Softwaredesign für Dynamische Integritätsmessungen bei Linux

K.-O. Detken[1] · M. Jahnke[1] · T. Rix[1] · A. Rein[2] · M. Eckel[2]

[1]DECOIT GmbH, Fahrenheitstr. 9, D-28359 Bremen
detken/jahnke/rix@decoit.de

[2]Huawei Technologies Düsseldorf GmbH, Feldbergstr. 78, D-64293 Darmstadt
andre.rein/michael.eckel@huawei.com

Zusammenfassung

Die meisten Sicherheitstools versuchen schädliche Programme anhand ihrer Signatur oder anhand ihres Verhaltens zu erkennen. Dies hat allerdings einen entscheidenden Nachteil, denn damit diese Erkennung funktionieren kann, muss das Schadprogramm oder Verhalten bereits bekannt sein. Ein anderer Ansatz ist es, die ausführbaren Programme direkt auf Änderungen im Programmcode zu überwachen. Dies macht z.B. die Integrity Measurement Architecture (IMA) im Linux-Kernel. Diese misst Userspace- und/oder Kernelspace-Programme bevor diese ausgeführt werden. Aber viele Systeme werden nur einmal gestartet und laufen dann lange Zeit, ohne dass sie neugestartet werden. Das bedeutet, dass während der Laufzeit auftretende Programmcode-Änderungen nicht erkannt werden können. Daher verfolgte dieses Kooperationsprojekt den Ansatz der *Dynamic Runtime Attestation* (DRA), basierend auf dem Vergleich zwischen geladenem Programmcode und bekannten Referenzwerten. DRA ist ein komplexer Ansatz, der verschiedene Komponenten und komplexe Attestierungsverfahren beinhaltet. Daher ist eine flexible und erweiterbare Architektur erforderlich. In dem Kooperationsprojekt wurde eine Architektur entwickelt und erfolgreich in einem Prototyp umgesetzt. Um die nötige Flexibilität und Erweiterbarkeit zu erreichen, ist in den beteiligten Komponenten die Attestierungsstrategie (Guideline) zentral umgesetzt. Die Guidelines definieren die nötigen Schritte für alle Attestierungsoperationen, wie z.B. Messungen, Referenzwert-Generierung und Verifikation.

1 Einleitung

Heutige Systeme sind gegenüber einer Vielzahl von Angriffen anfällig. Gerade Laufzeitangriffe werden zunehmend häufiger. Sehr ausgeklügelte Angriffe zielen auf die Modifikation von Programmen nach ihrer Ausführung direkt im Speicher ab. Diese Angriffe können von statischen Integritätsmessungen (z.B. IMA) nicht erkannt werden. Daher wurde das *Dynamic Runtime Integrity Verification and Evaluation* (DRIVE) [REIN17] entwickelt. DRIVE ermöglicht die Erkennung vieler Angriffe, die auf die Manipulation im Speicher abzielen, erfordert aber komplexe Strategien für eine erfolgreiche Attestierung. Daher ist es notwendig, eine flexible und erweiterbare Architektur zu entwickeln, die Software- und Hardware-Sicherheitstechnologien verbindet.

In dieser Arbeit wird das Konzept der Guidelines vorgestellt. Guidelines repräsentieren flexible und erweiterbare Strategien für alle notwendigen Attestierungsschritte, wie z.B. Messung,

P. Schartner · N.N (Hrsg.) · DACH Security 2017 · syssec (2017) pp-pp.

Referenzwertgenerierung und Verifikation. Architektur und Guideline-Ansatz sind von folgenden Eigenschaften geprägt:

1. **Flexibilität:** Unabhängige Attestierung von Systemkomponenten, z.B. Userspace- und Kernelspace-Komponenten/-Prozesse
2. **Starke Entkopplung:** Durch die Einführung von funktional verbundenen Guideline-Tripeln, die andere Tripel nicht beeinträchtigen.
3. **Erweiterbarkeit:** Durch klar und präzise definierte Schnittstellen in allen relevanten Komponenten, die eine einheitliche Kommunikation ermöglichen.

Um die entworfene Architektur zu verifizieren, wurden alle benötigten Komponenten und Guidelines für verschiedene Systemkomponenten implementiert.

2 Verwendete Technologien

Es werden an dieser Stelle die Basistechnologien des DRIVE-Ansatzes vorgestellt.

2.1 Trusted Platform Module (TPM)

Das *Trusted Platform Module (TPM)* ist ein Computerchip, der Schlüsselmaterial beinhaltet, mit dem ein Gerät authentifiziert werden kann. Das Schlüsselmaterial besteht aus RSA-Schlüsselpaaren, dem Endorsement Key (EK) und dem Storage Root Key (SRK), die vom TPM zusammen mit einem nutzerspezifischen Passwort generiert werden. Der TPM kann auch dazu genutzt werden, Messungen zu speichern, bzw. so zu signieren, dass deren Authentizität garantiert werden kann. Authentizität und Attestierung sind notwendige Schritte, um die vertrauenswürdige Verwendung von Computern oder anderen Plattformen zu sichern. Bei der Authentifikation wird sichergestellt, dass ein Gerät jenes ist, welches es vorgibt zu sein. Attestierung ist ein Prozess, der beweist, dass eine Plattform vertrauenswürdig ist und nicht kompromittiert wurde. Die Natur der hardwarebasierten Kryptographie stellt sicher, dass in Hardware gespeicherte Informationen besser vor Softwareattacken geschützt sind [TCG08]. Die in dieser Arbeit vorgestellte Software-Architektur verwendet hauptsächlich die Funktionen *tpm_extend()* und *tpm_quote()*, um die Messwerte zu sichern.

Für die Speicherung von Hashwerten hat der TPM mehrere *Platform Configuration Register (PCR*). PCR werden dazu genutzt Hashwerte zu halten, die per *extend*-Befehl generiert wurden. Der gespeicherte Hashwert kann unter anderem den Plattformstatus repräsentieren [WADC15]. Dabei verwendet der *extend*-Befehl immer einen Startwert *A* (dieser repräsentiert den aktuellen PCR Wert), verknüpft ihn mit einem neuen übergebenen Wert B, berechnet einen neuen Hashwert und ersetzt den originalen Wert A im PCR mit dem neu errechneten.

