Lab 2 Design Problem

Encrypted Ramdisk

Keaton Boyle — 103-882-791
Anthony Ortega — 503-953-545

February 21, 2013

Introduction

This project provides functionality for users and applications to encrypt and decrypt a ramdisk using a key, and to read from and write to the resulting ramdisk when the correct key is provided. Certain functions are made available to userspace applications via the `eosprd.h` header file. Since we're using the encryption algorithms provided by the kernel, we intend to support all of the block cipher encryptions that Linux already handles.

Interface

The goal of our interface is to provide a few simple functions for the user to allow them to establish a connection with an encrypted ramdisk and then conduct any further communication with the encrypted ramdisk using basic `read()` and `write()` system calls. These functions are provided in the `eosprd.h` header file and are summarized here.

`eosprd_open()`

`eosprd_open()` is a wrapper function that should be used in place of a normal `open()` system call whenever opening an encrypted ramdisk. It simply calls a system `open()` on the ramdisk and then provides the password via a newly-defined `ioctl()` call, `EOSPRDIOCOPEN`. The following is the basic structure of the `eosprd_open()` call, without any of the error checking.

```
static int eosprd_open(const char *pathname, int flags, char *
    key)
{
        int fd;
        fd = open(pathname, flags);
        ... on failure, return -1 ...

        ioctl(fd, EOSPRDIOCOPEN, key);
        ... on failure, close fd and return -1 ...

        return fd;
}
```

If anything fails, `-1` is returned. The call to `open()` can fail for any of the normal reasons, the `ioctl()` call will return `-1` if the provided key does not match the key that the ramdisk was encrypted with (`errno` will be set to `EKEYREJECTED`). Additionally, if a ramdisk is not encrypted, the `key` argument should be `NULL`, or else the `ioctl()` call will return `-1` and set `errno` to `ENOSYS`. (In such a use case, a simple `open()` could be used in place of `eosprd_open()`.) If the `ioctl()` does fail for any reason, the newly-opened file descriptor is closed before `eoprd_open()` returns.

If all is successful, the newly opened file descriptor will be returned and the calling process will be able to `read()` and `write()` to it using normal system calls (according to the specifed `flags`). These calls will transparently encrypt and decrypt data coming to and from the ramdisk without the calling process ever needing to provide the key again, until the process exits or the file descriptor is closed.

`eopsrd_encrypt()`

`eosprd_encrypt()` allows the user to encrypt, decrypt, or change the encryption key or scheme of a ramdisk. It takes four arguments: the path to the ramdisk file, the key the ramdisk was previously encrypted with (or `NULL` if the ramdisk was previously unencrypted), the new key to encrypt the ramdisk with (or `NULL` if the ramdisk should be unencrypted), and the name of the algorithm to be used to perform the encryption (one of the supported blocking encryption algorithms that Linux provides). The function is primarily a wrapper for a new `iotcl()` call, `EOSPRDIOCENCRYPT`, which is called with the function's argument packed into a `struct encrypt_args` structure. The following is the basic structure of the `eosprd_encrypt()` call, without some of the specifics.

```
static int eopsrd_encrypt(const char *pathname, char *oldkey,
    char *newkey, char *algo)
{
        int fd;
        struct encrypt_args e_args =
        {
                .oldkey = oldkey,
                .newkey = newkey,
                .algo = algo
        };

        fd = open(pathname, O_RDWR);
        ... on failure, return -1 ...

        ioctl(fd, EOSPRDIOCENCRYPT, &e_args);
        ... on failure, return -1 ...

        close(fd);
        ... on close failure, exit fatally- this leaves the
            ramdisk in a strange state ...

        return 0;
}
```

If the encryption or decryption is successful, `0` is returned. In case of failure, `-1` is returned. Failure can occur for any of the typical reasons associated with `open()`, or for problems with the `ioctl()` call as discussed in `eosprd_open()`, or if the `algo` argument isn't recognized (`errno` will be set to `ENOSYS`), or if another process has the the file open for encryption, reading, or writing (`errno` will be set to `EBUSY`).

This function, if successful, decrypts (if necessary) the ramdisk using the key `oldkey` and whatever scheme it was previously encrypted with and then re-encrypts it (if `newkey` is non-`NULL`) with the key `newkey` using the encryption algorithm specified by `algo`, or the previously-used algorithm if `algo` is `NULL`. If `algo` is `NULL`, `newkey` is non-`NULL`, and the ramdisk is coming from a non-encrypted state, the `ioctl` call will fail with `errno` set to `ENOSYS`.

In order to avoid race conditions and the possibility of one process blocking another's read or write by changing the encryption key, etc., this function will only succeed when no other process is reading, writing, or trying to encrypt/decrypt the ramdisk.

Notes:
If the user wishes to encrypt a file for the first time, `oldkey` should be `NULL`.
If the user wishes to decrypt a file, `newkey` should be `NULL`, and `algo` will be ignored. (This is also available via the `eosprd_decrypt()` function.)
If the user wishes to change the key used to encrypt a file without changing the encryption algorithm, `algo` may be left `NULL`.
If the user wishes to switch encryption schemes without changing the keys, `oldkey` and `newkey` must be the same, and `algo` should be the new encryption scheme.

