OS Security

Mobile Sandboxing & Linux Containers

Radboud University, Nijmegen, The Netherlands

Winter 2017/2018

A short recap - 1/2

- Last 2 lectures: Malware & Mandatory Access Control
- Malware evolution from PC to smartphone
- Early days: malware targeting Symbian OS users (`Cabir`, `Pbstealer`)
- Popular smartphone platforms affected
  - Android: first proof-of-concept malware released in 2008
  - iOS: `WireLurker`
  - Windows Phone: `FinSpy Mobile`
  - Blackberry: Trojans using the ‘Backstab’ technique
- Intrusion detection system
  - NIDS, HIDS
  - NIDS: (i) string, (ii) port and (iii) header condition signatures
  - HIDS: signature- and behaviour-based
- Intrusion prevention system
  - NIPS, HIPS
  - (i) signature-based detection, (ii) anomaly-based detection and (iii) protocol state analysis detection

A short recap - 2/2

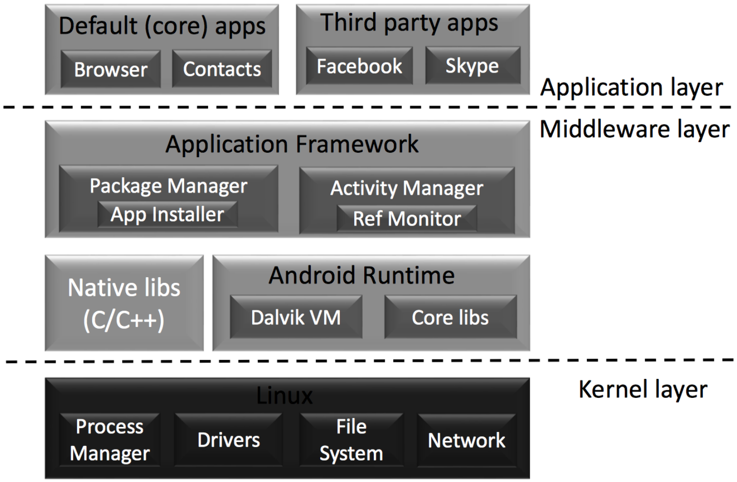
- Mandatory Access Control (MAC)
- A system implements MAC if the protection state can only be modified by trusted administrators via trusted software
- Bell-LaPadula model
- Objects and users are assigned security levels, namely:
  - Top secret
  - Secret
  - Confidential
  - Unclassified
- Biba model, for integrity protection
- Objects and users are assigned integrity levels, namely:
  - Crucial
  - Very important
  - Important

Role of the OS

- A major job of the OS is to enforce **protection**
- Prevent malicious (or buggy) programs from:
  - Allocating too many resources (denial of service)
  - Corrupting or overwriting shared resources (files, shared memory,...)
- Prevent different users, groups, etc. from:
  - Accessing or modifying private state (files, shared memory,...)
  - Killing each other's processes
- Prevent viruses, worms, etc. from exploiting security holes in the OS
  - Overrunning a memory buffer in the kernel can give a non-root process root privileges
- *How does the OS enforce protection boundaries?*
  - Virtualization and compartmentalization
  - This week: mobile sandboxing and linux containers

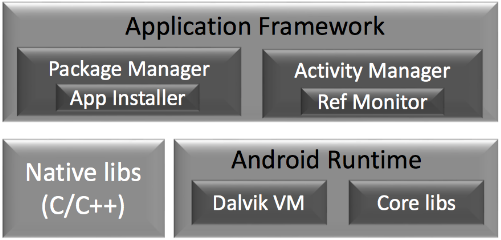
Mobile (Android) Sandboxing
(Part I of this lecture)

Android software stack



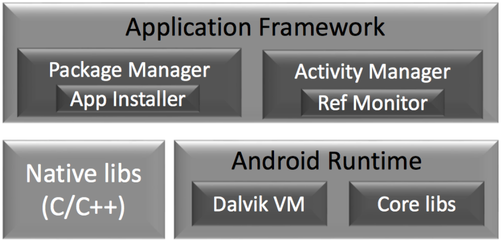
Middleware Layer - Native libraries

- C/C++ system libraries
- Exposed to developers via Android application framework
- Core libraries include: Libc (Bionic), media libraries, Surface manager, 3D libraries, SQLite, SSL



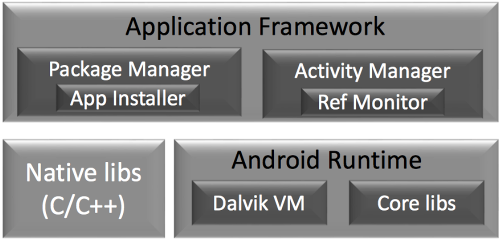
Middleware Layer - Android Runtime

- Dalvik Virtual Machine (DVM)
  - Virtual machine optimized for embedded environments
  - Runs optimized file format ".dex" and Dalvik bytecode generated from Java .class/.jar files at build time
  - Relies on underlying Linux kernel for threading and low-level memory management
- Core Libraries
  - Provide most of the functionality available in the core libraries of Java
  - Provides core APIs of Java