Der TPM-*quote*-Befehl generiert eine Signatur über ein Set von PCR-Werten. Diese Signatur erlaubt entfernten Systemen festzustellen, dass die empfangenen Werte nicht manipuliert worden sind. Das bedeutet, dass eine Liste von im TPM verankerten Werten (*extended*) nicht geändert wurde.

2.2 Speichermanagement

Beim Booten reserviert der Linux Kernel einen Teil des Arbeitsspeichers für den eigenen Bedarf. Dieser Bereich wird auch *Kernelspace Memory* genannt. Der restliche Arbeitsspeicher,

auch *Userspace Memory* genannt, steht dann den benutzerspezifischen Programmen zur Verfügung.

Innerhalb des Kernels gibt es verschiedene Arten von Adressen, um auf bestimmte Bereiche des Speichers zuzugreifen. Die physikalische Speicheradresse benennt die aktuelle Position im physischen Speicher. Des Weiteren existieren verschiedene virtuelle Adressen, von denen nur drei für die vorliegende Arbeit von Relevanz sind: Kernel-logische-Adresse und Kernel-/Benutzer-virtuelle-Adresse. Über die beiden Kernel-Adressen wird aus dem Kernel heraus auf den Speicher zugegriffen. Während die logische Adresse ein direktes Mapping auf eine physische Adresse darstellt, verwendet die virtuelle Adresse ein abstrahiertes Mapping und kann auf Speicher verweisen, der sich nicht mehr im Arbeitsspeicher befindet, also z.B. ausgelagert wurde. Die Benutzer-virtuellen-Adressen werden von Userspace-Prozessen genutzt. Dieser Adresstyp beginnt immer mit der Adresse 0x00 und ist fortlaufend innerhalb jedes einzelnen Prozesses. Der Kernel verbindet diese Adressen zu physischen Adressen im Speicher, die nicht zwingend fortlaufend sein müssen. Daneben gibt es noch die Bus-Speicher-Adresse, diese ist aber für vorliegende Arbeit nicht von Interesse und wird deshalb nicht weiter erklärt.

Der Kernel unterteilt den Systemspeicher in Seiten (Pages). Jede Seite hat dabei eine feste Größe. Bei auf x86 und x86_64 basierenden Systemen beträgt die Seitengröße meistens 4096 Bytes und stellt gleichzeitig die kleinste Einheit dar, die vom Kernel verwaltet werden kann. Jede Seite wird von einem Seitentabelleneintrag (*Page Table Entry, PTE*) identifiziert. Ein PTE enthält, abhängig von der CPU-Architektur, eine Anzahl von Flags, die unter anderem Seitenstatus, Zugriffsrechte und andere Metadaten darstellen. Diese Flags sind wichtig für das Speichermanagement.

Die Seitentabelleneinträge werden in einem Multi-Level-Wörterbuch organisiert. Dies ermöglicht das schnelle Suchen einer PTE im virtuellen Speicher. Hierfür wird die sogenannte Seitenrahmennummer (*Page Frame Number, PFN*) mit in die PTE kodiert. Die PFN identifiziert die Seite und kann mit Hilfe einer Bitshift-Operation auf die jeweilige Seite zugreifen.

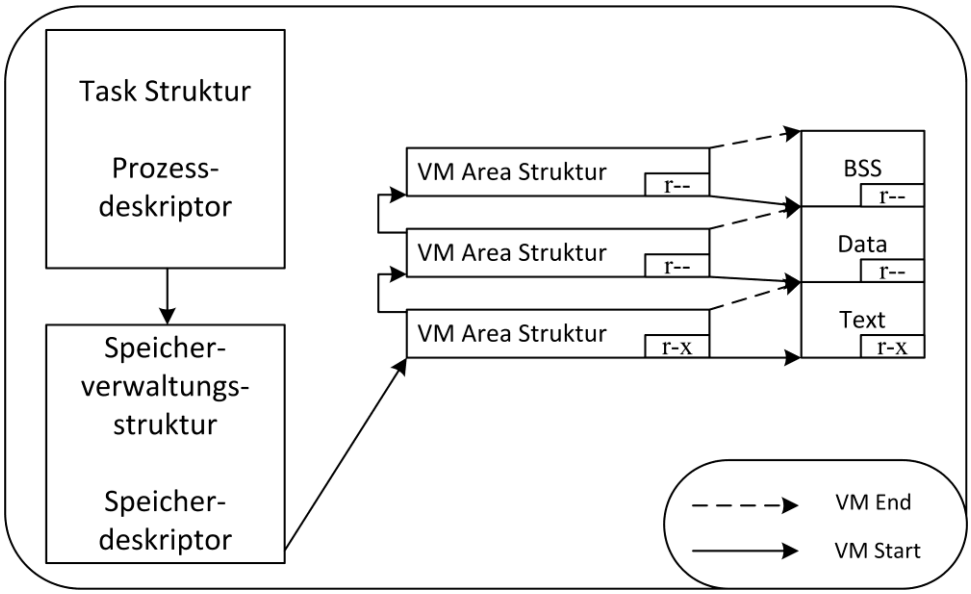

Abb. 1: Linux-Kernel Speichermanagement

Wenn das System immer mehr Arbeitsspeicher benötigt als physikalisch zur Verfügung steht, werden Seiten in den Auslagerungsspeicher (Swap-Speicher) verschoben. Der Swap-Speicher

ist ein speziell reservierter Speicherbereich auf der Festplatte des Systems. Dieser Prozess wird auch *Swapping* genannt und macht die betreffende Seite nicht mehr direkt erreichbar. Wird diese Seite benötigt, muss diese erst von der Festplatte gelesen und dann in den Arbeitsspeicher geladen werden. Dieser Vorgang dauert aufgrund der langsameren Zugriffszeiten und dem Kopiervorgang deutlich länger, als der Zugriff dauern würde, wenn die Seite direkt im Arbeitsspeicher läge. Aus diesem Grund wird diese Operation nur ausgeführt, wenn das System keinen freien Arbeitsspeicher mehr hat. Von diesem Prozess sind allerdings nur die Userspace-Seiten betroffen. Ob eine Seite im Speicher vorhanden ist, kann anhand der *Present-Flags* im PTE erkannt werden. Damit der Userspace einfacher verwaltet werden kann, nutzt der Kernel verschachtelte Datenstrukturen (siehe Abb. 1).

