Writeup: Assignment 4

Authors: Robert Sato, Keerthi Krishnan, and Pranav Nampoothiri
Cruz ID: rssato, kvkrishn, pnampoot

Due Date: June 2nd, 2019

Assignment Description:

In our implementation of encryption in the FreeBSD file system. We created a system call that sets an encryption and decryption key for a specific user ID. To create our file system layer, that enables encryption and decryption with the read and write system call, we used the file system in user space, FUSE, to apply the key to a file, so long as the key is set for the user and the sticky bit is turned on.

System Call:

To create our system call, we first decided what data structure we would use for handling the keys and users. Since we needed to handle upto 16 users, we settled on using a static data structure, in this case,an array of structs that holds a user ID and a key that we pass into our setkey system call. We created another struct, called setkey_args, which contains fields k0 and k1 (these occupy the rest of the key that aren't the most significant integers of the key), a userid variable, and a userkey variable. To be able to access all the fields in the setkey_args struct, we allocate memory using the FreeBSD version of memory allocation,with malloc.h, after which we use print statements to print out the unsigned integers and the UserID. We create a user_count variable to keep track of the number of users that we have assigned keys to.

Following this, we have a conditional that checks if our unsigned integers, k0 and k1, of the key are 0; if they are, we simply have a print statement stating that we are unable to set the key and return; this is because this occupies the rest of the key that is not 0 by default, so if this is 0 as well, we don't have a valid key that we can set a user, a restriction as per the assignment specifications. Now we use the user key field from the struct setkey_args, to hold the key from the unsigned integers k0 and k1; we shift left k0, in order to concatenate to the right of this part of the key, with key k1, and get the complete user key, including the significant integers set with value 0.

To do the actual setting of the key to a particular userID, we declare our array of structs with 'struct staticTable' type, to have every element refer to 'struct staticTable'. We index every element of 'staticTable' array of structs with the userCount to refer to a particular user struct. Using our userid field from every array struct element, we assign to this the userid of our 'struct setkey_args'. We do the same thing with the user key, where we assign to the 'key' field from our 'struct staticTable' the 'user_key' field from our struct set_key args. All of this is done under the conditional, stating that our user_count is less that our SIZE, which we define as 16, for accounting for up to 16 users. We then increment our user_count in this same conditional, for every time this conditional is accessed in the code. Otherwise, if the user_count is greater than

our size of 16 users, we simply print a statement, giving out an error message saying that there are more than 16 users. We then free our memory that we allocated for our setkey_args struct, to avoid memory leaks.

FUSE:

When creating our stackable filesystem layer using FUSE, we first installed FUSE on a FreeBSD machine. We mounted the file system to FreeBSD, by running FUSE example files on our FUSE-installed FreeBSD. We determined that to enable encryption and decryption with read and write system calls, we needed to modify these system calls, where writing encrypts a file, and reading decrypts a file. We located the fusexmp.c source file that contained the functions for read and write; these functions, called xmp_read and xmp_write, are in fact assigned as calls
to the read and write definitions, and defined as regular functions in the fusexmp.c source code; these read and write system definitions are treated as fuse operations in a static struct filled with different operations for fuse. We refer/call to these fuse operation definitions of read and write, by using the read and write system calls, to open and read the fusexmp.c file and jump to its instances of read and write. This allows us to add encryption and decryption code for the xmp_write and xmp_read functions for the fusexmp.c source file, when compiling our separate test file.

In xmp_write, we write our encryption code. We allocate memory to a struct stat pointer called buffer, and pass it to the stat system call along with a file path to get file data. To get the sticky bit, we assign the buffer struct field, 'st_mode' to another variable of type mode_t, called 'mode', to get the sticky bit. In an "if-else' conditional structure, we first check to see if the sticky bit represented by 'mode' is equal to 0 (in other words, the sticky bit is not set); if it is equal to 0, we simply do nothing. Otherwise we write our encryption code in the 'else' statement. Here we declare 4 uint64_t bit arrays for the ctr_inode, value, key, and data, for which we allocate memory for the ctr_inode, value, and key uint64_t bit arrays. We get the file ID number and put it in the first element of the ctr_inode, ctr_inode[0]. The second element of ctr_inode, ctr_inode[1], is set to the CTR value which is a byte offset of 16-byte chunks. The rest of the key is the high-order bits of the key, so we assign 0 to the first element of the 'key' uint64_t bit array; we then convert the rest of the key from a string to a long, and put it in the second element of the 'key' uint64_t bit array. We initialize a 'leftover' variable to keep track of how many bits are yet to be written; we get this from the variable 'size' that is passed in as a parameter into the xmp_write function.

