td>000000000062</td>
<td>000000000062 Unallocated</td>
</tr>
<tr>
<td>02:</td>
<td>00:00</td>
<td>000000000063</td>
<td>0000096389</td>
<td>0000096327 Dell Utilities FAT (0xde)</td>
</tr>
<tr>
<td>03:</td>
<td>00:01</td>
<td>0000096390</td>
<td>0019647494</td>
<td>0019551105 NTFS (0x07)</td>
</tr>
</tbody>
</table>
Back to reassembling RAID:

• Through the use of hooking, we can use arbitrary programs with the RAID driver transparently:
  – The IO Hooker hooks system calls to enable other tools to work with the reassembled image.
  – Example:

  
  ./bin/iowrapper -i raid -blocksize 4k -slots 3 -offset 63s -map 0.1.P.P.2.3.5.P.4 -filenames d?.dd -- ./bin/fls -r -f linux-ext2 foo
Recover the Logical Image:

- Use PyFlag to hook dd to copy the logical image to a dd image:

  ./bin/iowrapper -i raid -blocksize 4k -slots 3 -offset 63s -map 0.1.P.P.2.3.5.P.4 -filenames d?.dd -- dd if=foo > /tmp/recovered.dd
Is there a shortcut?

- **Shortcuts make life easier**
  - Sometimes the shortcuts don't work
  - If you can save 2 hours by trying something that takes a few seconds it might be worth it.

- **PyFlag's hooker reassembles the array on the fly:**
  - The logical image is reassembled on an as needed basis by the external program.
  - It takes no time to try.

- **We can brute force the RAID parameters!!!!**
Brute Forcing RAID Parameters

• Can we tell when the parameters are wrong?
  – We need a program that gives an indication of success.
  – We use it to guess the map from a library of possible maps
  – We need to assume a number of slots and a block size:

```shell
~/pyflag$ ./launch.sh utilities/raid_guess.py -o 63s -s 3 -b 4k d1.dd d2.dd d3.dd
```
What if one of the disks is dead?

- PyFlag can recover a single disk from a RAID 5 array:
  - If PyFlag fails to open the file, it assumes its missing:

```
./bin/iowrapper -i raid -blocksize 4k -slots 3 -offset 63s -map 0.1.P.P.2.3.5.P.4 -filenames d1.dd d2.dd foo
-- ./bin/fls -r -f linux-ext2 foo
```

Set number of slots to 3
Set file number 0 as d1.dd
Set file number 1 as d2.dd
Set file number 2 as foo
Could not open file foo, marking as missing
What else can go wrong?

- Sometimes there is a header at the start of the disk:
  - Common with software RAID (e.g. Microsoft LDM, HP Smart Array)
  - You can find those by searching for the partition table:

```bash
bash$ mmls -t dos header.dd
DOS Partition Table
Units are in 512-byte sectors

<table>
<thead>
<tr>
<th>Slot</th>
<th>Start</th>
<th>End</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:</td>
<td>0000000000</td>
<td>0000000000</td>
<td>0000000001</td>
<td>Primary Table (#0)</td>
</tr>
<tr>
<td>01:</td>
<td>0000000001</td>
<td>0000000062</td>
<td>0000000062</td>
<td>Unallocated</td>
</tr>
<tr>
<td>02:</td>
<td>00:02</td>
<td>0000016064</td>
<td>0000016002</td>
<td>Hibernation (0x12)1</td>
</tr>
</tbody>
</table>
```
How to find the header

- We can guess the header size by trying to find the logical image's partition table:

bash$ for i in `seq 0 4 1000`; do \
    echo trying offset ${i}k; \
    ./bin/iowrapper -i advanced -offset ${i}k -filename d1.dd -- \
    ./bin/mmls -t dos foo; \
done 2>&1 | less
Can we Hook the kernel?

Sometimes it would be nice to mount the image over the loopback device:

- This uses the kernel's own filesystem drivers – supports many more types of filesystems than Sleuthkit.
- The kernel loopback driver does not use the C library to access the image – hence we need to hook in kernel space.
- Fuse (Filesystems in user space) allows us to hook system calls in the kernel directly, passing them to a user space program to implement a virtual filesystem.
Filesystems in User Space

Program → C Library → Kernel

Hooker ← Fuse ← VFS Switch

Kernel FS Driver

User space

Kernel Space
PyFlag's Fuse hooker

• The Fuse hooker hooks IO in the kernel:
  – Works on all programs due to very low level hooking.
  – Creates a pseudo file representing the raw image. This file can then be mounted using the standard loopback device.

```
~/pyflag# ./launch.sh \
./utilities/fuse_loopback_subsystem.py /tmp/mnt/ \
  -i raid -header 0 -blocksize 4k -slots 3 -map \ 
  0.1.P.P.2.3.5.P.4 -offset 63s -filename d?.dd

~/pyflag# mount -o loop /tmp/mnt/mountme /mnt/data/
```
How does this information affect me?

- Traditionally RAID acquisition was difficult and slow:
  - Now we can acquire individual disks in parallel:
    - For example 3TB Apple X-Raid has 14 disks of 250GB each
    - Assuming about 2GB/Min acquisition rate it would take around 4 hours to acquire each disk.
    - Acquiring linearly would take approximately 36 hours and would require a 3.5 TB server to place the image into.
  - Traditional methods involve copying over USB/FireWire or Ethernet – both would achieve far less than 2GB/Min typically.
Conclusions

- Acquiring Large RAID arrays is now practical and possible
- Automated techniques were presented for recovering RAID logical images
- Data recovery of RAID arrays is possible.