Die *Task-Struktur* bildet einen kompletten Userspace-Prozess ab und stellt somit die Top-Level-Struktur des Linux-Kernel-Speichermanagements dar. Darunter angeordnet ist die Speicherverwaltungsstruktur, die diesem Prozess zugewiesene Informationen über den Speicher enthält.

Der Prozessspeicher ist in verschiedene Segmente unterteilt, aus denen der Prozess geladen wurde. Jedes Segment wird durch die ELF-Datei (*Executable and Linking Format*) definiert und über die virtuelle Speicherbereichsstruktur (*Virtual Memory Area, VM Area*) dargestellt. Die VM Area enthält neben anderen Informationen, die virtuelle Adresse, welche den Anfang und das Ende des Segmentes bezeichnet, sowie einige Flags. Die VM-Area-Flags stellen eine höhere und plattformunabhängige Darstellung der PTE-Flags dar. Alle Seiten eines Speichersegments müssen diese Flaggen auf ihrem individuellen PTE reproduzieren. Der Kernel bietet Funktionen zum Auflösen eines PTE aus einer virtuellen Benutzeradresse und der entsprechenden Speicherverwaltungsstruktur.

2.3 Statische und Dynamische Integritätsmessung

Statische Integritätsmessungen werden im Allgemeinen durch die *Integrity Measurement Architektur (IMA)* durchgeführt. IMA ist eine Erweiterung des Linux-Kernels und ermittelt geänderte Programme. Hierfür verwaltet IMA ein Messprotokoll mit Prüfsummen, welche von ausführbaren Instruktionen (z.B. Programmen, Bibliotheken) und Strukturierten Daten (z.B. Konfigurationen) stammen. Das Messprotokoll ist gegen den TPM gesichert, indem jede Prüfsumme an den TPM gesendet wird, der daraus einen Hash mit dem vorherigen berechnet und diesen wieder in einem PCR ablegt. Um zu verifizieren, ob das System kompromittiert ist, kann das Messprotokoll dahingehend überprüft werden, ob die gemessenen Prüfsummen den erwarteten entsprechen. Der Prozess kann lokal auf dem System oder auf einem entfernten System ausgeführt werden. Dieser Ablauf wird auch Remote Attestation genannt und ist in Abb. 2 vereinfacht dargestellt.

IMA erstellt eine SHA1-Prüfsumme, wenn die Datei eines Programmes oder eine gemeinsam verwendete Bibliothek ausgeführt, bzw. genutzt wird. Das Ergebnis wird dann in dem Messprotokoll abgelegt und gegen den TPM gesichert. Ein lokal oder entfernt realisiertes Verifikationssystem nimmt die Liste, verifiziert deren Integrität auf Basis einer PCR-Nachberechnung und überprüft anschließend, ob alle Werte den erwarteten entsprechen. Ist das der Fall, kann das System als integer angesehen werden und ist somit vertrauenswürdig. Diese Information kann dann z.B. als Entscheidungsgrundlage für ein *Network Access Control (NAC)* System dienen und dem Gerät Zugriff auf das Firmennetz gewähren. Dynamische In-

tegritätsmessung, *Dynamic Runtime Integrity Verification and Evaluation* DRIVE [REIN17], beschreibt ein Konzept zur Erfassung und Verifikation von Integritätsdaten zur Programmlaufzeit. DRIVE basiert auf ähnlichen Konzepten wie IMA, jedoch ist das Mess- und Verifikationsverfahren weitaus komplexer. Dadurch ergeben sich diverse Anforderung an die Architektur und das Design des Systems, die im Folgenden besprochen werden.

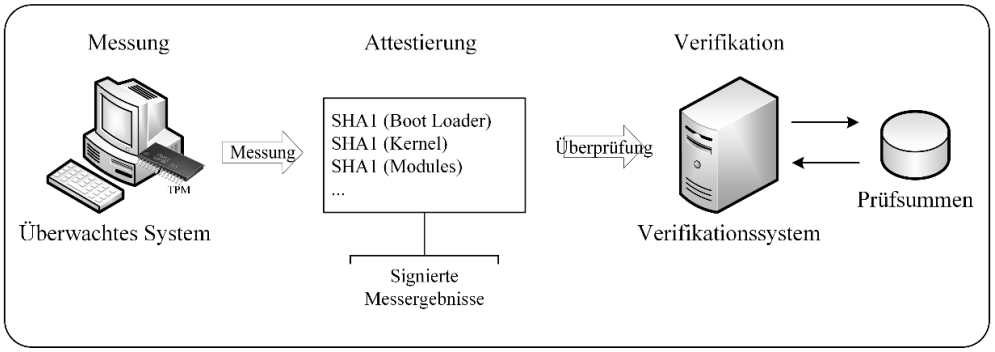

Abb. 2: IMA-Prozessablauf

3 Software-Design

In diesem Kapitel wird das entwickelte Softwaredesign beschrieben und erklärt, wie das Mess- und Verifizierungssystem strukturiert ist, welche Komponenten es beinhaltet, welche Aufgaben diese erfüllen und wie sie interagieren.

3.1 Allgemeine Architektur

Allgemein gesehen besteht das gesamte System aus drei Komponenten, die in Abb. 3 verdeutlicht sind. Weiterhin ist zu erkennen, wie die drei Komponenten miteinander interagieren. Auf dem zu überwachenden System wird ein Kernelmodul (DKM) ausgeführt. Dieses stellt einen Dateisystemknoten bereit, um mit dem Modul kommunizieren zu können und Messungen von laufenden Userspace-Prozessen oder Kernelmodulen und den Kernel selbst zu initiieren.