Within this same else conditional, we create a loop structure in which we go through the leftover bits that we are to write, and encrypt the key using the SPECK encryption algorithm. We then take our data in 16 byte chunks and XOR with the value specified to encrypt each 16 byte chunk of data. We encrypt 8 bytes at a time, encrypting the first 8 bytes and then encrypting the second 8 bytes, using data[0 +counter] with value[0] and data[1+counter] with value[1] repectively. We then decrease the leftover counter by 16 in order to make sure we are keeping track of how many bytes we have encrypted, and we increment the offset to keep track of how many bytes are left to encrypt. We then free the arrays value, ctr_inode, key. Once we

have done our encryption, we then open the file and write the encryption to the file, to complete the file encryption. The process for decryption, done in xmp_read, is the exact same, except we first open the file in order to get the encryption of the file, before we get the file data, allocate the memory for out uint64_t bit arrays, and decrypt with the same exact process as encrypting to reverse the encryption.

Protectfile:

In our protectfile, we first have the SPECK encryption algorithm provided to us by our instructor. In our main method following this, we first have a conditional printing out an error message whenever we have less than 4 arguments in our command line. Then, we initialize our arguments by assigning the second element of our arguments array to a character array, called ‘option’. If the first element of this is neither the character ‘e’ nor ‘d’ (in other words the arguments to indicate encryption or decryption, is invalid), then an error message is printed, saying the argument entered in invalid. Below this, we make some declarations and initializations for the different arguments needed to run, assigning the 2nd and 3rd elements of the arguments array to a character file array and character input key array. Memory is allocated for the buffer in order to get file data and encryption bit, the same way it was done in FUSE. We then get the sticky bit from a ‘mode’ variable which was initialized with a buffer field for the sticky bit, and check to see if it is 0 (if the sticky bit is not set). If it is not set, then we check to see if the first argument is ‘d’ and if it is, then we know that the file has been decrypted, and a message is printed saying that we do not need to decrypt. If the ‘mode’ variable is not 0 (if the sticky bit is set), then we check to see if the first argument is ‘e’ and if it is, then we know that the file has been encrypted; again here, a message is printed saying that we do not need to encrypt, in this case. We then declare a char array that holds a string containing the permissions of the file. Using that string and the strtoll library function, we convert the string to a long, and then we call chmod to set the privileges of the field specified. When handling the file, if there is an error with chmod, we have a print statement stating that we have encountered an error and the file cannot be opened. We also turn the sticky bit off in this portion of the code, because we do not want to have the encryption and decryption done by the read and write system calls when manually encrypting and decrypting here with the protectfile.

Similar to the encryption and decryption process used for the read and write system calls, we initialize 4 uint64_t bit arrays for the value, ctr_inode, key, and data. We retrieve the file ID number from the first element of the ctr_inode array, and assign the byte offset to the second element of the ctr_inode. We set the first element of the key array to the higher order bits which are 0, and the second element of the key array to the lower order bits, by using strtoll, a library function that converts strings to longs.

We then move on to file handling, where we create a file descriptor variable fd and assigned to it the open syscall in order to open the file in read and write mode. If the file descriptor returns a -1, then we realize that the file cannot be open and print error messages accordingly.

We then move onto the implementation of the encryption and decryption, where we have a loop that reads in data in 8 byte increments. We first read in 8 bytes, and if allowed, we read

in another 8 bytes. Otherwise, we get a new value for each 16 byte offset. We then make a call to Speck Encryption Algorithm to encrypt the key followed by some debug code. We then move on to the encryption of the file, XOR'ing the data with the value, using the data bit arrays and value bit arrays. The first 2 elements of the data array represent the entire 16 byte block and we encrypt in 16 byte block increments. We read the number of bytes up to 16 bytes and use lseek to go back to the beginning of our 16 byte chunk in order to write the encrypted content. This is followed by debugging code. We then check if the count is less than 9, in other words, we are still in the first half of the 16 byte chunk, then we write to the first half of the 16 bytes. Otherwise, we write to both halves of the data using the write system call by referring to the first 2 elements of the data bit array. We then close the file descriptor, turn the sticky bit back on if it is encrypted, and vice versa if it is decrypted. We then free all arrays.

What We Could Improve On:

Throughout this assignment, our biggest issue was understanding how the different parts of the assignment worked together to implement the cryptoFS. One thing we could have worked on with this issue is spending more time understanding the assignment document as well as how the different specifications of the assignment work together. We also had trouble implementing the syscall, so one thing we could work on next time is understanding exactly what the syscall was asking as well as reading up more documentation on how to write a syscall function and incorporate it within the kernel. This was a challenging assignment, and it taught us to be more proactive in fully understanding every detail of the assignment and to not be afraid to experiment with the kernel.