BentoCrypt

A Stacked Encrypted File System Implemented in Safe Rust

Prathik Gowda
CS Department
Grinnell College
Bloomington, Illinois, USA
gowdapra@grinnell.edu

***Abstract*—Memory safety and data encryption are crucial to the security and efficacy of modern file systems. However, most contemporary solutions are either implemented in C, which sacrifices memory safety, or lack file-based data encryption. BentoCrypt is an attempt to bridge this gap by providing a stacked, encrypted file-system implemented in the memory-safe Rust language - one of the first of its kind. It does this by leveraging the Bento [1] project: a new file systems development framework which empowers the user to write their kernel file system in entirely safe Rust.**

I. Introduction

Two of the biggest threats to the security of file systems are memory safety vulnerabilites and unencrypted data. For evidence of the catastrophic consequences that memory vulnerabilities can have, look to the OpenSSL library's Heartbleed bug. The Heartbleed bug allowed attackers to read private data past the scope of a valid request. This was due to a lack of proper bounds checking - an issue that would have been avoided if OpenSSL had been implemented in a memory-safe language. BentoCrypt avoids these memory vulnerabilities by implementing the file system in safe Rust using the Bento file systems development framework. Bento provides a FUSE-like (Filesystem in Userspace) API while also allowing your file system to be run at the kernel level.

This leaves the issue of data encryption. Most file systems leave your data unencrypted on the disk. This means that if a hacker gains access to your hard-drive, there is nothing stopping them from reading your data. BentoCrypt addresses this by providing file-based data encryption - meaning that your sensitive data lies encrypted on the disk. We expect to provide the aforementioned memory safety and file-based data encryption while also being faster than FUSE-based equivalents like gocryptfs [2] due to speed benefits intrinsic to the Bento framework.

II. Related Work

We can categorize the related work into two categories: traditional file systems and cryptographic file systems. The traditional file systems include Ext4, BtrFS, and OpenZFS. All of them are implemented in C (and thus are prone to memory exploits). Ext4 and OpenZFS now provide encryption, but BtrFS still does not.

National Science Foundation

In the category of cryptographic file systems, we have implementations such as gocryptfs [2], eCryptfs [3], and EncFS [4]. However, gocryptfs is the only one which provides a memory-safe implementation (in the Go language) and file-based data encryption. What differentiates gocryptfs from BentoCrypt is the level at which it runs. gocryptfs is implemented as a FUSE module which limits it to running in user space (decreasing performance by up to 83 percent [5]). Our implementation leverages the Bento development framework to run at the kernel level with access to kernel primitives which significantly increase performance.

III. Approach

BentoCrypt's goal is to provide file-based encryption using Bento's safe framework. To achieve file-based encryption, we need a master key to encrypt and decrypt file data. For AES256-GCM encryption (our authenticated encryption algorithm of choice), our key must be 32-bytes long - which is prohibitively long for a user to remember naturally. Thus we should store our master key safely so that it can not be retrieved by others. Our solution is to encrypt our master key using AES256-GCM encryption with a key derived from the login user's encrypted password (retrieved from /etc/shadow). Since our master key will be encrypted, it can then be safely stored anywhere.

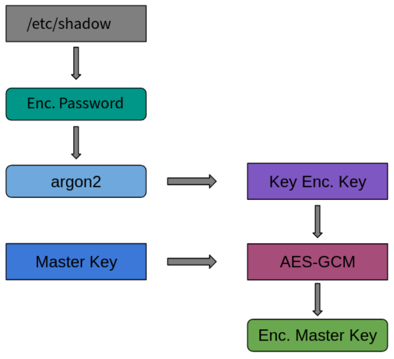

Fig. 1. Password Helper Diagram

Since we need 'sudo' privileges to access /etc/shadow, we split this functionality into a separate password helper program (see Fig. 1. above) to minimize the amount of code run with

’sudo’ permissions. Our password helper program retrieves the current user’s encrypted password from /etc/shadow and produces a key encryption key (a key to encrypt our master key) from it by using the key derivation algorithm argon2 [6]. Next, we derive our master key using bytes from /dev/urandom (this is as far as we have gotten so far) and encrypt it using our key encryption key. Our solution prevents other users from accessing your data, since the key encryption key is seeded from the encrypted password which is unique to the appropriate user.

That leaves the most important component of BentoCrypt: the file system itself. The Bento framework works by registering itself as a valid Linux VFS (Virtual File System) called BentoFS. However, instead of actually executing the calls and requests it receives, it translates them from unsafe C data structures to safe Rust data structures using the Bento library. Finally, it passes these safe requests to our file system BentoCrypt; which can process and execute the calls in entirely safe Rust (see Fig. 2. below).

