FsVerify

Xenia

Reflexion zum
AP Informatik

Contents

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	-

In der implementierung dieses Layouts viel dann auf, dass es keinen Sinn macht, die Datenbankgröße in MB festzulegen

Die zweite Version fügt ein weiteres Feld hinzu um die Einheit der Datenbankgröße festzulegen:

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

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	-
table unit	1 byte	datentyp des Feld “table size”	-

Die nächste version teilte die Signatur in zwei teile auf. Da minisign signaturen aus einem kommentar, einer vertrauten signatur, einem weiteren kommentar und einer nicht vertrauten signatur

```
<magic number> <untrusted signature hash> <trusted signature hash>
<filesystem size> <table size> <table unit>
```

Feld	Größe	Nutzen	Wert
magic number	2 bytes	Sanity check	0xACAB
untrusted signature	100 bytes	nicht vertrauter signatur	-
trusted signature	88 bytes	vertraute signatur	-
filesystem size	4 bytes	größe des originalen Dateisystems in GB	-
table size	4 bytes	größe der Datenbank in MB	-
table unit	4 bytes	datentyp des Feld “table size”	-

2.1.2 Datenbanklayout

Nachdem der Header der Partition festgelegt wurde, muss festgelegt werden, wie die Datenbank festgelegt ist. bbolt, die Datenbankbibliothek die fsverify nutzt, hat ein key/value system, das heißt, dass jeder Wert mit einem Schlüssel verbunden ist. Zudem

benutzt bbolt das konzept von “Buckets”, einem Eimer in dem Datenpaare sortiert werden können.
Das erste Layout war für eine implementation von fsverify die nur auf einem Thread läuft, besteht aus einem Bucket “Nodes”, in dem jede Node gespeichert wird. Eine Node sieht wie folgt aus:

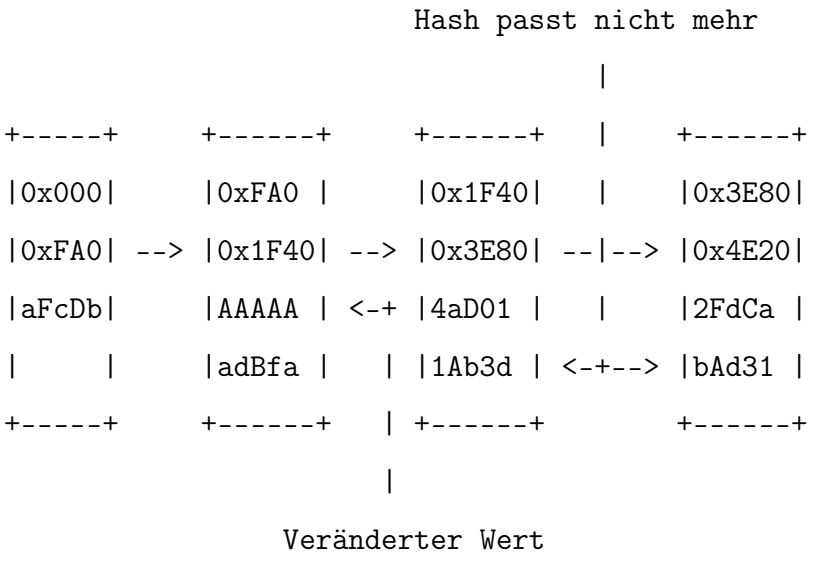
```
// Node.go
type Node struct {
  BlockStart int
  BlockEnd int
  BlockSum string
  PrevNodeSum string
}
```

Feld	Nutzen
BlockStart	Der hex offset and dem der Block anfängt
BlockEnd	Der hex offset and dem der Block ended
BlockSum	Der sha1 hash des Blocks
PrevBlockSum	Der sha1 hash aus allen Feldern der vorherigen Node

Jeder Block, welcher 2kb groß ist, bekommt eine Node zugewiesen, diese Nodes werden in der Datenbank aneinandergereiht, mit dem wert von PrevBlockSum als den key. Der Wert PrevBlockSum erlaubt es, während der Verifizierung Fehler in der Datenbank zu finden. Wird eine Node verändert, stimmt der PrevBlockSum der nächsten Node nicht mehr, dass heißt, dass es nicht mehr möglich ist, den Key zu der nächsten Node zu berechnen, wodurch die Verifizierung fehlschlägt.

```
+-----+     +------+     +------+        +------+
|0x000|     |0xFA0 |     |0x1F40|        |0x3E80|
|0xFA0| --> |0x1F40| --> |0x3E80| -----> |0x4E20|
|aFcDb|     |cDfaB |     |4aD01 |        |2FdCa |
|     |     |adBfa |     |1Ab3d |        |bAd31 |
+-----+     +------+     +------+        +------+
```

Wird hier eine Node verändert, stimmt die restliche Kette nicht mehr



Da die erste Node keinen vorränger hat, von dem es PrevNodeSum berechnen kann, wird ihr der wert "Entrypoint" gegeben.
Diese Datenbankstruktur hat ohne Probleme funktioniert, jedoch war fsverify viel zu langsam wenn es auf einem Thread läuft. Das Problem bei dem Multithreading jedoch ist, dass man Nodes nicht wahrlos aufgreifen kann, sondern eine vorherige Node oder die entrypoint Node braucht. Die Lösung ist recht einfach, die anzahl der Threads wird in verifysetup bereits angegeben und somit in fsverify fest einprogrammiert. Somit gibt es in der Datenbank mehrere entrypoint Nodes, die sich durch eine hinzugefügte Nummer unterscheiden. Dadurch ist es weiterhin möglich die Datenbank zu verifizieren, während es multithreaded läuft.

