Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Ausarbeitung Anwendungen II SoSe 2010/2011

Theodor Nolte

Implementierungsframework für die Schadsoftwareerkennung auf Android

Related Work

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die Zielvorgabe des SKIMS-Projekts [10] ist die Konzeption, Entwicklung und Analyse einer schichtenübergreifenden kooperativen Sicherheits-Umgebung für mobile Geräte. Hierbei stellt das Erkennen von Schadsoftware einen wichtigen Grundstock dar, um darauffolgend Gegenmaßnahmen ergreifen zu können. Für das Auffinden von Schadsoftware kommen grundsätzlich unterschiedliche Mechanismen in Frage. Ein Vorgehen beispielweise ist die Entropie-Analyse von Datenströmen im Netzwerk. Dieser Ansatz wird von unserem Kommilitonen Benjamin Jochheim untersucht und weiterentwickelt. [2] Jedoch sind auch andere Ansätze möglich, etwa das Überprüfen von Dateien anhand von Virensignaturen, wie es von klassischen Antivirenprogrammen angewendet wird.

Das Szenario mobiler Endgeräte unterscheidet sich signifikant von konventionellen Endbenuter-Systemen:

- Die Ressourcen-Verfügbarkeit sowie die Rechenkapazitäten von mobilen Endgeräten sind beschränkt.
- Die Tatsache, dass mobile Endgeräte als ein *Single Spot* privater Daten verwendet werden, macht sie interessant für Angriffe.
- Unterschiedliche Funk-Schnittstellen ermöglichen neue Angriffsformen: Der *„Äther“* ist ein inhärent allen zugängliches Übertragungsmedium, zugleich sind jedoch Angriffe über dieses Medium lokal begrenzt.

Methoden zur Entdeckung von Schadsoftware haben diesen Besonderheiten Rechnung zu tragen.

Hier hat man es also im Gegensatz zur klassischen PC-Landschaft mit veränderten Rahmenbedingungen zu tun, so dass ein flexibles Rahmenwerk für die Schadsoftwareerkennung wünschenswert wäre. Dies würde es einerseits ermöglichen, sowohl bewährte, auf mobile Umgebungen adaptierte Verfahren einzusetzen, als auch neue Ansätze und Verfahren zur Anwendung einzubringen.

1.2 Inhaltlicher Aufbau der Ausarbeitung

In 2.1 beschreibe ich zunächst meinen Ansatz zur Architektur des Frameworks zur Schadsoftwareerkennung, anschließend wenden wir uns in 2.2 einem konkreten Analyseverfahren zu. In Abschnitt 3.1 folgend wir einigen Grundsätzlichen Überlegungen. In Abschnitt 3.2 wird AASandbox vorgestellt, und in Abschnitt 3.3 betrachten wir das KBTA-Verfahren. In 4.1 die Betrachtungen zusammengefasst und zuletzt wird in 4.2 ein Ausblick gegeben.

2 Framework zur Schadsoftwareerkennung

Im Wintersemester 2010/2011 lautete mein Thema „Mobile Honeypot“ [9]. Jedoch hat es sich herausgestellt, dass dieses Thema innerhalb des DFN-CERT (ich arbeite dort) nicht von mir weiterverfolgt wird. So hat nun mein neues Thema den Titel: „Implementierungsframework für die Schadsoftwareerkennung auf Android“. Obgleich ein neues Thema, gibt es doch thematische Überschneidungen, so dass ich nicht „bei null“ anfangen musste, und erworbenes Wissen weiter verwenden kann.

2.1 Architektur

Aufgabenstellung meines neuen Themas ist es, ein Framework für die Schadsoftwareerkennung zu entwerfen, für die Android-Umgebung zu implementieren und schließlich zu analysieren und zu bewerten. Hierbei ist es wichtig, dass die einleitend genannten besonderen Rahmenbedingungen einer mobilen Plattform berücksichtigt werden. Die bedeutendsten Einschränkungen hierbei sind die begrenzte Verfügbarkeit von Rechenleistung und die Tatsache, dass der Betrieb des Frameworks den Prozessor des Gerätes nicht fortwährend auslasten kann, weil sonst die verfügbare Energie des Akkus allzuschnell verbraucht wäre und somit der der eigentliche Gebrauchszweck des Gerätes in Frage gestellt wäre.

Eine weitere Anforderung ist es, einen möglichst generischen Entwurf zu entwickeln, der auch auf andere Geräteplatformen adaptiert werden kann. Dies stellt eine nicht zu unterschätzende Herausforderung dar, da sich die derzeitigen Betriebssysteme für mobile Geräte signifikant voneinander unterscheiden [4]. Trotzdem soll durch die Implementierung für das Android-Betriebssystem ein „Proof-of-Concept“ dargelegt werden.

