DISK ENCRYPTION ON JOS

-Pallavi Mutyala (108601884)

-Vinay Krishnamurthy (108800980)

Abstract: *In this Project, we have implemented a miniature or prototype disk encryption for JOS. Entire disk is encrypted except disk 0 (which has the kernel image). We have used the symmetric key algorithm for encryption. The volume encryption key is stored on the last disk. Once the password is given, that password is used for decrypting the key that has been used for volume encryption. The encryption and decryption are implemented in the same environment as FS environment. The encryption and decryption works correctly which has been tested using the testfile.c and writemotd.c that has been created for Lab 5 which sufficiently checks our code.*

Introduction

Disk Encryption protects information by converting it into unreadable code that cannot be read easily by unauthorized people. Disk encryption uses a key for encrypting the whole data on disk. So an efficient encryption standard is used in this project to encrypt the disks. Most of commercially available disk encryption software either use TPM as during the first stage of the validation or use a password. In this project, we decided to use the password (reasons for not using TPM are discussed below). This project was partially inspired by BitLocker^{TM} software, disk encryption software by Microsoft.

Design

There were many encryption algorithms to choose from. The asymmetric key cryptographic algorithms were out of the question, because their performance is very bad, when there is a large amount of data to encrypt or decrypt. Of the available symmetric key cryptographic algorithms, we chose *Advanced Encryption Standard* (AES), algorithm because of it is very fast, and provides a good amount of security too. We actually went with the AES-ECB mode with a 256 bit key, in favor of CBC mode, because CBC mode would require chained encryption or decryption, resulting in degradation of performance.

The disk encryption software for JOS encrypts all the blocks are encrypted except block 0. The reason we chose to leave out block 0 is adding the encryption and decryption routine to the first stage of bootloader would increase its size to more than 512 bytes, and it would not fit within the first sector, thereby complicating the boot process. The first stage of the bootloader cannot be encrypted obviously, because there has to be some code to bootstrap the decryption of the volume (most of the commercial software leave out one part of the bootloader too).

We are encrypting entire disk by encrypting each used block on the disk, based on the data in bitmap. The free blocks are left unencrypted, during this stage of the encryption. However, if the block is allocated, the data is encrypted before putting it on the disk. The last block is reserved for book-keeping to store the AES context (Encryption and Decryption Round keys), and the 256-bit key. The last block is in-turn encrypted, using a password taken as input from the user. This password is never stored anywhere, it is discarded as soon as its job is done. In order to decrypt the last block, user is again asked for the password. A maximum of tries is provided to the user, and if the user gets the password wrong all the 3 times, JOS will continue to boot up and fail.

We have provided hooks to call the encryption and decryption routines in the FS environment, rather than having it in a separate user environment. Ideally, it should have been in the same environment or should be interacting with Block Device layer, but since Block Device Layer is implemented with the FS environment, we chose to have our functions defined in and called from other functions in the FS environment. However, unlike file-system encryption, our code still encrypts all the data contained in a block, including file metadata.

We had plans to use the TPM to store the keys to decrypt volume encryption keys, and for hardware measurement to ensure that the software above it is not tampered with. We had researched Software based TPM emulator (Berlios software TPM emulator). However, we ran into issues with it, as it required implementation of TPM driver stack on JOS, which is not an easy task to do, with the time and resources given to implement the project.

There will be a lot of IO when the disk is being encrypted for the first time. This is because, we decided to implement to this in a greedy fashion. Considering the fact that there won't be much of an impact on the performance as such, we did not implement buffered lazy writes to the disk. This ensures that the latest data is there on the disk. However, it is not intelligent enough to pick up from where it left off, if at all if it were interrupted.

Analysis

The disk encryption routines use three extra pages, mapped below UTEMP. Two of the pages are used as scratch space, while one page has the last block mapped onto it. The overhead introduced by the decryption and encryption while writing the data, is very negligible. This is evident while running *testfile.c*, the *largefile* test takes about the same time as if it were executed on a JOS environment with no encryption or decryption support. While encrypting the disk for the first time takes a bit of time, as it has to encrypt almost 8867 blocks .

Testcase

The test case we used is the *testfile*.c and *writemotd* that was there for Lab5. It sufficiently tests the encryption and decryption part of our code. When we run that program for the first time, we see that the disk is being encrypted and when it is run for the second time, the OS boots as it were running on an unencrypted disk. Since, we cannot see the actual contents of the disk anywhere; we have provided an option to dump out the first 16 bytes of the data on the disk, and what is mapped. This option can be activated by uncommenting *#define TEST* line in *disk_crypt.c* file. That should dump the contents of the first 16 bytes of block 8867 (by default) on disk, and its contents mapped in memory.

The working of our code can be tested by running:

make clean (if running for the first time)
make
make run-testfile.

Conclusion

To conclude, although this is not a full-blown implementation of a disk encryption software, it does its job as a prototype. There are certain shortcomings in the implementation, but some of them were rooted in JOS, and there's definitely room for improvement for this project.

References

- http://en.wikipedia.org/wiki/Disk_encryption
- http://en.wikipedia.org/wiki/Advanced_Encryption_Standard
- http://lists.gnu.org/archive/html/qemu-devel/2007-10/msg00750.html
- http://wiki.qemu.org/Features/TPM