FsVerify

Xenia

Reflexion zum
AP Informatik

Inhaltsverzeichnis

1 Idee

Die Idee einer Dateisystem verifizierung ist nichts neues, oft wird sie in embedded geräten oder handys implementiert, wo die sicherheit und integrität des systems von großer bedeutung ist. Jedoch ist sie bei Desktop betriebssystemen wie Windows, MacOS oder Linux Distributionen nicht sehr herkömmlich. Dies liegt meißt daran, dass eine effektive verifizierung des Dateisystems sich darauf verlässt, dass das Dateisystem sich nicht über zeit verändert, welches man bei Desktop-Betriebssystemen nicht gewährleisten kann, da Programme oft direkt in den Root des Betriebssystems schreiben (/usr in *nix, C:
Program Files in windows).

Mittlerweile gibt es jedoch in der Linux-welt eine neue art von Distribution, die sogenannten 'Immutable' Distributionen, welche sich darin unterscheiden, dass der Root schreibgeschützt ist, oder wie bei NixOS oder Gnu GUIX garnicht erst richtig existiert. Dadurch können Programme nur direkt in den Homeordner des Nutzers installiert werden, und das eigentliche System bleibt original bestanden.

Hierdurch wird Dateisystem-verifizierung möglich, da der originale zustand sich nie ändern wird, kann ein Programm ohne probleme verizieren, dass sich nichts verändert hat.

1.1 Implementierungen

Für die Verifizierung eines Dateisystems gibt es verschiedene Methoden:

- Per-Datei verifizierung

  Bei der Per-Datei Verifizierung wird der Hash von jeder Datei die verifiziert werden soll mit einem vorbestimmten, vertrauten, Hash verglichen, falls der Hash übereinstimmt ist Datei unmodifiziert, wenn sie jedoch abweichen, ist die Datei modifiziert und kann nicht vertraut werden.

- Festplattenverifizierung

  Hier wird ein Hash von der ganzen Festplatte oder Partition mit einem vorgegebenen Wert verglichen. Im vergleich zu der Per-Datei verifizierung werden hier auch neue Dateien erkannt, welche eine Per-Datei verifizierung ignoriert hätte. Jedoch kann dies auch erheblich langsamer sein, da die ganze Partition, welche sehr groß werden kann, in einem Thread gehasht wird.

- Blockverifizierung

  Dies ist ähnlich zu der Festplattenverifizierung, jedoch werden hier nur einzelne Blöcke gehasht und verifiziert, dies ermöglicht es, die Verifizierung durch Multithreading zu beschleunigen, während man weiterhin die ganze Festplatte/Partition verifiziert.

Alle drei arten der Verifizierung haben eine Sache gemeinsam, sie brauchen eine vertraute quelle von der sie den korrekten Hash für eine Datei/Partition/Block lesen können.

1.2 Hash Quellen

Wie bereits gesagt, braucht das Verifizierungsprogramm eine vertraute Quelle für die korrekten Hashes. Hier gibt es auch verschiedene Lösungsansätze, was jedoch alle gemeinsam haben ist, dass sie eine Quelle und eine sichere Methode um diese Quelle zu verifizieren brauchen.
Für die Quellen gibt es viele verschieden möglichkeiten, bei der Entwicklung von fsverify hatte ich die Wahl auf zwei möglichkeiten begrenzt, da beide sehr einfach zu implementieren sind, und dadurch die Verifizierung der Quellen auch einfach ist.

- Externe Partition

  Hier wird eine Datenbank an Hashes zusammen mit allen Metadaten in eine extra Partition geschrieben, diese Partition kann auf ein Externes medium geschrieben werden, und nur dann angeschlossen sein, wenn das System die Verizifizierung durchführt. Jedoch braucht dies entweder eine seperate Partition auf der Festplatte, wodurch die nutzbare Speicherkapazität sich verringert, oder ein externes Medium, welches nicht immer vorhanden ist.

- Einfache Datei

  Hier wird die Datenbank einfach in einem Ort gespeichert, auf die das Program während der Verifizierung zugreifen kann. Dies ist sehr einfach zu implementieren und benötigt keine externen Partitionen oder Speichermedien. Das Problem ist es jedoch, die Datei an einem ort zu speichern, bei der man nicht unverifizierte Dateisysteme anhängen muss oder ungeschützt ohne schreibschutz offen ist.

