File System Block-Mapping under Linux

Daniel Ridge, *newt@scyld.com* ; v1.0.17, 16 March 2000

The Linux kernel includes a powerful, filesystem independant mechanism for mapping logical files onto the sectors they occupy on disk. While this interface is nominally available to allow the kernel to efficiently retrieve disk pages for open files or running programs, an obscure user-space interface does exist. This is an interface which can be handily subverted (with `bmap` and freinds) to perform a variety of functions interesting to the computer forensics community, the computer security community, and the high-performance computing community.

1 Downloading

bmap is publicly available at the following location

- Web page: *http://www.scyld.com/software/bmap.html*
- Source: *ftp://ftp.scyld.com/pub/bmap/bmap-1.0.17.tar.gz*

1.1 Redistribution

2 Usage

The bmap package consists of 2 tools and a development library. The standalone tools `bmap` and `slacker` are provided as both useful standalone utilities and reference implementations of `libbmap` applications.

2.1 Building bmap

`make` and `make install` should take care of it.

At this time, we have only worked out a bmap implementation on Linux.

These tools will install under `/usr/local` by default.

2.2 Invoking the tools

2.2.1 `bmap` invocation

`bmap [<OPTIONS>] [<target-filename>]`

Where **OPTIONS** may include any of:

–doc VALUE

where VALUE is one of:

version
display version and exit

help
display options and exit

man
generate man page and exit

sgml
generate SGML invocation info

–mode VALUE

where VALUE is one of:

map
list sector numbers

carve
extract a copy from the raw device

slack
display data in slack space

putslack
place data into slack

wipeslack
wipe slack

checkslack
test for slack (returns 0 if file has slack)

slackbytes
print number of slack bytes available

wipe
wipe the file from the raw device

frag
display fragmentation information for the file

checkfrag
test for fragmentation (returns 0 if file is fragmented)

–outfile <filename>

write output to ...

–label

useless bogus option

–name

useless bogus option

–verbose

be verbose

–log-thresh <none | fatal | error | info | branch | progress | entryexit>

logging threshold ...

–target <filename>

operate on ...

2.2.2 `slacker` invocation

`slacker [<OPTIONS>] [<path-filename>]`

Where **OPTIONS** may include any of:

–doc VALUE

where VALUE is one of:

- **version**
  display version and exit
- **help**
  display options and exit
- **man**
  display man and exit
- **sgml**
  generate SGML invokation info

–mode VALUE

where VALUE is one of:

- **capacity**
  measure slack capacity of path
- **fill**
  fill slack space with user data
- **frob**
  fill slack space with random data
- **pour**
  write out the contents of slack space
- **wipe**
  clear the contents of slack space

–outfile <filename>

write output to ...

–verbose

be verbose

–log-thresh <none | fatal | error | info | branch | progress | entryexit>

logging threshold ...

–path <filename>

operate on ...

–recursive

descend into subdirectories

2.3 Limitations

The bmap works against filesystems mounted on block devices. You will not to be able to operate against filesystems mounted via Samba, NFS, or any other network filesystem.

If you simply cannot use bmap on the machine storing the block device or block device image, you can try the linux network block driver to export the block device to the machine from which you wish to bmap. Also, Scyld's userfs user-level filesystem code includes a sample application, bush, which is linked against bmap.

2.4 Technical Description / Implementation

2.4.1 VFS – Linux 'Virtual Filesystem Switch'

These notes are based on Linux 2.2.5 but should be widely portable to other versions of the Linux kernel.

compiled with

```
#include <linux/fs.h>
```

lives at

```
((struct inode_operations *)foo)->bmap
```

and is prototyped as

```
int foo_bmap(struct inode *inode,int block);
```

2.4.2 FIBMAP – userspace interface via ioctl

```
#include <sys/ioctl.h>
#include <linux/fs.h>

retval=ioctl(fd,FIBMAP,&block_pos);
```

Where `block_pos` passes the index of the block you wish to map and returns the index of that block with respect to the underlying block device. It is important to understand exactly what these arguments expect and what they return:

blocksize

`stat()` is is happy to provide callers with a blocksize value. This blocksize is often not the right one for use with bmap. The `stat()` man page indicates that `stat.st_blksize` is for efficient filesystem I/O. The blocksize suited for use with bmap is available via ioctl: `ioctl(fd,FIGETBSZ,&block_size)` when performed against a file descriptor returns the file block size in bytes.

index of the block you wish to map

index is computed in units of blocksize per the above discussion. index is zero-based.

offset of that block with respect to the underlying block device

index is computed in units of blocksize per the above discussion. index is zero-based. **NOTE:** This offset is against the start of the block device on which the filesystem is mounted. This is usually a partition – not the physical device on which the partition sits. Files with holes usually return 0 as their block offset for blocks that exist in the hole.

2.4.3 Device Determination;

`bmap` and `slacker` contain code that allows them to do I/O against the raw block device. Under linux, it takes a bit of work just to determine **where** a file is located. `stat()` returns the major/minor of the block device via `stat.st_dev` – but this is difficult information to use.

Three ways leap immediately to mind:

`mknod`

A new device node could be created somewhere with the major/minor numbers supplied by `stat()`. A serious downside is that a writeable volume must exist on the system in order for the device nodes to be created.

walk /dev

This method can be done with an existing filesystem, but the cost can be high. A /dev tree may feature thousands of entries on a modern system and the target entry may be buried hundreds or thousands of entries deep. This penalty could be extreme if the /dev tree were located on a remote system – although this situation should be extremely rare.

maintain an internal mapping

This method is an attempt to speed up lookups in /dev by build-time precomputing a table with major/minor and node names for many block devices. The target device is checked to determine that the major/minor numbers are actually correct as a check.