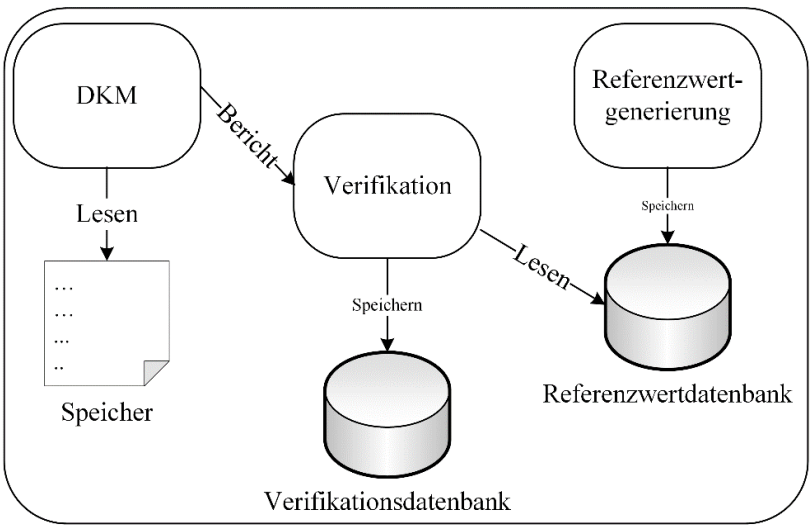

Abb. 3: High-Level-Architektur

Die Referenzwertgenerierung stellt die zweite Komponente dar und sollte auf einem sicheren System ausgeführt werden. Sie erzeugt die Referenzwerte, die zur Validierung benötigt werden, und speichert diese in einer Datenbank. Der Vergleich der Referenzwerte mit den vom DKM gemessenen Werten erfolgt in der Verifikation. Dieser erhält die Messwerte vom DKM, sucht dann in der Referenzwertdatenbank nach den entsprechenden Referenzwerten und vergleicht diese miteinander. Das Ergebnis wird in der Verifikationsdatenbank zur späteren Nachvollziehbarkeit abgelegt.

3.2 DRIVE-Kernel-Modul (DKM)

Das *DRIVE-Kernel-Modul (DKM)* ist ein Kernelmodul für den Linux-Kernel. Es implementiert ein Framework für die Messung von Userspace- und Kernelspace-Speicherbereichen und stellt eine Schnittstelle zu den gemessenen Werten bereit. Die Architektur des Moduls kann in fünf Hauptbereiche unterteilt werden (siehe Abb. 4).

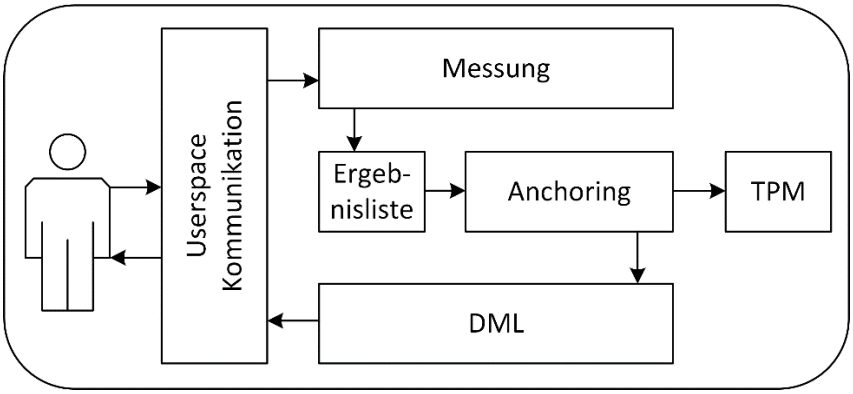

Abb. 4: DKM-Architektur

Die erste Komponente (*Userspace Kommunikation*) verwaltet die Kommunikation mit dem Userspace mithilfe eines Character Device oder per SecurityFS. Beide werden im Folgenden als Device bezeichnet. Das Device unterstützt das manuelle Messen von einzelnen Messzielen oder das gesamte System. Messziel kann ein einzelner Prozess, ein Kernelmodul oder das Abbild des aktuell laufenden Kernels sein. Das andere Szenario für das Device wäre das Auslesen der aktuellen Messliste mithilfe eines Lese-Befehls auf dem Device. Die Messliste wird dann in einem Binärformat an den lesenden Prozess (z.B. *cat*) gesendet. In dieser Beispielumsetzung wird das Format *Concise Binary Object Represantation (CBOR)* [CBPH12] genutzt. Allerdings ist es möglich, dieses durch eine andere Implementation auszutauschen, wie z.B. durch *ASN1*.

Die zweite Komponente (*Messung*) des DKM führt die eigentliche Messung unter Berücksichtigung des Messziels aus. Sie beinhaltet das Anreichern der internen Datenstruktur mit weiteren Informationen über das Messziel, das Auswählen und Ausführen der entsprechenden Guidelines (siehe **Fehler! Verweisquelle konnte nicht gefunden werden. Fehler! Verweisquelle konnte nicht gefunden werden.**) und die Generierung der eigentlichen Messwerte. Das Ausführen dieser Operationen innerhalb der Komponente erfolgt asynchron unter Zuhilfenahme des Linux-Kernel-Arbeitsvorrats (*Work Queue*). Dieser wird entweder mit Befehlen aus dem Userspace aufgerufen oder regelmäßig von einem Timer ausgeführt. Letzteres wird durch das Einfügen eines Messauftrages in den Work Queue realisiert, der eine vollständige Systemmessung auslöst, sobald die definierte Zeitverzögerung abgelaufen ist. Nach der voll-

ständigen Systemmessung wird automatisch wieder ein Messauftrag mit der gleichen Zeitverzögerung in die Work Queue eingefügt. Die Zeitverzögerung kann beim Laden des Moduls per Parameter definiert werden. Der gleiche Parameter erlaubt auch das Deaktivieren des Timers.

Die Ergebnisse der vorherigen Komponente werden in einer verketteten Liste eingefügt, welche die Ergebnisse für die Weiterverarbeitung zwischenspeichert. Die Einträge der Liste werden von der dritten Komponente (*Anchoring*) abgerufen. Diese Komponente entfernt Metadaten aus den Einträgen der vorherigen Liste, die für die weitere Verarbeitung nicht mehr benötigt werden, wie z.B. Informationen zum Messziel und einige Zeitstempel die während der Messungen erstellt wurden. Dadurch wird auch der Speicherverbrauch der Messergebnisse reduziert. Die reduzierten Messungen werden dann an die *Dynamic Measurement List* (DML), gesendet. Diese stellt die letzte Komponente des Kernelmoduls dar. Danach wird die DML gegen den TPM via *extend*-Befehl gesichert.