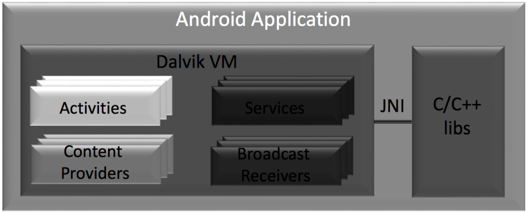
Middleware Layer - Application Framework

- Provides developers API to basic functionalities and services, for e.g. set alarms, access location information, set up a phone call, etc.
- Activity Manager
  - Includes a Reference Monitor, which mediates access requests to critical services, for e.g. SMS, Contacts, Location, based on permissions
  - Responsible for starting applications
- Package Manager
  - Installation of new applications
  - Management of permissions and applications



Application Layer

- Applications are written in Java
- Each app is executed within its own Dalvik VM instance
- Applications also include native code via Java Native Interface (JNI)
- Android applications consist of the following components:
    1. Activities (user interfaces)
    2. Services (background processes)
    3. Broadcast receivers (application mailboxes)
    4. Content providers (SQL-like databases)



Android Security Framework

- Application Isolation
  - VMs and Sandboxing
  - Apps are written in Java (but native code can be used)
- Application Access Control
  - Permission framework at middleware layer
  - Discretionary Access Control (DAC) to file system at Linux kernel level
- Code Integrity
  - System code is signed by Google
  - Applications are signed by developers, using a developer key
- Application Distribution
  - Apps go through a vetting process before they are uploaded to the official app market

Application Isolation via Sandboxing

- Each application is isolated in its own sandbox
  - Apps can access only own resources
  - Access to sensitive resources depends on the application's capabilities, i.e. permissions
- Sandboxing is enforced by Linux
  - Each app is assigned a unique UserID and runs in a separate process (more on that later)
  - Each app has a private data folder

Application Sandboxing

- It specifies which system resources the application is allowed to access
- It limits malicious apps to perform action only within the sandboxed environment
- Two levels of sandboxing:
    - **At process level:**
    - Each app is executed in a dedicated process
    - Access to sensitive resources depends on permissions
    - **At filesystem level:**
    - Each app has its private data directory
    - Only the app can access its own data directory

App Sandboxing: Process Level

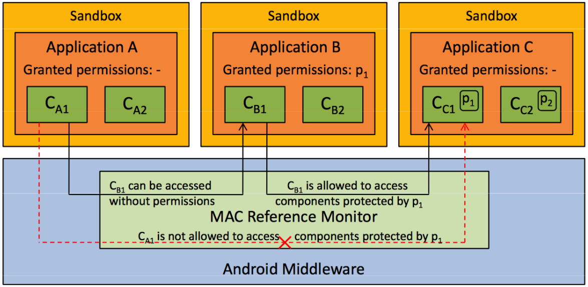
- Android system assigns a unique User ID (UID) to each Android app
- A UID is generated at install-time
- A UID is often called an AppID
- It runs each app as a separate process with its own UID
- Apps run within the sandboxing environment in the kernel

App Sandboxing: Filesystem Level

- Each application is assigned a dedicated data directory
- Only application has permission to read and write to its own directory (in theory!!)
- Sandboxing applies to all applications, including native ones

When can things go wrong?

- Permission escalation attack (see full paper[1])
- An application with less permissions (a non-privileged caller) is not restricted to access components of a more privileged application (a privileged callee)



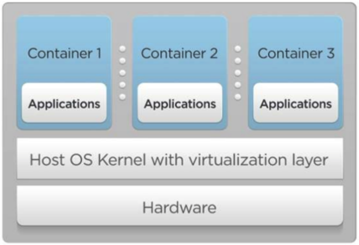
[1] https://www.trust.cased.de/fileadmin/user_upload/Group_TRUST/PubsPDF/DDSW2010_Privilege_Escalation_Attacks_on_Android.pdf

Linux Containers

(Part II of this lecture)

What are Linux Containers (LXC)?

- LXC is an OS-level virtualization method for running multiple isolated Linux systems (containers) on a single control host (LXC host)
- LXC provides a virtual environment that has its own CPU, memory, block I/O, network, etc. space and the resource control mechanism.
- This is facilitated by **namespaces** and **cgroups** features in Linux kernel on LXC host. It is similar to a chroot, but offers much more isolation.
- Benefits: fast provisioning, bare-metal like performance, lightweight



Namespaces

- Namespace restricts what a container can see
- It provides process level isolation of global resources
- Processes have illusion they are the only processes in the system
- There are currently 6 namespaces:
  1. mnt (mount points, filesystems)
  2. pid (processes)
  3. net (network stack, NICs, routing)
  4. ipc (System V IPC)
  5. uts (hostname)
  6. user (UIDs, what uid and gid are visible?)