`bmap` and freinds maintain an internal mapping for fast lookups. This saves measureable time when bmap is invoked as the object of a file-system walk over tens of thousands of files. Currently, however, they do not search or store this mapping very efficiently.

2.5 Advanced Block Map Techniques

2.5.1 Undeleting files (brute force)

1. Determine byte offset of string with respect to beginning of block device containing filesystem
2. Compute sector(s) containing string
3. Generate inode sector lists exhaustively over the filesystem

   ```
   find * -exec bmap {} >> /another_file_system/blocks \;
   ```

4. Sort lists from step (3) into a single list

   ```
   cat /another_file_system/blocks | sort -n | uniq > > /another_file_system/blocks.sorted
   ```

5. Identify the contiguous set of unallocated sectors surrounding the sectors from step (4)
6. Extract the sector set identified in step (5)
7. Done

2.5.2 Undeleting files (openinode)

Scyld's `openinode` kernel patch relieves most of the complexity of 'undeleting' files. However, a simple postprocessing step is often useful when attempting to validate recovered files – a check should be made to determine if file blocks from the recovered file have been subsequently allocated to other files.

1. Generate inode sector list for the recovered file
2. Generate inode sector lists exhaustively over the filesystem

   ```
   find * -exec bmap {} >> /another_file_system/blocks \;
   ```

3. Sort lists from step (3) into a single list

   ```
   cat /another_file_system/blocks | sort -n | uniq > > /another_file_system/blocks.sorted
   ```

4. See if any of the sectors reported for the recovered file
5. Done

Unfortunately, lack of collisions is not enough to guarantee that a recovery is correct. Consider:

1. User `tom` creates a file F(tom) containg the details of his baseball card collection. This results in the creation of an inode I(tom) mapped into the inode space of the filesystem and a vector of blocks V(tom) containing file data or metadata.
2. User `tom` deletes F(tom). Presuming that no other links to I(tom) exist, the filesystem is now free to reclaim (seperately) both the inode entry I(tom) and the blocks listed in V(tom).
3. User `dick` creates a file F(dick) containing a great new picture of two midgets and a horse from alt.rec.stepladders.and.livestock. This results in the creation of an inode I(dick) mapped into the inode space of the filesystem and a vector of blocks V(dick) containing file data or metadata. Let us stipulate, for the example, that V(dick) exactly equals V(tom) – which is to say that the picture of midgets now occupies the blocks previously dedicated to the baseball cards.
4. At this point, V(dick) may contain blocks reclaimed from V(tom). This does not imply that I(dick) is mapped into the filesystem on the same inode number as I(tom). We can detect this block reuse when recovering F(tom) by exhaustively comparing the elements of V(tom) against the elements of every other V() associated with every other I() in the filesystem – we would learn that V(dick) contains blocks reclaimed from V(tom). Obviously, we must regard at least portions of F(tom) as unrecoverable if its blocks have been recycled!
5. User `dick` deletes F(dick). Presuming that no other links to I(dick) exist, the filesystem is now free to reclaim (seperately) both the inode entry I(dick) and the blocks listed in V(dick).
6. At this point, a simple validation pass (as per above) would fail to reveal that V(tom) was reused as V(dick) because F(dick) has been removed. If we had failed to consider this point (as analysts surely have) we might have already fired `tom` from his job J(tom) for the midget picture! Perhaps we could increase the sophistication of the validation pass to survey every V() associated with every inode in the inode space – we could maybe see that a file,F(dick), was created after F(tom) and contained blocks reclaimed from V(tom).

7. The waters muddy further when user `harry` creates a file F(harry) containg his Christmas shopping list. This results in the creation of an inode I(harry) mapped into the inode space of the filesystem and a vector of blocks V(harry) containing file data or metadata. Let us stipulate, for the example, that I(harry) is mapped onto the same inode number that I(dick) was mapped onto.

8. At this point, we are still tempted to believe that our recovered F(tom) contains a picture of midgets ; further that `tom` was deliberately hiding his pictures under a fake name. Unlike previous steps where a mechanism existed for determining that elements of V(tom) had been reallocated, every record of F(dick) – namely I(dick) and V(dick) – has been obliterated.

While that situation sounds dire, there may still be hope for `tom` before he's (wrongly) sent off to jail for child pornography. Modern journalling filesystems may contain extra information that allows us to exactly determine whether tom's original file is recoverable.

2.6 Library Interface

```
#include <bmap.h>

extern int bmap_get_slack_block(
                int fd,
                long *slack_block,
                long *slack_bytes,
                long *block_size);
extern int bmap_get_block_size(int fd);
extern int bmap_get_block_count(
                int fd,
                const struct stat *statval);
extern int bmap_map_block(int fd,unsigned long block);
extern int bmap_raw_open(
                const char *filename,
                mode_t mode);
extern void bmap_raw_close(int fd);
```

3 Credits

I would like to thank the *NASA Office of Inspector General* for having the special needs that caused me to write this utility in the first place.

I would like to thank Bob Hergert of the *Defense Computer Forensics Lab* for developing the `xscale` companion utility and for testing this product.

I would like to thank the FBI SWG-DE (Scientific Working Group on Digital Evidence) for working to establish and promulgate guidelines that make it feasable to apply high-performance computing techniques to the computer forensics process.