Die Beispielumsetzung des DKM unterstützt sowohl den TPM 1.2, als auch den TPM 2.0 [TCG15]. Aufgrund der begrenzten TPM 2.0-API-Unterstützung des Linux-Kernels können jedoch nicht alle Funktionen des TPM 2.0 genutzt werden. Zum Beispiel werden nur SHA-1 Hashes bei der PCR-*extend*-Operation unterstützt.

Die reduzierten Messergebnisse werden in das CBOR-Format umgewandelt und über den daraus resultierenden Bytestring wird erneut eine SHA-1-Prüfsumme gebildet. Die Prüfsumme wird dann per PCR-*extend*-Befehl an den TPM gesendet. Das konkrete PCR muss beim Laden des Moduls angegeben werden. Der PCR-*extend*-Befehl nimmt den aktuellen PCR-Wert, fügt den neuen Wert an und berechnet dann eine neue Prüfsumme, die im PCR als neuer Wert abgelegt wird.

Die fünfte und letzte Komponente, die DML, ist eine verkettete Liste, mit allen Messergebnissen, die seit dem Modulstart erzeugt wurden. Sie fügt neue Werte hinzu und ermöglich das Lesen der Liste. DML ist, bis auf das Einfügen von Werten, unveränderlich.

3.3 Referenzwert-Generierung (RVG)

Die Referenzwert-Generierung (*Reference Value Generation, RVG*) ist das Gegenstück zum DKM. Hier wird ebenfalls ein Framework zur Verfügung gestellt, das in diesem Fall die Referenzwerte erzeugt. RVG verwendet ebenfalls Guidelines (siehe **Fehler! Verweisquelle konnte nicht gefunden werden. Fehler! Verweisquelle konnte nicht gefunden werden.**), die, je nach zugehörigem Messziel, spezifisch definiert und implementiert sind. RVG ist ein Userspace-Prozess, der normalerweise auf einem anderen System ausgeführt werden sollte. Dieser benötigt alle ELF-Dateien, die auf dem zu überwachendem System gemessen werden sollen. Diese Dateien müssen die exakt gleiche Version haben, wie die auf dem zu überwachendem System vorhandenen Dateien. Allerdings sollten sie aus einem sicheren Vergleichssystem stammen, bei dem sichergestellt wurde, dass es nicht kompromittiert ist.

Die ELF-Dateien können innerhalb eines Ordners oder Ordnerstruktur hinterlegt sein, z.B. als ein Auszug aus dem sicheren Vergleichssystem. Wird RVG auf dieses Verzeichnis konfiguriert, liest es alle ELF-Dateien, die er findet und generiert die dazugehörigen Referenzwerte. Wie die Referenzwerte gebildet werden, wird in der jeweiligen Guideline definiert.

Neben dem Generieren von Referenzwerten bietet RVG die Möglichkeit Parameter über das überwachende System zu hinterlegen. So beinhaltet es unter anderem das PCR, welches vom DKM für den PCR-*extend*-Befehl genutzt werden soll, die von dem TPM auf dem Zielsystem erzeugte Version der *QUOTE_INFO*-Datenstruktur und die Konfiguration des aktuell laufenden Kernels. Diese Informationen werden für die Integritätsverifikation der empfangenen DML benötigt. Darüber hinaus können weitere Parameter hinterlegt werden.

Nachdem die Referenzwerte erzeugt wurden, werden die Werte in einer Datenbank abgelegt. Es ist nötig, dass sowohl RVG, als auch die Verifikation Zugriff auf diese Datenbank haben.

3.4 Verifikation

Die Verifikation komplettiert die benötigten Tools, um die generelle Architektur des Systems benutzen zu können. Es verifiziert die vom DKM empfangenen Messwerte, mithilfe der Referenzwerte, die von RVG erzeugt wurden. Anhand der Verifikation kann also bestimmt werden, ob ein gemessener Systembestandteil kompromittiert ist oder nicht. Wird ein kompromittierter Bestandteil erkannt, können weitere Schritte eingeleitet werden (Remediation). Die Verifikation könnte daher in ein Remote-Attestation-System integriert werden, das wiederrum Teilsystem eines Remediation-Systems ist. Anhand des Attestierungsergebnisses könnten dann wiederrum entsprechende Maßnahmen eingeleitet werden, wie z.B. das System ausschalten oder neu starten.

Auf dem zu überwachenden System wird mithilfe eines Tools ein *System State Report (SSR)* erstellt, der ebenfalls in CBOR codiert ist. Dieser besteht aus einer Liste, die zwei Elemente beinhaltet. Das erste Element besteht aus dem TPM-Quote zum Zeitpunkt der Erstellung des SSR und das zweite Element ist die in CBOR codierte DML. Der SSR wird nicht im DKM erzeugt, sondern mit dem sogenannten SSR-Generator. Dieser nutzt die entsprechenden TPM-Tools, um den aktuellen TPM-Quote zu erhalten und liest die DML über das Device aus.

Die Architektur der Verifikation kann in zwei Module unterteilt werden, die verschiedene Eigenschaften verifizieren. Das erste Modul verifiziert die Integrität der enthaltenen DML. Dies wird durch das Nachberechnen der TPM-Quote erreicht. Hierzu wurde der Mechanismus der *extend*- und *quote*-Befehle nachimplementiert. Dabei wird jeder errechnete Wert mit dem darauffolgenden Wert aus der DML verkettet und gehasht. Anschließend wird eine TPM-Quote ausgeführt und überprüft, ob die berechnete Quote mit dem SSR-gemeldeten übereinstimmt. Ist das nicht der Fall wird der nächste DML-Eintrag genommen bis keine Einträge mehr vorhanden sind. Wird eine passende Quote gefunden, obwohl noch weitere DML Einträge vorhanden sind, werden die übrigen ignoriert und nicht weiter verifiziert. Dieses Vorgehen ist notwendig, da die TPM-Quote im DKM asynchron ausgeführt wird. Daher kann es sein, dass Einträge in der DML existieren, die noch nicht gegen den TPM gesichert sind. Diese Messwerte müssen dann als noch nicht vertrauenswürdig angesehen werden, und können somit erst in einer nachfolgenden Verifikation betrachtet werden.