Mein Entwurf ist auch in der Hinsicht generisch, als dass er nicht auf das eigentliche Verfahren zum Erkennen von Schadsoftware festgelegt ist. So können mehrere Analyseverfahren (auch gleichzeitig) verwendet werden, welche die Daten aus unterschiedlichen oder den selben Quellen herstammend untersuchen, indem die einzelnen Komponenten (Anlayzer, SourceController, Source) lose gekoppelt sind. Dies ermöglicht eine Flexibilität, die es erlaubt neuartige zukünftige Analyseverfahren in das Framework einzubinden. Abbildung 1 zeigt das Klassendiagramm des Frameworks. Der Broker stellt der Anwendungsschicht ein Mapping zur Verfügung, so dass sie zunächst herausfinden kann, welche Analyseverfahren verfügbar sind. Anschließend kann sie dem Observer-Entwurfsmuster folgend die Bewertungen von einem bestimmten Analyzer abbonieren. Das Starten des Analyzers und der Mechanismus zur Bereitstellung der von ihm benötigten Daten (SourceController und Source) erfolgt automatisch. Auch das Stoppen bzw. Pausieren erfolgt selbsttätig, mit der Absicht, unnötigen (Rechen-) Ressourcenverbrauch zu vermeiden.

Abbildung 1: Klassendiagramm des Frameworks zur Erkennung von Schadsoftware

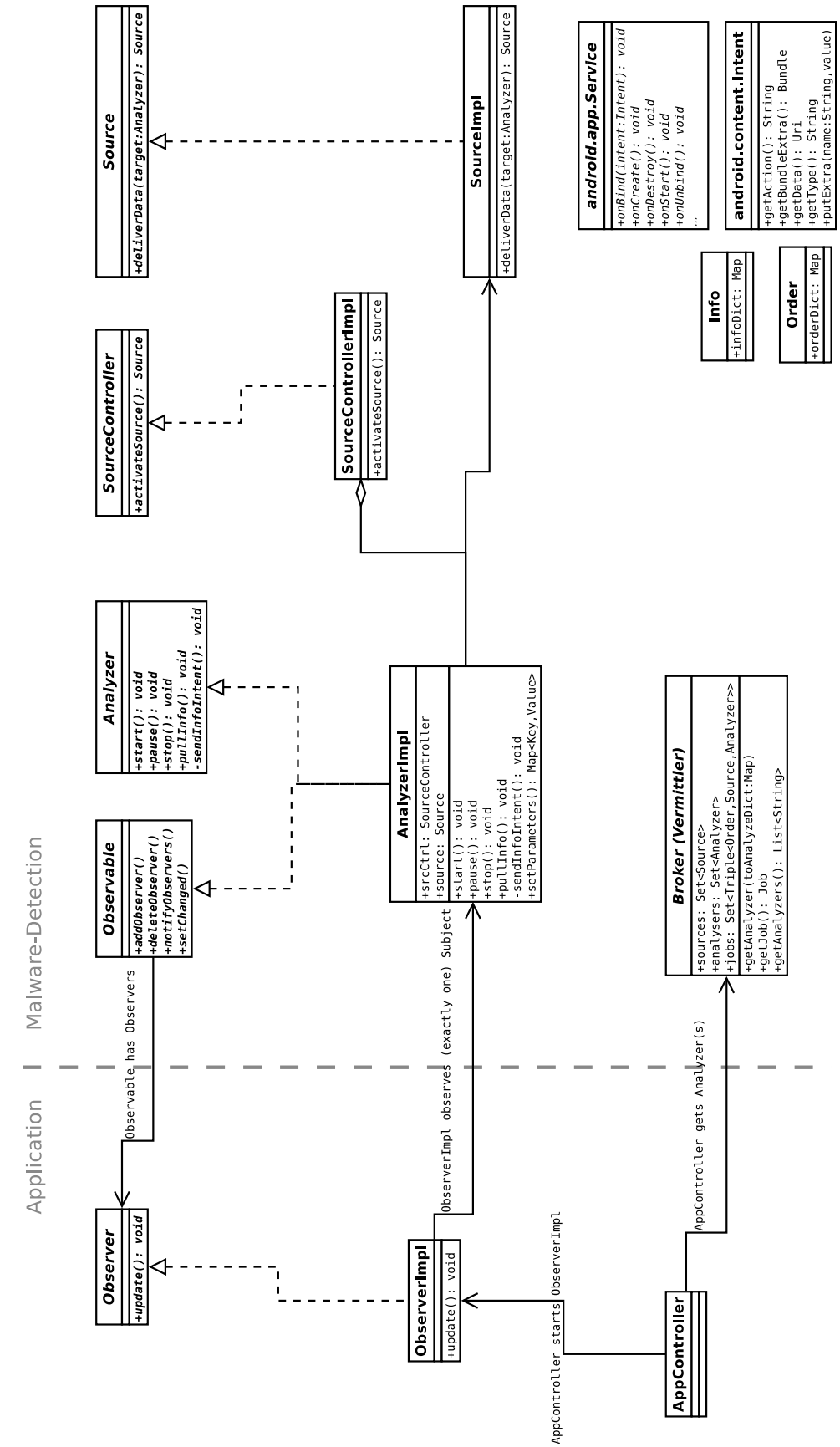
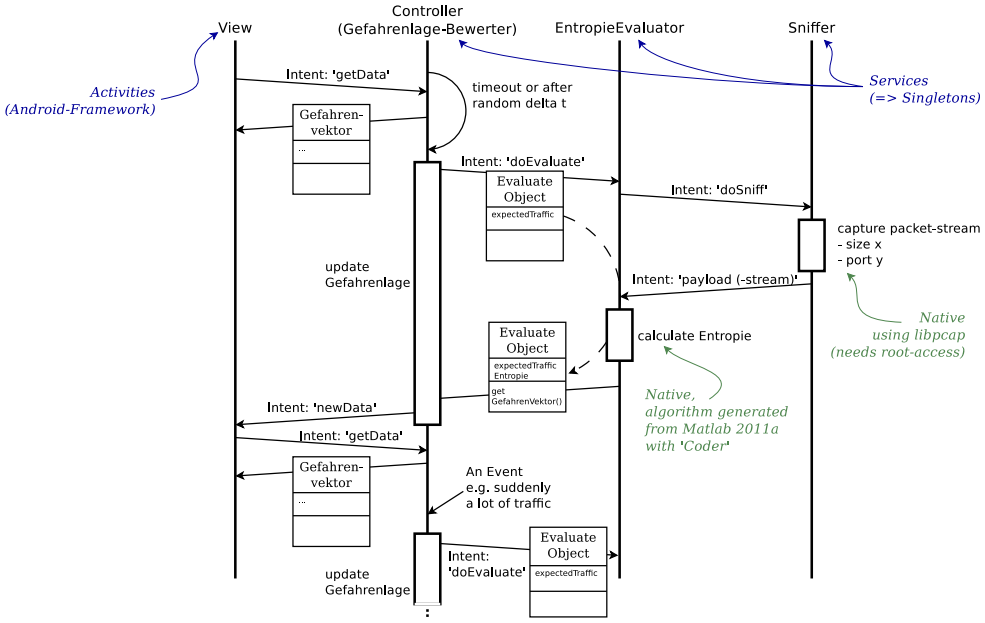