Um die Quelle zu schützen beziehungsweise zu Verifizieren, gibt es zwei Methoden:

- Kryptographische Verifizierung

  Die Entwickler des Betriebssystems müssen hierbei bei dem aufsetzen des Verifizierungsprogramms die Hash Quelle Kryptographisch mit ihren privaten Schlüsseln signieren (zum Beispiel mit GnuPG oder Minisgin), das Verifizierungsprogramm erhält den öffentlischen Schlüssel der Entwickler, die Signatur und die Quelle, wodurch es anhand der Signatur verifizieren kann, dass die Quelle von den Entwicklern stammt und nicht modifiziert wurde.

  Hierbei ist das größte Problem, dass der öffentliche Schlüssel gut geschützt werden muss, damit die Signatur und Schlüssel nicht mit der eines Attackers ersetzt werden kann.

- Verschlüsselung

  Die Quelle ist mit einem zufällig generierten Schlüssel verschlüsselt, welcher in dem Quellcode des Verifizierungsprogrammes geschrieben wird, um somit den Schlüssel direkt im Programm zu speichern. Dadurch können keine Schlüssel ersetzt werden, jedoch ist es immer möglich, den Schlüssel aus dem Programm zu extrahieren, ohne überhaupt auf das System zugreifen zu müssen, da man das Betriebssystem selber installieren kann. Sobald der Schlüssel bekannt ist, kann die Datei einfach verschlüsselt und ohne Probleme modifiziert werden.

1.3 Gewählte Implementation

Im anbetracht existierender Dateiverifizierungsprogrammen wie Androids dm-verity und mein vorheriges, ähnliches Projekt FsGuard.
Für die Implementation habe ich die Blockverifizierung ausgewählt, da sie durch Multithreading sehr schnell sein kann, aber auch neue Datein bemerkt, welches die Per-Datei Verifizierung nicht gewährleistet.
Um die Hashes zu Speichern wird ein eigenes Partitionsschema benutzt, welches alle Metadaten und die Datenbank beinhaltet. Der minisign öffentliche Schlüssel kann durch mehrere Methoden gespeichert werden, wie einer Textdatei oder einem gerät welches über USB-Serial den Schlüssel übergibt.
Weitere Eintscheidungen für die Implementation sind:

- Programmiersprache: go

go ist mir vertraut und memory safe, welches für die Sicherheit des Programmes eine große Rolle spielt.

- Datenbank: bbolt
  bbolt ist eine Datenbank welche direkt in go geschrieben wurde und somit ein Robusteren API als sqlite hat, zudem ist bbolt unter einer richtigen lizens lizensiert und wirkt moderner.

2 Realisierung

Das Projekt kann in drei Unterprojekte eingeteilt werden. Fsverify, also die verifizierung selber, verifysetup, ein Program um das system richtig zu Konfigurieren um die nutzung von fsverify möglich zu machen und fbwarn, ein program welches den Nutzer graphisch über eine fehlgeschlagene Verifizierung informiert.

2.1 fsverify

Da das Konzept der Festplattenverifizierung nichts neues ist, habe ich mir erstmals bereits existierende Projekte angeschaut, um zu sehen, wie es in anderen Betriebssystemen realisiert ist. Hierbei war google's dm-verity, welches in Android und ChromeOS geräten genutzt wird, die beste Hilfe, da es am besten dokumentiert und ausgetestet ist.

2.1.1 Partitionslayout

Inspiriert an dm-verity, entschied ich mich dafür, die Datenbank auf eine eigene Partition zu speichern, also war das erste Ziel ein gutes Partitionslayout zu Entwickeln, in der die Datenbank und Metadata so gut wie möglich gespiechert werden kann.
Die erste Version des Layouts war recht simpel, es hatte alles was wirklich wichtig war, eine magic number, die signatur, größe des Dateisystems und größe der Datenbank:

```
<magic number> <signature> <filesystem size> <table size>
```

Feld	Größe	Nutzen	Wert
magic number	2 bytes	Sanity check	0xACAB
signature	302 bytes	minisign signatur	-
filesystem size	4 bytes	größe des originalen Dateisystems in GB	-
table size	4 bytes	größe der Datenbank in MB	-