Wurde die Integrität der DML erfolgreich bestätigt, wird die DML von dem zweiten Modul weiter verarbeitet. Dieses Modul vergleicht die eigentlichen, in der DML enthaltenen, Messwerte mit den Referenzwerten. Um die korrekten Referenzwerte in der Referenzwertdatenbank zu finden, benötigt die Verifikation zusätzliche Informationen vom zu überwachenden System, wie z.B. eine eindeutige Kennung (z.B. IP-Adresse), Patch-Level des Systems und weitere Metadaten.

Das Modul vergleicht dann die Messewerte, die in der DML vorhanden sind, mit den dazugehörigen Referenzwerten. Stimmen die beiden Werte nicht überein, so muss das System als kompromittiert angesehen werden. Der Vergleich wird dabei von den jeweiligen Guidelines durchgeführt. Dazu ist in der DML zu jedem Messwert auch die entsprechende Guideline hinterlegt.

Durch den Guideline-Ansatz ist es nicht nötig, dass zu jedem Ergebnis ein eindeutiger Referenzwert vorhanden ist. Es ist möglich, dass die Guideline im DKM einen bestimmten Wert ermitteln soll, der dann innerhalb der Guideline der Verifikation nur in einem bestimmten Bereich liegen soll oder einem bestimmten Wert entsprechen soll.

Nach dem Verifizieren der gesamten DML wird ein Bericht erstellt, der in der Verifikationsdatenbank abgelegt wird. Die Verifikation kann als erfolgreich angesehen werden, wenn sowohl die Integrität der DML an sich festgestellt wurde und jeder einzelne Messwert den erwarteten Referenzwerten entspricht. Ist auch nur ein Wert nicht korrekt, ist das Ergebnis der Verifikation negativ zu bewerten und das System wird ab diesem Zeitpunkt als kompromittiert angesehen.

3.5 Guideline

Die in den vorhergehenden Kapiteln beschriebenen Architektur-Frameworks benötigen zusätzliche Geschäftslogik, um die konkreten Operationen auszuführen. Diese Logik wird in dieser Arbeit als Guideline eingeführt. Dabei ermöglichen Guidelines ein sehr dynamisches Verhalten des Systems, da sie einfach durch Implementierung einer definierten Schnittstellenfunktion erweitert oder ergänzt werden können. Guidelines werden immer im Set von drei implementiert, eine für jede Hauptkomponente. Die Implementation selbst ist dann auf die jeweilige Aufgabe spezialisiert. Da Guidelines jederzeit hinzugefügt oder wieder entfernt werden können, dürfen sie keine Funktionen verwenden, die von anderen Guidelines abhängig sind. Wenn mehrere Guidelines ein und dieselbe Funktion verwenden oder implementieren, sollte diese in das Framework aufgenommen werden und dann entsprechend referenziert werden.

Alle drei Implementierungen der Guideline müssen sich auf das gleiche Format der Messwerte einigen. Des Weiteren muss ein systemweiter eindeutiger Guideline-Name gewählt werden, damit die Verifikation die entsprechenden gültigen Referenzwerte ermitteln kann.

Eine DKM-Guideline-Implementierung muss mindestens Messwerte für ein Messziel erzeugen können. Das Messziel kann dabei ein Userspace-Prozess, Kernelspace-Modul oder das Abbild des Kernels sein. Auch muss eine öffentliche Funktion zur Verfügung gestellt werden, die mit der Schnittstellendefinition übereinstimmt und vom DKM definiert wurde. Diese Funktion enthält eine Datenstruktur, die das zu bearbeitende Messziel beschreibt. Ist die Guideline so implementiert, dass sie mehr als nur ein Messziel bearbeiten kann, muss sie eine eigene Unterscheidung der Messziele implementieren, damit sie die Messungen korrekt ausführen kann. Da der Linux-Kernel und die Kernel-Module statisch sind, müssen die Guidelines in der aktuellen Implementation direkt in den DKM-Code mit aufgenommen werden und dem Guideline-Auswahl-Algorithmus bekannt gemacht werden. Das dynamische Hinzufügen und Entfernen von Guidelines ist für eine spätere Implementierung angedacht, aber aktuell nicht Teil der Beispielumsetzung.

Die Implementierung einer Guideline für RVG muss Referenzwerte generieren, die von der Verifikation verwendet werden können. Die Implementierung für RVG erfolgt als Python-Modul und muss eine öffentliche Funktion zur Verfügung stellen, die der definierten Schnittstelle von RVG entspricht. Diese Funktion erhält vom Framework eine ELF-Datei, die von der Guideline verarbeitet werden muss. Die Referenzwerte können dabei von beliebiger Art sein, solange alle drei Implementierungen diese verarbeiten, erzeugen und interpretieren können. Hierbei können es Referenzwerte sein, die direkt verglichen werden können oder weitere Berechnungen erfordern oder es handelt sich um bestimmte Voraussetzungen, die erfüllt sein müssen. Damit RVG eine Guideline verwenden kann, muss diese dem RVG bekannt sein. Dies kann dadurch erreicht werden, indem über die Konsolen API die Guideline registriert wird.

Die dritte benötigte Guideline-Implementierung wird von der Verifikation benötigt. Diese muss in der Lage sein die Messergebnisse, die vom DKM exportiert werden, zu erhalten und diese mit den Werten aus RVG zu vergleichen. Beide Werte werden der Guideline-Implementierung über die Schnittstellenfunktion zur Verfügung gestellt. Für einige Ergebnisse kann es erforderlich sein, dass die Verifikations-Guideline weitere Berechnungen durchführen muss, ehe der eigentliche Vergleich erfolgen kann. Dies ist z.B. erforderlich, wenn Speicher-Offsets berücksichtigt werden müssen. Für solche Ergebnisse erzeugt RVG die für die Berechnung erforderlichen Basiswerte und die DKM-Ergebnisse enthalten zusätzliche Einträge mit den zum Zeitpunkt der Messung vorhandenen Offsets. Die Verifikations-Guideline nimmt dann den Basiswert, kombiniert diesen mit den empfangenen Offsets und vergleicht das Ergebnis mit dem Wert aus dem SSR. Die aktuelle Implementierung erfordert eine Guideline, die in der Verifikation integriert und so dem Guideline-Auswahl-Algorithmus bekannt ist. Wie schon beim DKM ist hier ein dynamischer Ansatz angedacht, der aber in der aktuellen Phase nicht umgesetzt wird.