Abbildung 2: Sequenzdiagramms des Ablaufs der Untersuchung auf Schadsoftware mittels Entropieanalyse



2.2 Entropie-Analyse von Netzwerktraffic

Klassische Virenscanner verwenden eine Signatur-Datenbank, um Schadcode zu entdecken. Um effektiv zu bleiben, muss eine solche Signaturdatenbank oft aktualisiert werden. Zwangsläufig ergibt sich hieraus eine zeitliche Lücke für neue, bisher unbekannte Schadsoftware bis ein Signatur-Update greift. So wäre es wünschenswert, ein Verfahren zu Verfügung zu haben, dass in der Lage ist, auch Zero-day Exploits, d.h. bisher unbekannte Schadsoftware zu entdecken.

Genau dieses Ziel verfolgt der „Signaturfreie Ansatz zur Schadsoftwareerkennung durch Entropieanalyse von Netzwerkströmen“ [2] in Echtzeit. Hierbei werden die übertragenen Daten mittels eines statistischen Verfahrens analysiert, dass auf der Shannon Entropy [7] basiert.

Dieses Analyseverfahren binde ich nun in das Framework ein. Dabei stellte es eine Herausforderung dar, die libpcap [8] auf der Android-Umgebung zur Anwendung zu bringen: Einerseits gelingt es mir bisher nur, Binaries die Netzwerkpakete mitschneiden für den ARM zu cross-kompilieren, ohne native Filter einzusetzen. Andererseits Bedarf es eines Kniffes, um aus der Sandbox, in der eine Android-App ausgeführt wird, auszubrechen und das native Binary mit Root-Rechten zum Mitschneiden der Pakete auszuführen. Abbildung 2 zeigt den Ablauf innerhalb des Frameworks zur Erkennung von Schadsoftware mittels der Entropieanalyse.