`eopsrd_decrypt()`

`eosprd_encrypt()` is a simple inline function that simply calls `eosprd_encrypt()` with the correct arguments for decryption. The entire function is shown here.

```
static inline int eosprd_decrypt(const char *pathname, char *
    oldkey)
{
  return eosprd_encrypt(pathname, oldkey, NULL, NULL);
}
```

Sample Code

The following is a simple sample application that encrypts a ramdisk, opens it, reads and writes to it, and closes it. Note that simple `read()`, `write()`, and `close()` calls are all that is necessary if the proper `eosprd_...()` calls have been used ahead of time.

```
#include <unistd.h>
#include "eosprd.h"

int main(int nargs, char **argv)
{
        char buf[128];
        int fd;

        // encrypt the /dev/osprda ramdisk with key "password"
        // and the "blowfish" encryption algorithm
        eosprd_encrypt("/dev/osprda", NULL, "password", "
            blowfish");

        // open the file using the encryption key
        fd = eosprd_open("/dev/osprda", O_RDWR, "password");

        write(fd, "foo", 3);
        // foo will be encrypted and placed in /dev/osprda

        read(fd, buf, 3);  // buf will now contain foo

        close(fd);
        // other processes will not be able to read "foo"
```

```
        // without knowing the key

        return 0;
}
```

Implementation

The internal readings and writings to the ramdisk are done using largely the same code as in the bulk of Lab 2. The difference is that when encryption is enabled, we'll call kernel encryption and decryption functions before actually sending data to or from the ramdisk. Additionally, we'll need to keep track of if a ramdisk is encrypted, we'll need to know what encryption algorithm was used, we'll need some way to check if the user has provided the right key, and we'll need the file operations to somehow "remember" the key that the opening process provided, so that the user can use simple `read()` and `write()` calls without having to provide the key again. Furthermore, we need to be careful with concurrency issues, so that no process can reencrypt a file while another is trying to read it, or something of that sort. And of course, we'll actually need to do the encryption.

Keeping track of encryption state

Solving this challenge is simple enough. We've added a flag, `int encrypted`, to the `osprd_info_t` structure. It is initially set to `0` when the ramdisk is first created. Later calls to `eosprd_encrypt()` with non-`NULL` `newkey` values will set this flag to `1` so that all further calls to `eosprd_encrypt()` and `eosprd_open()` check for the correct key. Note that basic `open()` and `read()` calls (without first calling `eosprd_open()`) will still succeed, but they will be reading the encrypted garbage data. A basic `write()` call (without first calling `eosprd_open()`) will fail to write and return `0`.

Calls to `eosprd_encrypt()` with a `NULL` `newkey` (i.e., calls to `eosprd_decrypt()`), if they provide the correct `oldkey`, will set this flag back to `0`.

Keeping track of encryption algorithm

This too is fairly simple. Calls to `eosprd_encrypt()` will use the newly-added `char *algo` field of the `osprd_info_t` structure to retrieve and store the name of the algortihm used for encryption. This is initialized to `NULL` at the creation of the ramdisk.

Checking key correctness

We want some way to tell the user whether or not the correct key has been provided, but storing it somewhere in plaintext and checking against that seems to defeat some of the purpose of encryption. Instead, when the key is first provided by the `eopsrd_encrypt()` function, we'll encrypt it with itself and store the result in the newly-added `char *key` field of `osprd_info_t`. Then, when a user attempts to open the encrypted ramdisk, we'll decrypt `key` with the provided password, and if they match we'll be in the clear. Otherwise, the `eosprd_open()` attempt will fail. This does leave access to the key user-side, but it avoids storing it in some fixed location.

Remembering the key that a process has provided

Again, we'd rather not have a plaintext list of processes with priviledged access sitting around near our ramdisk, nor would we want to store the keys they've provided alongside them. But we need some way of remembering which open file descriptors have provided keys so that `read()`s and `write()`s on those file descriptors can use those keys. To do this, we'll store a pointer to the key in the `f_security` field of the `struct file` attached to the open file descriptor. This way, every time `read()` or `write()` is called, we can refer to the `f_security` field to do encryption and decryption. This information will disappear with the closing of the file descriptor.

Concurrency issues

Encryption or decryption should only occur when there are no other file descriptors open on a ramdisk. To guarantee this, encryption operations will always try to grab write locks on the ramdisk and will fail immediately if they are unable to. This isn't perfect, perhaps blocking would be better, but generally the intention is that initial encryption happens very rarely and should never be in parallel. Subsequent `write()` call encryption will occur transparently and reliably using already-implemented locking mechanisms.

Encryption

Encryption will follow the basic encryption schemes provided by the kernel's crypto functions and will decrypt and encrypt one sector at a time.

Results

We'll have more information on our results once this has all been put to code, but generally we expect this to be a decently-working solution. It's unfortunate that keys, at some point or another, will be floating around in plaintext, but that's tough to avoid. The interactions and subtleties between user and kernel code is always a challenge, but we're working through it and trying to rely as much as possible on the pre-existing Lab 2 interfaces.

Work Division

Again, we'll have more information when we're finished, but design was worked out together by Mr. Boyle and Mr. Ortega. Most of the design document was written by Mr. Boyle, Mr. Ortega integrated the password interface into the `osprdaccess` program, and the rest of the coding is to be determined.