4 Beispielumsetzung einer Guideline

Im Folgenden wird die Implementation einer Guideline anhand eines Beispiels erklärt. Es werden alle drei benötigten Guidelines und die notwendigen Operationen detaillierter erläutert. Ein Grundwissen des Linux-Speichermanagements ist für das genauere Nachvollziehen von Vorteil. Hierzu sei auf das Kapitel 2.2 verwiesen.

Zunächst wird die Implementation der DKM Guideline erklärt, danach die Guideline für die Referenzwert-Generierung und zum Schluss die der Verifikation. Die hier vorgestellte Guideline misst ausschließlich Userspace-Prozesse.

4.1 DKM-Implementierung

Sobald ein Messziel ausgewählt wurde, liefert das DKM eine Datenstruktur, die einen Verweis auf die Kernelstruktur enthält, in der sich das aktuelle Ziel befindet. Die Datenstruktur enthält alle Informationen, die zur Identifikation und Verarbeitung des Ziels erforderlich sind. Die Daten werden an die Guideline-Implementierung, mit einer leeren Liste, weitergeleitet. Die Guideline muss die leere Liste befüllen.

Das Ziel der Beispiel-Guideline ist der Zugriff auf den Speicher eines Userspace-Prozesses, das Auslesen seiner Speichersegmente, die ausführbaren Quellcode enthalten, und die Be-

rechnung einer Hash-Prüfsumme jedes Segments. Der Hash ist eines der Ergebnisse dieser Guideline. Des Weiteren liest die Guideline die Zugriffsrechte des jeweiligen Speichersegmentes aus und überprüft, ob es andere Seitentabelleneinträge (PTE) mit unterschiedlichen Zugriffsrechten gibt. Dies ist laut Definition zwar möglich, darf allerdings nicht vorkommen. Beides erzeugt wieder ein Ergebnis, das in die Liste eingetragen wird. Als Resultat gibt die Guideline eine Baumstruktur für jedes Speichersegment zurück, das ausführbaren Code enthält.

Um die betreffenden Speichersegmente zu identifizieren und darauf zuzugreifen, muss die Guideline die Speicherverwaltungsstruktur aus der Taskstruktur extrahieren und über die so erhaltene Liste der darin enthaltenen VM-Areas iterieren. Jede VM-Area repräsentiert ein Speichersegment, das von dem Prozess genutzt wird. Dabei wird nicht jedes Speichersegment vom Prozess selbst befüllt. Der Kernel ordnet die Speichersegmente von *Dynamic Link Libraries* dem virtuellen Speicher des Prozesses zu. Somit enthält die Liste der VM-Areas auch alle Speichersegmente der Bibliotheken. Da auch diese Segmente gemessen werden müssen, wird über die gesamte VM-Area-Liste iteriert.

Um die zu verarbeitenden Segmente zu identifizieren, prüft die Guideline die Flags auf Lese- und Ausführungsrechte. Zusätzlich wird geprüft, ob das Shared-Flag (geteiltes Speichersegment) nicht gesetzt ist, da nur Segmente gelesen werden müssen, die für den Prozess privat sind. Entsprechen die Flags den Anforderungen, werden diese extrahiert und gespeichert. Da die Flags plattformunabhängig sind, kann RVG später eine Bit-Maske erzeugen. Diese kann dann aus dem DKM heraus auf die Flags angewendet werden, um zu prüfen, ob die korrekten Flags gesetzt sind. Daher ist in diesem Schritt keine weitere Verarbeitung nötig und das Ergebnis kann direkt zu der Ergebnisliste hinzugefügt werden.

Für jedes ermittelte Segment verwendet die Guideline die Start- und Endadresse aus der VM-Area, um die Anzahl der Seiten, die in dem Segment vorhanden sind, zu ermitteln. Für jede dieser Seiten muss die Guideline sicherstellen, dass diese auch im Speicher geladen sind. Dies wird dadurch erreicht, indem dem Kernel signalisiert wird, dass die Guideline auf diese Seiten zugreifen möchte. Nachdem die Seite in den Speicher geladen wurde, wird diese ausgelesen und zu dem inkrementellen Hash-Algorithmus hinzugefügt. Der Algorithmus kann zur Kompilierungszeit definiert werden. Standardmäßig ist SHA-1 ausgewählt, aber es kann jeder Algorithmus verwendet werden, der von der Kernel-API unterstützt wird.

Zusätzlich liest die Guideline die zu der Seite gehörende PTE aus und testet, ob die korrekten Zugriffs-Flags gesetzt sind. Ist dies nicht der Fall wird ein Ergebniszähler erhöht und der Ergebnisliste hinzugefügt. Sind alle Seiten für ein Segment bearbeitet, beendet die Guideline die Hash-Operation und fügt die Ergebnisse der Ergebnisliste hinzu. Dies beendet die Messung auf der DKM-Seite.

4.2 RVG-Implementierung

RVG arbeitet mit ELF-Dateien des Messzieles und aller dazugehörigen Bibliotheken, die dynamisch gelinkt sind. Um Referenzwerte für die VM-Area-Flags zu generieren, erstellt die Guideline eine Bit-Maske, die mit der tatsächlich erwarteten übereinstimmt. Dieses Verfahren kann deshalb angewandt werden, da die VM-Area-Flags plattformunabhängig dargestellt werden. Im Beispiel der Speichersegmente, die von der Beispielumsetzung verarbeitet werden,

ergibt sich der Referenzwert 0x5, da bei diesen nur die Lese- und Ausführungs-Flags gesetzt sein dürfen.

Die Generierung des Referenz-Hashes für die Speichersegmentinhalte ist dabei komplizierter. Zunächst muss das betreffende Segment in der ELF-Datei lokalisiert und in einem Puffer extrahiert werden. Hierzu wird die ELF-Datei mit entsprechenden Bibliotheksfunktionen geladen und eine Datenstruktur erzeugt. Diese ermöglicht den Zugriff auf die einzelnen Teile und Informationen, die in der ELF-Datei gespeichert sind. Wurde das entsprechende Speichersegment lokalisiert, muss es der System-Seitengröße (Page Alignment) angepasst werden. Das bedeutet, an dem Segment werden solange 0-Bytes angehangen, bis eine Größe erreicht ist, die dem Vielfachen der Seitengröße des Zielsystems entspricht. Diese Prozedur ist notwendig, damit die kleinste Speichereinheit, mit der der Kernel arbeiten kann, eine Seite ist. Der Kernel allokiert so viele Seiten, dass das Speichersegment in diese passt und die übriggebliebenen Bytes mit 0x0 aufgefüllt werden. Diese Prozedur wird von der Guideline ebenfalls durchgeführt, damit korrekte Referenzwerte erzeugt werden können.