3 Related Work

3.1 Grundsätzliche Betrachtungen zur Sicherheit in Mobilen Umgebungen

Jon Oberheide und Farnam Jahanian von der *Electrical Engineering and Computer Science University of Michigan* haben grundsätzliche Überlegungen angestellt, worin die besonderen Herausforderungen bezüglich der Sicherheit in mobilen Systemen liegen.[4]

Sie stellen wichtige Unterschiede mobiler Systeme zu ortsgebundenen Systemen dar. Zum einen sei hier auf die schon in der Einleitung erwähnten beschränkten Ressourcen mobiler Systeme hingewiesen. Dieser Umstand bedeutet, dass selbst einfache signatur-basierte statische Analysen viele Rechenressourcen an sich binden. Als Beispiel wird hier die ClamAV Antivirus Engine für das Nokia N800 Mobil-Telefon herangezogen, welche allein für die Initialisierung 57 Sekunden benötigen würde bei einem Arbeitsspeicherverbrauch von 40 Megabyte.[5] Daher seien traditionelle Ansätze für mobile Umgebungen nicht einfach „1-zu-1“ übertragbar, da sie zuviel Rechen- und Energieressourcen verbrauchen. Als ein vielversprechender Ansatz wird hier aufgeführt, rechenintensive Malware-Erkennung „off-device“ als einen Netzwerkdienst verfügbar auszulagern, was eine transparente Erweiterbarkeit und Skalierbarkeit ermöglicht.[5][3]

Auch die Angriffe selber auf mobile Geräte unterscheiden sich von herkömmlichen Angriffen auf lokale Rechner. Aufgrund der beschränkten Ressourcen und der nicht stetig verfügbaren und wechselnden Internetanbindung sind mobile Geräte nachrangig interessant für traditionelle Bot-Netze, welche vorwiegend dem Spamversand, verteilten Denial-of-Service Angriffen und Phishing-Angriffen dienen. Hierfür sind Host besonders wertvoll, die über hohen Durchsatz, geringer Latenz und stabiler Internet-Verbindung verfügen, was für mobile Smartphones nicht zutrifft. Stattdessen dürften für Angreifer die persönlichen Daten auf den mobilen Geräten in Form von Kontoführungs-Credentials, SMSe, Kalenderdaten, GPS-Tracks und sonstige vertrauliche Informationen von Interesse. Hierbei stellt sich die Frage, ab wann es sich für einen Angreifer vom Aufwand her lohnt, diese Daten durch Malware-Angriffe zu stehlen zu versuchen. Jedoch wird es wohl durchaus wahrscheinlich sein, dass dieser Umstand von Blackhats „ausprobiert“ wird, um zu erfahren, ab wann sich entsprechende Angriffe lohnen.

3.2 AASandbox

Thomas Bläsing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe und Sahin Albayrak haben in [1] eine *Android Application Sandbox (AASandbox)* zur statischen und dynamischen Analyse von Android-Applikationen (Apps) vorgestellt.

Anstatt eine App auf dem Gerät selber zu installieren und somit den eingeschränkten Ressourcen unterworfen zu sein, findet die Analyse auf einem herkömmlichen Rechner oder „in der Cloud“ statt. So kann auf hohe Rechenkapazität zurückgegriffen werden. Dabei wird auf einen Emulator zurückgegriffen, der ein Android-Gerät nachstellt.

Für die Analyse sind zwei Verfahren vorgesehen. Einerseits wird die nicht installierte App, d.h. die .apk-Datei statisch überprüft. Dies bedeutet, dass sie einer klassischen Virensignaturprüfung unterzogen wird. Der Vorteil hierbei ist, dass diese Überprüfung schnell, relativ einfach und automatisch vorgenommen werden kann. Des weiteren wird der Dalvik-VM Bytecode dekompiliert, so dass wieder Java-Code vorliegt. Dieser Java-Code wird gescannt auf Patterns wie dem Verwenden des Java Native Interfaces welches für das dynamische Laden von nativen Libraries genutzt werden kann, welche etwa (bekannte oder bisher unbekannte) Sicherheitslücken im Linux-Kernel ausnutzen können. Ein weiteres Pattern ist der Aufruf `System.getRuntime().exec(...)`, welcher einen nativen Kind-Prozess erzeugt, der nicht mehr an das Android-Framework gebunden ist.[1]

Im zweiten Verfahren wird die zu untersuchende App in dem Standard-Emulator der zum Google Android SDK gehört installiert. Anschließend wird die installierte App gestartet. Dabei ist die AASandbox im Kernel-Space platziert und „entführt“ system calls, um sie zu loggen. Abbildung 3 zeigt die Einzel-Schritte bei einem `read()` System-Aufruf auf User-Space Ebene. Abbildung 4 zeigt die Einzel-Schritte bei einem „entführten“ System-Aufruf auf User-Space Ebene. Diese gleichen den Schritten aus Abbildung 3 bis zu dem Punkt, an dem die System-Call Routine `sys_read()` erreicht wird. Hier wird der weitere Verlauf durch eine Policy vorgegeben und kann gegebenenfalls unterbunden werden. Anschließend wird das so erzeugte Logfile aufbereitet (Umwandlung in mathematische Vektoren) und analysiert.