Our design is a ”stacked” encrypted file system - meaning that we are stacking a layer on top of an existing file system to provide encryption. The question is where we want to insert our encryption layer. At the core we have the kernel itself: adding encryption to read and write routines at this level would allow any implementing file system to provide file-based encryption. However, adding encryption at the kernel level also prevents us from using the Bento framework with safe Rust code; thus making us prone to the memory vulnerabilties we have already enumerated. Inserting our encryption layer at the BentoFS level produces similar issues. Modifying BentoFS routines exposes us to unsafe C data structures - since BentoFS’s job is to translate these structures from C to Rust.

Thus, we believe our encryption layer is best placed above the xv6fs2 file system [7] provided with Bento. This level saves us from dealing with unsafe C requests (they have been translated by BentoFS by this point). Our layer adds encryption on write operations and decryption on read operations. This ensures that our file data is encrypted while at-rest on disk (due to our encryption-on-write policy), but still can be decrypted and read by the appropriate user.

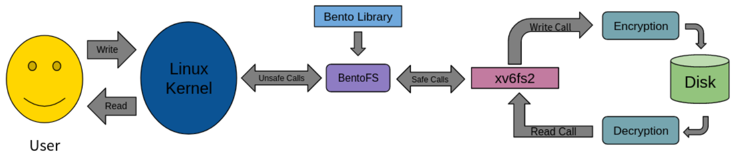

Fig. 2. BentoCrypt Diagram

IV. Methods for Evaluating BentoCrypt

We plan on evaluating the performance of BentoCrypt by running the IOzone filesystem benchmark tool [8] and comparing our results to those of the traditional and cryptographic file systems mentioned earlier. Broadly, we expect BentoCrypt to provide better performance than FUSE-based cryptographic equivalents like gocryptfs (because BentoCrypt runs at the kernel level) and comparable performance to traditional file systems with encryption enabled.

Evaluating the security of BentoCrypt is more tricky. The Bento framework ensures that there is no unsafe code in our implementation. The cryptographic algorithms we have chosen (AES256-GCM and argon2) are well received in the cryptographic community: AES-GCM is used for end-to-end encryption in Firefox and argon2 was the winner of the Password Hashing Competition in 2015. However, logical errors and vulnerabilities in our code unrelated to memory safety or algorithms are possible. To further ensure the security of BentoCrypt, a security audit by a third party could be helpful.

V. Conclusion

By utilizing the Bento framework we plan to create one of the world’s first kernel-level encrypted file systems implemented in a memory-safe language. Our implementation avoids the dangerous pitfalls omnipresent to traditional file system developers by opting for safe Rust instead of C - and ensures the security of your sensitive data via file-based data encryption.

Acknowledgment

I would like to thank my mentor Professor Chandy for his incredible guidance and help through this REU. I would also like to thank Samantha Miller for heading the Bento project and for helping me adapt the Bento framework to work with newer versions of the Linux kernel and Rust. I would also like to thank Professor Khan for organizing this REU and introducing us to technical writing - this paper would not have been possible without him. Lastly, I must thank the National Science Foundation for generously supporting this research.

References

[1] S. Miller, K. Zhang, M. Chen, R. Jennings, A. Chen, D. Zhuo, and T. Anderson, “High velocity kernel file systems with Bento,” in USENIX, 19th USENIX Conference on File and Storage Technologies, February 2021.
[2] rfjakob, ”Home Page - gocryptfs,” May, 2020. [Online]. Available: https://nuetzlich.net/gocryptfs/. [Accessed July 1, 2021].
[3] D. Kirkland, ”Home Page - eCryptfs,” 2020. [Online]. Available: https://www.ecryptfs.org/. [Accessed July 1, 2021].
[4] V. Gough, ”About Page - EncFS,” May, 2020. [Online]. Available: https://vgough.github.io/encfs/. [Accessed July 1, 2021].
[5] B. Vangoor, V. Tarasov, and E. Zadok, “To FUSE or not to FUSE: performance of user-space file systems,” in USENIX, 15th USENIX Conference on File and Storage Technologies, February 2017.
[6] A. Biryukov, D. Dinu, and D. Khovratovich, “Argon2: the memory-hard function for password hashing and other applications,” University of Luxembourg, Luxembourg, December 2015.
[7] S. Miller, ”Root of xv6fs2 implementation,” June, 2021. [Online]. Available: https://gitlab.cs.washington.edu/sm237/bento/-/tree/master/xv6fs2. [Accessed July 1, 2021].
[8] D. Capps, ”Home Page - IOzone,” Jan., 2016. [Online]. Available: http://iozone.org/. [Accessed July 1, 2021].