Nachdem dieser Schritt erfolgt ist, berechnet die Guideline verschiedene Hashwerte mit unterschiedlichen Hash-Algorithmen der jeweiligen Inhalte und speichert diese in der Referenzwertdatenbank mit dem dazugehörigen angewandten Hash-Algorithmus. Später kann die Verifikation dann anhand des verwendeten Algorithmus im DKM die entsprechenden Referenzwerte ermitteln. Für die PTE-Flags werden keine weiteren Referenzwerte erzeugt, da die Guideline-Implementierung davon ausgeht, dass dieser Wert immer gleich ist, um als gültig angesehen zu werden. Jeder andere Wert bedeutet ein kompromittiertes System.

4.3 Verifikationsimplementierung

Die Implementierung der Guideline auf der Verifikationsseite nimmt die Messergebnisse, die vom DKM berichtet wurden, und vergleicht diese mit den Referenzwerten aus RVG. In diesem Fall ist dieser Prozess simpel, da keine weiteren komplizierten Berechnungen erfolgen müssen.

Für jedes Messziel, das vom DKM gemessen wurde, erhält die Verifikation eine Liste von Ergebnissen. Diese Ergebnisliste enthält alle gemessenen Werte eines Userspace-Prozesses. Die Verifikations-Guideline iteriert durch alle Einträge in der Liste und führt die erforderlichen Validierungen durch. Für jeden Eintrag wird ein Report erstellt, der den Verifikationsstatus enthält, und in der Verifikationsdatenbank abgelegt wird. Diese Reports können später von einem Tool verwendet werden, das einen detaillierten Verifikationsbericht erstellt.

Die PTE-Flag-Prüfung erfolgt mit dem Vergleich des Wertes aus dem DKM und Null, denn so wurde dieser Wert innerhalb der drei Guidelines definiert. Repräsentiert der Referenzwert eine andere Zahl, so ist dieser nicht valide. Für die anderen Ergebnisse in der Liste ermittelt die Guideline den entsprechenden Referenzwert und vergleicht diesen. Die Ergebnis- und Referenz-Hashes werden bitweise verglichen. Stimmen diese überein, gelten sie als valide. Andernfalls gelten sie als invalide. Um die VM-Area-Flags zu testen, werden die vom DKM übermittelten Flags mit denen aus RVG über bitweise Operationen verglichen. Zunächst wird ein bitweises UND sowie eine Bit-Maske berechnet, die alle vier Bits von Interesse enthält. Diese entsprechen den Lese-, Schreib-, Ausführungs- und Shared-Flags. Danach wird das Ergebnis der Operation mit dem Referenzwert verglichen. Entspricht der Wert der Bit-Maske

0xF, so sind alle erlaubten Flags gesetzt. In diesem Fall kann das Ergebnis als valide angesehen werden.

Wird kein Referenzwert für ein Ergebnis gefunden, so muss dieses als invalide angesehen werden. Der Hintergrund ist, dass es keine unerwarteten Ergebnisse auf dem Zielsystem geben darf, weil das System komplett definiert ist. Gleiches gilt auch für Referenzwerte die keinem Messergebnis zugeordnet werden können, denn in diesem Fall fehlt ein Prozess oder Modul auf dem Zielsystem. Die einzige Ausnahme zu diesem Fall stellt die Berechnung der Hash-Werte dar, denn diese werden für mehrere Algorithmen durchgeführt und in der Referenzwertdatenbank abgelegt. Die Guideline wird beendet, wenn alle übergebenen Ergebnisse bearbeitet wurden.

5 Zusammenfassung

In dieser Arbeit wurde eine flexible und erweiterbare Architektur vorgestellt, die eine sichere Messung und Verifizierung von dynamischen Laufzeitinformationen für auf Linux-basierende Betriebssysteme ermöglicht und Hardware-basierte Sicherheitstechnologien verwendet. Die vorgestellte Lösung ermöglicht die Attestierung von Messwerten durch das vorgestellte Modell der Guidelines. Die vorgestellten Guidelines stellen ein spezielles Tripel von Regeln dar, um die Attestierung eines zu überwachendem Systems auf Basis von generierten Referenzwerten auszuführen. Weiterhin wurde anhand der Messung und Attestierung von Userspace-Prozessen eine Beispielumsetzung dieser Guideline-Tripel und der nötigen Schritte vorgestellt. In zukünftigen Arbeiten wollen wir das Guideline-Konzept weiterführen und eine konkrete Umsetzung für Loadable Kernel Module (LKM) und den Linux-Kernel implementieren.

Literatur

[ARKK12] Arun K. Kanuparthi: *Architecture Support for Dynamic Integrity Checking*. IEEE Transactions on Information Forensics and Security, Vol. 7, No. 1, February 2012. p. 321-332, DOI: 10.1109/TIFS.2011.2166960

[TCG08] Trusted Computing Group: *Trusted Platform Module (TPM) Summary*. https://trustedcomputinggroup.org/trusted-platform-module-tpm-summary/, Veröffentlichungsdatum: 01. April 2008

[WADC15] Will Arthur, David Challener: *A Practical Guide to TPM 2.0*. 2015, P13, EAN: 9781430265849

[CBPH12] C. Bormann, P. Hoffman: *Concise Binary Object Representation (CBOR)*. Internet Engineering Task Force (IETF), RFC 7049, IETF in October 2013

[TCG15] Trusted Computing Group: *Trusted Platform Module (TPM) 2.0. a brief introduction*. https://www.trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-A-Brief-Introduction.pdf, 2015

[REIN17] Andre Rein: *DRIVE: Dynamic Runtime Integrity Verification and Evaluation*. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17). ACM, New York, NY, USA, 728-742. DOI: https://doi.org/10.1145/3052973.3052975