Nun wurden etwa 150 Apps aus dem Android-Market dieser dynamischen Analyse unterzogen. Anschließend wurde auch eine selbstgeschriebene Malware („fork bomb“) ebenfalls der dynamisch analysiert. Bei der Auswertung Log-Dateien konnte ein signifikanter Unterschied der selbstgeschriebenen Malware zu den anderen Apps aufgezeigt werden. Jedoch sagen die Autoren aus, dass sie für eine Verfeinerung des Verfahrens weitere echte Malware benötigen.

[1] Genau diesen Mechanismus verwende ich in der Implementierung meines Frameworks, um nativen Code mit Root-Rechten auszuführen.

Abbildung 3: Schritte zur Ausführung eines `read()` System-Calls auf User-Space Ebene. Jeder Pfeil entspricht einem Instruktionssprung, Quelle: [1]

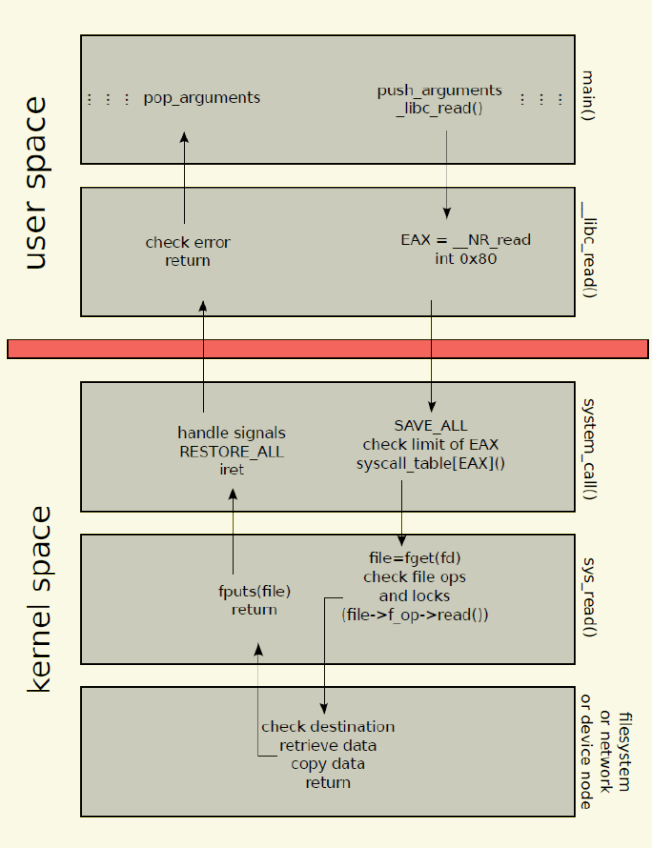


Abbildung 4: Schritte zur Ausführung eines „entführten“ `read()` System-Calls (system call hijacking). Quelle: [1]