Mount

- Short name: mnt
- Purpose: different processes have different views of the mount points (“next-gen chroots”)

```
# propagation between host & namespaces
mount --make-(r)shared  / (2-way sharing)
mount --make-(r)private / (no sharing)
mount --make-(r)slave   / (1-way sharing)
```

PID

- Purpose: processes in different namespaces can have the same pid
  - pid inside namespace != pid outside namespace
  - Each container (*) can have its own init (pid 1)
  - Multiple namespaces create multiple nested process trees
  - Migrate containers (*) across hosts keeping the same internal pids

Network

- Short name: net
- Purpose: different network devices, IP addresses, routing tables etc. per namespace

IPC

- IPC Isolation
- Isolating a process by the IPC namespace gives it its own interprocess communication resources, for e.g. System V IPC and POSIX messages
- Objects created in an IPC namespace are visible to all other processes that are members of that namespace, but are not visible to processes in other IPC namespaces
- When an IPC namespace is destroyed (i.e., when the last process that is a member of the namespace terminates), all IPC objects in the namespace are automatically destroyed

Unix Timesharing System

- Short name: UTS
- Purpose: each namespace can have different hostname + domainname
- UTS namespaces provide isolation of two system identifiers: the hostname and the NIS domain name.
- These identifiers are set using `sethostname` and `setdomainname`, and can be retrieved using `uname`, `gethostname`, and `getdomainname`

User

- Purpose: User namespaces isolate security-related identifiers and attributes, in particular, user IDs and group IDs, the root directory, keys, and capabilities
- A process's user and group IDs can be different inside and outside a user namespace
- In addition, a process can have a normal unprivileged user ID outside a user namespace while at the same time having a user ID of 0 inside the namespace; in other words, the process has full privileges for operations inside the user namespace, but is unprivileged for operations outside the namespace

Control groups (cgroups)

- Namespaces provide per process resource isolation solution
- cgroups provide resource management solution (handling groups)
- Set upper bounds (limits) on resources that can be used
- Fair sharing of certain resources
- Examples of cgroup modules:
  - cpu: weighted proportional share of CPU for a group (`mm/memcontrol.c`)
  - cpuset: cores that a group can access (`kernel/cpuset.c`)
  - block io: weighted proportional block IO access (`net/core/netprio_cgroup.c`)
  - memory: max memory limit for a group (`security/device_cgroup.c`)

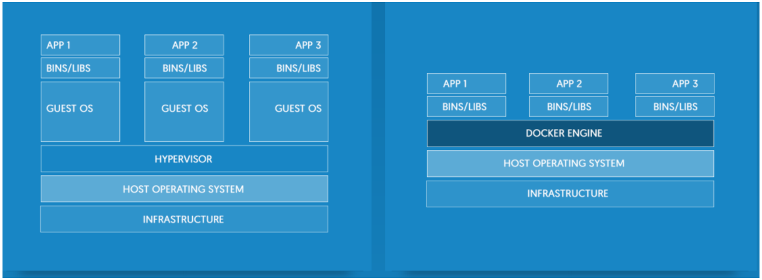
An example - Docker 1/2

- An open-source project that automates the deployment of applications inside software containers by providing an additional layer of abstraction and automation of OS-level virtualization on Linux
- Docker uses resource isolation features of the Linux kernel such as cgroups and kernel namespaces to allow independent“containers” to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines
- Docker includes the libcontainer library as its own way to directly use virtualization facilities provided by the Linux kernel, in addition to using abstracted virtualization interfaces via libvirt, LXC (Linux Containers) and systemd-nspawn

An example - Docker 2/2

- Docker implements a high-level API to provide lightweight containers that run processes in isolation.
- A Docker container, as opposed to a traditional virtual machine, does not require or include a separate OS (comparison on next slide).
- It relies on the kernel's functionality and uses resource isolation (CPU, memory, block I/O, network, etc.) and separate namespaces to isolate the application's view of the operating system.

Traditional virtualization vs Docker



Summary

- How can OS enforce protection boundaries?
- Mobile sandboxing
  - Android software stack (kernel, middleware and application layers)
  - Android security framework
  - Application isolation via sandboxing
    - Process level
    - Filesystem level
  - Example: Permission escalation attack
- Linux containers
  - OS-level virtualization
  - Can be used for running multiple isolated Linux systems
  - Makes use of *namespaces* and *cgroups* features
  - Example: Docker