Abstract

As a final project I chose to update and rewrite an existing cryptographic application called Project MARUTUKKU, more commonly called “Rubberhose”. Rubberhose is a filesystem intended to provide deniable encryption, and was created by WikiLeaks founder Julian Assange for use by journalists. However, the project has been abandoned for longer than a decade. In order to update it to modern standards I needed to implement an AES CBC cipher which could be used for its block encryption.

Introduction

The primary purpose of rubberhose is to achieve cryptographic deniability. The filesystem accomplishes this by filling a disk with random data-- creating what is termed an “extent”-- and then encrypting any desired plaintext files and across the disk randomly. This creates what are called “aspects”, and multiple aspects can be stored on a single extent, each with its own separate password.

This leads to a situation where even if the password of one of the aspects is compromised and an attacker successfully decrypts and pieces together the sections of the extent where it was stored, there is no way to determine whether the remainder of the extent may contain other encrypted material from a different aspect, or if it is simply random junk.

Related Work

Sekura

Sekura is an encryption tool written in Go which is heavily inspired by the Rubberhose filesystem. According to the read-me, it "allows for multiple, independent file systems on a single disk whose existence can only be verified if you posses the correct password."

StegFS

StegFS is a free steganographic filesystem for Linux based on the ext2 filesystem. The last version was released in 2001. It functions by storing hidden files in unused blocks of a partition which already contain normal file being managed by the operating system.

Vanish

Vanish is a research project intended to "give users control over the lifetime of personal data stored on the web". It functions by encrypting information entered by the user with a key which is unknown to this user. The key is then broken up and distributed using distributed hash tables across the internet. These distributed hash tables will have a set lifetime after which they will be overwritten, erasing the original key and making the encrypted data irretrievable.

Methodology

Well, the methodology was essentially a whole lot of trial and error. Rubberhose originally had a bespoke build system aptly called "confused", which I opted to abandon entirely. Instead I manually copied sections of the code into new files and compiled it using my own build scripts.

Often times one file would have a dependency on code in another file, so I frequently had to chase down specific methods in order to ensure they were available for compilation. There were also times when

code itself would not compile because it was out of date in some way, in which case I would have to take time to update it.

Once I had the bulk the code functioning I took some time to implement AES into C. This was not overly difficult because I already had an existing Java implementation to reference from my lab work, and there were actually some things which were much easier to do in C than in Java. I did of course have to implement the decryption mode of AES myself, as well as cipher-block chaining in general. Finally I integrated the AES functionality into the existing Rubberhose cipher suite. This took some tinkering, but ultimately was not very difficult to accomplish. I also made it a point to implement an alternate “free” function which would fill the contents of a pointer with random data before freeing it, to guarantee that none of the artifacts from the encryption or decryption process were left in RAM.

Experiments

There were a few parts of Rubberhose which I wanted to implement or improve upon and did not get the chance to do so. In terms of the existing functionality. Rubberhose had a pair of kernel modules which would allow a user to attach an opened aspect as a block device under “/dev/maru0”, which could then be mounted normally into the filesystem. Unfortunately these kernel modules had been written for version 2.0 of the Linux kernel, and there have been so many changes to character and block device driver APIs that I ended up sinking a lot of time into trying to update them without much success.

The other eixsting functionality were the aspects and extents themselves, which could simply be created as files on the filesystem without the kernel modules exposing them as block devices. I was very close to completing this but unfortunately I ran out of time, as the aspect creation and mapping process was very complex and as a result difficult to debug.

Discussion and Analsysis

Key Storage

When a new aspect is created, the password chosen for the aspect is not actually what is used to encrypt the aspect itself. Rather, a master key is randomly generated and used for the encryption instead. This master key, as well as a mapping of where the fragments of the aspect are stored across the extent, are then encrypted under the user's actual provided password and stored in an external file called a keymap.

Like an extent the keymap is initially generated from random data, which is then overwritten with the encrypted masterkey and mapping when a new aspect is created. This makes it impossible to determine how many aspects a keymap is being used for; however the simple existence of a keymap file is sufficient to demonstrate that a device does indeed use the rubberhose filesystem, somewhat detracting from the cryptographic deniability.

A better potential approach would be to store the encrypted masterkey and mapping information for each aspect-- henceforth the "aspect header"-- inside of the extent itself. Of course this raises the question: where should it be stored? If stored at the beginning of the extent there is essentially no difference between this method and having a separate file, as the beginning of the extent would be an incredibly obvious place for an attacker to look.

A better approach may be to use the user's password which is encrypting the aspect header to derive some sort of offset value. For instance the password could also be passed through a SHA-256 hashing algorithm, and the resultant hash value could then be modulized by the length of the extent, to

determine an offset where the aspect header should be stored. This offset will be the same every time it is calculated as long as the password and size of the extent remain fixed.

Rubberhose Analysis Attack

The so-called “rubberhose” analysis attack, in which keys are obtained from an owner of an encrypted device through compulsion of some kind, will lead to the following scenario when faced with the filesystem after which it is named:

1. The owner of the device surrenders a password to one of the plaintexts present on the device
2. The attacker decrypts the plaintext using the password, confirming there is at least some encrypted material on the device
3. The attacker concludes that there may be additional encrypted material on the device, and continues compelling the owner to surrender the remaining keys
4. Steps one through three repeat until all of the plaintext on the device has actually been decrypted
5. The attacker concludes there is more plaintext on the device and continues attempting to compel the owner to surrender more keys
6. Having no more keys to surrender, but possessing no way to prove there is not any more plaintext on the device, the owner is left at the mercy of their attacker

Though this outcome seems quite bleak-- and indeed it would be an undesirable position to wind up in, particularly depending on the exact method of compulsion being used by the attacker-- if the owner of the device is themselves aware of this outcome, they will know that their situation in step 6 is no different from their situation in step 1.

That is, knowing that their decision to divulge any passwords for their device will end them up in a position where they are at the mercy of the attacker regardless, it is in the best interest of the owner to withhold their passwords and at least maintain the secrecy and security of any sensitive data they might have stored.

Disk Wear Analysis Attacks

A disk wear analysis attack is a cryptanalytic and hardware attack which involves determining where important data might be physically stored on a disk based on how much specific sectors have been worn down. If a disk sector has been accessed or written to many times it is more likely to contain sensitive information than a section of the disk which is otherwise pristine.

In the words of the creator, "What we want to do here is to make sure the non-data carrying magnetic/other properties of the disk substrate are as close to a Jackson Pollock painting as is possible."

This is accomplished by randomly choosing several blocks to read and then rewrite identically every few seconds. Additionally any time the user actually writes to the disk there is a chance that the aspect will be re-split have its physical locations moved to different sections of the disk.

Conclusion

References / Bibliography

Assange, Julian, Ralf P. Weinmann, and Suelette Dreyfus. "Rubberhose filesystem." *Archive available at: http://web. archive. org/web/20120716034441/http://marutukku. Org* (2001).

https://github.com/Cookie04DE/Sekura

Anderson, Ross; Needham, Roger; Shamir, Adi (1998). The Steganographic File System (PDF). Lecture Notes in Computer Science. Vol. 1525. pp. 73–82. CiteSeerX 10.1.1.42.1676. doi:10.1007/3-540-49380-8_6. ISBN .

"' Vanish: Increasing Data Privacy with Self-Destructing Data" (PDF). vanish.cs.washington.edu. Retrieved 2010-12-07.