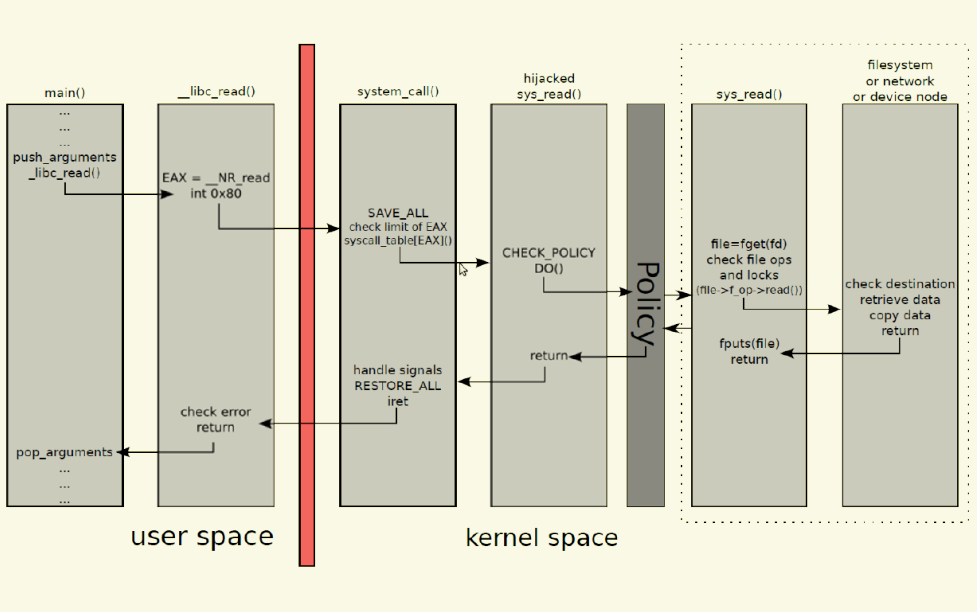
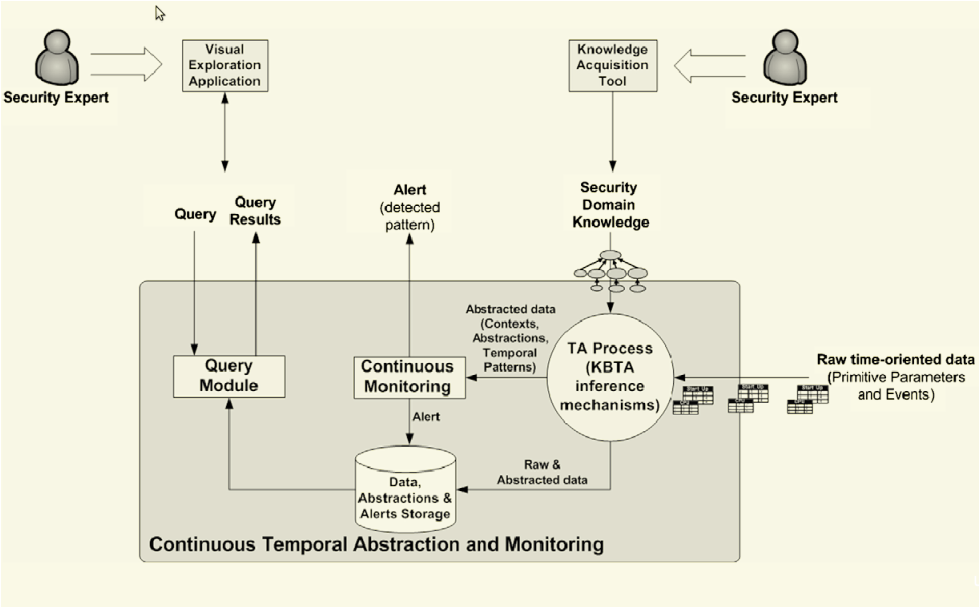


Abbildung 5: Darstellung des KBTA Verfahrens Quelle: [6]



3.3 KBTA

Asaf Shabtai, Uri Kanonov und Yuval Elovici von den Deutsche Telekom Laboratories der Ben-Gurion University, Israel habe in [6] das „knowledg-based temporal abstraction (KBTA)“ Verfahren vorgestellt. Dieses Verfahren repräsentiert eine Ontologie dar, um aus rohen, zeitorientierten „Sicherheitsdaten“ (security data) automatisch abstrakte, zeitorientierte Muster abzuleiten, die „zeitliche Abstraktionen“ („temporal abstractions“) genannt werden. Diese zeitlichen Abstraktionen werden über die Zeit überwacht um bei einem verdächtigen Muster Alarm zu schlagen. Dafür gibt es eine Datenbank mit Mustern zeitlicher Abstraktionen von Malware auf die verglichen wird. Abbildung 5 zeigt das KBTA-Framework auf. Hierbei ist festzustellen, dass das der Aufbau der Datenbank, auf der die Ontologie basiert, von einem Sicherheitsexperten vorzunehmen ist, der über ausreichende Kenntnisse verfügt, die für die Erstellung erforderlich sind.

4 Zusammenfassung und Ausblick

4.1 Zusammenfassung

Die grundsätzlichen Betrachtungen zu Sicherheit mobiler Geräte in [4] haben aufgezeigt, dass und worin die Unterschiede zu normalen PCs liegen. Insbesondere weil einerseits die Absichten möglicher Angriffe und andererseits die Ausführbedingungen aufgrund der eingeschränkten Ressourcen stark von Angriffen auf traditionelle Rechner abweichen, ist mit neuartigen Angriffen zu rechnen, so dass das Erkennen von Zero-Day Exploits einen besonders großen Stellenwert auf mobilen Systemen einnimmt.

In dem Verfahren der AASandbox wurde eine Möglichkeit aufgezeigt, Zero-Day Exploits für Android-Umgebungen zu entdecken. Sie unterscheidet sich von meinem Ansatz in der Form, als das die Analyse nicht auf einem echten Gerät gemacht wird, sondern innerhalb eines Emulators erfolgt. Eine weitere Einschränkung hierbei ist, dass ausschließlich Apps auf Malware untersucht werden. Jedoch gibt es auf einer mobilen Plattform darüber hinaus vielfältige Angriffsmöglichkeiten die mit diesem Verfahren nicht abgedeckt werden. Interessant für mein Framework ist es aber dennoch. So könnte die dynamische Analyse der AASandbox als ein mögliches Analyseverfahren meines Frameworks genutzt werden, indem die zu untersuchende App des echten mobilen Gerätes an eine per Netzwerkdienst abrufbare AASandbox-Instanz übermittelt wird und das Ergebnis der Untersuchung wieder empfangen wird und bei Malware-Befund Grundlage für etwaige Schutzmaßnahmen bildet.