2.1.3 Datenbanksignatur

Um sicherzustellen, dass die Datenbank nicht modifiziert wurde, wird eine Signatur generiert die mit der gelesenen Datenbank verglichen wird.
Wie bereits erwähnt, wird für die Signatur das Programm minisign benutzt. Minisign beruht auf ein public/private key system, wodurch eine Signatur von dem privaten Schlüssel generiert wird und durch den öffentlichen Schluss verifiziert werden kann.
Die Signatur wurde bereits im Partitionsheader gespeichert, was übrig bleibt ist der öffentliche Schlüssel.

Da der öffentliche Schlüssel und die Signatur gebraucht werden, um eine Datenbank

zu verifizieren, muss sichergestellt werden, dass beide seperat gespeichert werden und zumindest der öffentliche Schlüssel nicht bearbeitet werden kann.
Die erste Idee um dies zu lösen wäre, dass man einfach den Schlüssel in eine Datei schreibt, und die Datei schreibgeschutzt Speichert. Jedoch ist bei diesem weg der speicherort der Datei das problem, wie soll man sicher sein, dass nicht das ganze Dateisystem verändert wurde um einen neuen Schlüssel zu beinhalten?

Das heißt, dass man ein Schreibgeschütztes, möglichst seperates und immer vertraubares Speichermedium braucht, auf der man den Schlüssel speichert.
Die lösung: Microcontoller. Sie können über usb-serial (also /dev/ttyACM* in Linux) Daten übertragen, können durch das Modifizieren bestimmter Sektoren permanent schreibgeschützt werden, und sind sehr klein, also können sie von dem Nutzer mitgetragen werden oder in dem Gerät direkt verbaut sein.
Um dieses konzept zu testen, habe ich einen Arduino UNO genutzt, dieser ist zwar immer schreibbar, hat aber keine Technischen unterschiede die die Datenübertragung ändern würden.
Der Code für den Arudino sieht wie folgt aus:

```
// publicKey.c
void setup() {
  Serial.begin(9600); // set up a serial tty with the baud rate 9600
  Serial.print("\tpublic key\t"); // Write the public key to the tty
}
void loop() {}
```

Es wird eine Serielle Konsole auf einer Baudrate von 9600 geöffnet, auf der einmalig der öffentliche Schlüssel ausgegeben wird. Es ist wichtig zu beachten, dass der Schlüssel mit Tabstops (t) ausgegeben wird, diese benutzt fsverify um zu wissen, ob der volle Schlüssel aufgenommen wird, fehlt der Tabstop am anfang oder am Ende, ist es sehr wahrscheinlich, dass der Schlüssel auch nicht vollständig aufgenommen wurde.

2.1.4 Optimierung

Wie bereits gesagt, lief die erste version von fsverify auf einem Thread, dies führte zu einer laufzeit von über einer Stunde bei einer Partitionsgröße von 1gb. Da fsverify beim booten

des systems ausgeführt werden soll, ist eine laufzeit von einer Stunde nicht akzeptabel.
Die ersten schritte der Optimierung war es, die größe der Blocks zu verringern und von sha256 zu sha1 zu wechseln. Da das lesen von daten viel schneller erfolgt als das hashen von daten, ist es besser mehr zu lesen und dadurch kleinere Datenmengen zu hashen, der wechsel von sha256 zu sha1 mag erstmal schlecht wirken, jedoch macht dies keine Probleme, da hier keine Passwörter oder ähnliches verschlüsselt werden, also sind bruteforce attacken hier kein risiko.
Mit diesen Optimierungen hat sich die Laufzeit etwas verbessert, von 60 Minuten zu ungefähr 50. Nicht viel besser.
Der nächste schritt war es, fsverify mit multithreading zu implementieren, die dafür notwendigen änderungen in der Datenbank wurden bereits erklärt. In fsverify selber hat sich die art geändert, wie die Daten von der Partition gelesen werden. Anstatt alles auf einmal zu lesen und durchzugehen, wird die größe der Partition genommen, durch die anzahl der Threads geteilt, und somit für jeden Thread genau die anzahl an bytes gelesen, die für die Node-kette nötig ist. Dies stellt sicher, dass keine Kette sich überlappt und korruptionen von Nodes in ketten auffallen, da sie durch Korruptionen versuchen könnten, bytes zu lesen die sie garnicht lesen sollten.
Durch das multithreading hat sich die Laufzeit von den singlethreaded 50 Minuten zu nur 6 Sekunden verringert.

Fsverify hatte eine Laufzeitoptimierung von 60000% in einer Woche:

```
10.02.2024: fsverify takes 60minutes to complete for 1gb
optimizations: none

12.02.2024: fsverify takes 52minutes to complete for 1gb
optimizations: block size 2k, sha1 instead of sha256

17.02.2024: fsverify takes ~6 seconds to complete for 1gb with 12 threads (p7530)
optimizations: block size 2k, sha1 instead of sha256, multithreaded, db batch operatio
unoptimizations: manual connecting of arduino, ~1 second penalty
```