Das KBTA-Verfahren findet auf dem mobilen Gerät selber Anwendung mit vielversprechendem Ergebnis. So hat die Evaluierung des Verfahrens ergeben, dass bei einer geringen durchschnittlichen Prozessorauslastung von ca 3 Prozent eine Erkennungsrate von über 90 Prozent erreicht wurde. Allerdings sind diese wahrlich traumhaften Werte unter „Laborbedingungen“ gemacht worden, und es wäre interessant zu erfahren, wie die Erkennungsrate dieses Verfahrens unter „real-World“ Bedingungen ausfällt. Ich werde versuchen, dieses Verfahren als ein weiteres Analyseverfahren einzusetzten. Jedoch bin ich hierbei auch skeptisch, da es einerseits ein recht kompliziertes Verfahren ist, andererseits die Daten auf die die Ontologie basiert, von einem Experten im Vorraus zu erstellen ist. So bin ich skeptisch, ob wirklich alle Zero-Day Exploits gefunden werden können.

4.2 Ausblick

Bei meinem weiteren Vorgehen werden ich aus den generellen Betrachtungen von [4] den meisten Nutzen ziehen. Insbesondere habe ich das Ziel, Zero-Day Exploits erkennen zu können. Dabei möchte ich darauf achten, dass das Framework nicht zu kompliziert wird (Es gibt nicht umsonst den Ausspruch (sinngemäß): „Komplexität ist der Feind der (Computer-) Sicherheit“). Gleichzeitig wird (hoffentlich) der größte Wert des Frameworks darin bestehen, in einer flexiblen Art und Weise auf unterschiedliche Analyseverfahren zurückgreifen zu können und auch zukünftige Analyseverfahren einbinden zu können. Hierbei bin ich gespannt, was für eine

Erkennungsrate mittels des Entropieverfahrens zu erwarten ist.

Glossar

Arpspoof

Arpspoof ist ein Unix-Kommandozeilentool um den Paketfluss in einem geswitchten LAN durch gefälschte ARP-Pakete künstlich umzuleiten (ARP-Spoofing oder auch ARP Request Poisoning, vgl. RFC1027). Dadurch wird es ermöglicht mit weiteren Tools wie dsniff und fragrouter den Netzwerkverkehr abzuhören. Solch ein Netzwerkeingriff ist in öffentlichen Netzwerken nicht erlaubt und kann mithilfe von arpwatch aufgedeckt werden. Arpspoof ist freie Software. Arpspoof wurde von Robbie Clemons in Form einer sogenannten App für Android unter der GPL stehend portiert und nutzt unter anderem die libpcap. Hierbei hat er sich maßgeblich an die Kriterien für „Clean Code“ gehalten, so dass sich der Code sehr gut zum Nachvollziehen eignet, wie die libpcap auf einem Android-Gerät aus einer App heraus verwendet werden kann.

BusyBox

BusyBox umfasst viele Unix-Dienstprogramme in einer einzelnen ausführbaren Datei. Der Umfang der enthaltenen Unix-Kommandos orientiert sich an der Single Unix Specification (SUS) und dessen, was Anwender eines Linux-Systems an Befehlen erwarten würden. Wegen seiner geringen Größe (ca. 1 MB) findet es Anwendung in Embedded-Linux-Systemen, so auch auf gerooteten Android Geräten. Das 1996 von Bruce Perens geschriebene BusyBox wird auf der Projektseite (`http://www.busybox.net`) als das „Schweizer Taschenmesser für embedded-Linux“ angepriesen. Es ist freie Software und steht unter der GPLv2. Das Android-Grundsystem ist nur sehr rudimentär ausgestattet. Die Installation von BusyBox stellt hier den üblichen Weg dar, das System mit allgemein gebräuchlichen Unix-Kommandos auszustatten.

Dropbear

Dropbear ist eine SSH-Server und -Client Implementierung des Protokolls der Secure Shell Version 2 (SSH2) für POSIX-Plattformen. Es ist aufgrund der Tatsache, möglichst wenig Speicher- und Prozessorressourcen zu verbrauchen, für eingebettete Systeme optimiert und eigenet sich somit für Android Geräte. Die Homepage von Dropbear lautet: `http://matt.ucc.asn.au/dropbear/dropbear.html`. Dopbear ist freie Software und steht unter der MIT-Lizenz. Anders als die Shell-Sitzung über die Android-Debugging-Bridge (adb shell) erhält man mittels Dropbear ein „richtiges“ Terminal, u.a. mit dem Komfort der Autovervollständigung.

GNU Bison und flex

GNU Bison ist ein Parsergenerator, d.h er erzeugt basierend auf einer Beschreibung einer kontextfreien Grammatik ein C-, C++- oder Java-Programm, welches eine Folge von Token parsen kann, die der Grammatik folgt; es erzeugt sogenannte LALR-Parser. Die Beschreibung der kontextfreien Grammatik erfolgt in einem der Backus-Naur-Form

ähnlichen Format. GNU Bison ist freie Software und steht unter der GNU Public Licence (GPL). flex steht für *fast lexical analyzer generator* und ist ein lexikalischer Scanner (auch Tokenizer genannt). Er zerlegt Text (z.B. Quellcode) in Folgen logisch zusammengehörender Token. Auch flex ist freie Software, allerdings nicht unter der GPL, sondern unter der BSD-Lizenz. flex und GNU Bison werden oft gemeinsam genutzt, so auch in der libpcap, um das Parsen und Kompilieren von individuellen Filtern zu bewerkstelligen.

libpcap

libpcap ist die in C geschriebenen Implementierung für Unix-artige Systeme einer API, die Methoden des *packet capturing* bereitstellt, also dem Mitschneiden von Datenpaketen des Netzwerkverkehrs. Es stellt durch Setzten von Filtern bereits auf Kernel-Ebene die Möglichkeit bereit, nicht benötigte Pakete ressourcenschonend zu verwerfen. libpcap wurde im offiziellen Android-Entwicklungszweig bereits für Android angepasst, so dass es (prinzipiell) auch auf Android-Geräten eingesetzt werden kann. Jedoch ist es im Normalfall nicht installiert. libpcap ist freie Software und steht unter der BSD-Lizenz.

tcpdump

tcpdump ist ein Unix-Kommandozeilentool zur Überwachung und Auswertung des Netzwerkverkehrs auf einem System. Es verwendet die libpcap, um auf die Netzwerkpakete zuzugreifen. Wie libpcap ist tcpdump ebenfalls für Android im offiziellen Entwicklungszweig angepasst (und ist ebenfalls im Normalfall nicht installiert). tcpdump ist freie Software und steht unter der BSD-Lizenz.

Literatur

[1] Thomas Bläsing, Aubrey-Derrick Schmidt, Leonid Batyuk, Seyit A. Camtepe, and Sahin Albayrak. An android application sandbox system for suspicious software detection. In *5th International Conference on Malicious and Unwanted Software (Malware 2010)*, Nancy, France, France, 2010.

[2] Benjamin Jochheim, Thomas C. Schmidt, and Matthias Wählisch. A Signature-free approach to malicious code detection by applying entropy analysis to network streams. In *Proc. of the TERENA Networking Conference*, page Poster, Amsterdam, Mai 2011. Terena. Student Poster Award.

[3] Jon Oberheide, Evan Cooke, and Farnam Jahanian. CloudAV: N-Version Antivirus in the Network Cloud. In *Proceedings of the 17th USENIX Security Symposium*, San Jose, CA, July 2008.

[4] Jon Oberheide and Farnam Jahanian. When mobile is harder than fixed (and vice versa): demystifying security challenges in mobile environments. In *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications*, HotMobile '10, pages 43–48, New York, NY, USA, 2010. ACM.

[5] Jon Oberheide, Kaushik Veeraraghavan, Evan Cooke, Jason Flinn, and Farnam Jahanian. Virtualized In-Cloud Security Services for Mobile Devices. In *Workshop on Virtualization in Mobile Computing (MobiVirt '08)*, Breckenridge, Colorado, June 2008.

[6] Asaf Shabtai, Uri Kanonov, and Yuval Elovici. Intrusion detection for mobile devices using the knowledge-based, temporal abstraction method. *Journal of Systems and Software*, 83(8):1524 – 1537, 2010. Performance Evaluation and Optimization of Ubiquitous Computing and Networked Systems.

[7] C. E. Shannon. A mathematical theory of communication. In *Bell System Tech. Journal*, vol. 27, pages 379–423, 623–656, July, October 1948.

[8] Tcpdump/Libpcap. TCPDUMP/LIBPCAP public repository. http://www.tcpdump.org/, 31. August 2011.

[9] Theodor Nolte. Mobile Honeypot. http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master10-11-aw1/nolte, 31. August 2011.

[10] Thomas C. Schmidt. SKIMS - A Cooperative Autonomous Immune System for Mobile Devices. http://www.realmv6.org/skims.html, 31. August 2011.