Black Hat Abu Dhabi 2011

Yet Another Android Rootkit

/protecting/system/is/not/enough/

Research Engineer – Tsukasa Oi

Fourteenforty Research Institute, Inc.

http://www.fourteenforty.jp

Introduction: *root*ing Android

- Gaining Administrative Privileges in Android OS
  - Normally, *root* cannot be used by Apps
  - Gaining *root* Privilege using...
    - Local Exploits (dangerous)
    - Fake Firmware Updates (relatively safe)
- What for?
  - Customization, Overclocking
  - Malicious Use (e.g. DroidDream)
- *root* in Android platform works differently
  - Permission Checks
  - Software-based UID/PID checks

Introduction: Japanese smartphones

- Vendors and Careers want to:
  - Protect Users
  - Protect Career-specific / Vendor-specific Services
  - Ensure Smartphones are not Altered and “Radio Legal”
  - Protect their Business Model ☺
- Answer: “Protect Smartphones”
  - Prevent Firmware Modification
  - Patch Framework and Kernel in order to Secure the device

Agenda

- *root*ing and Android Security
  - Android Internals and Security Model
  - Bypassing Security and Gaining Privileges
- Vendor-Specific Protection
  - Kernel-based Mechanism
- Yet Another Android Rootkit
  - User-Mode Rootkit Bypassing Vendor-Specific Protections
  - Hook User Applications
- So what was wrong?
  - Open source, Closed platform

*root*ing Android is not the end of the story.

***ROOT*ING AND ANDROID SECURITY**

*root*ing is Sometimes Easy

- Five known *root* exploits affecting unmodified version of Android
  - CVE-2010-1185 (exploid)
  - [no CVE number] (rage against the cage)
  - CVE-2011-1149 (psneuter)
  - CVE-2011-1823 (GingerBreak)
  - [no CVE number] (zergRush)
- More of that: Chip/Vendor-specific Vulnerabilities

*root*ing : Vulnerabilities (1)

- Logic Errors in *suid* programs
  - Android Tablet [xxx]: OS command injection

```
$ /system/bin/cmdclient  \
    misc_command         \
    '; COMMAND_IN_ROOT'
```

The attacker can invoke arbitrary command in root privileges.

*root*ing : Vulnerabilities (2)

- Improper User-supplied buffer access
  - Android smartphone [xxx]: Sensor Device Driver

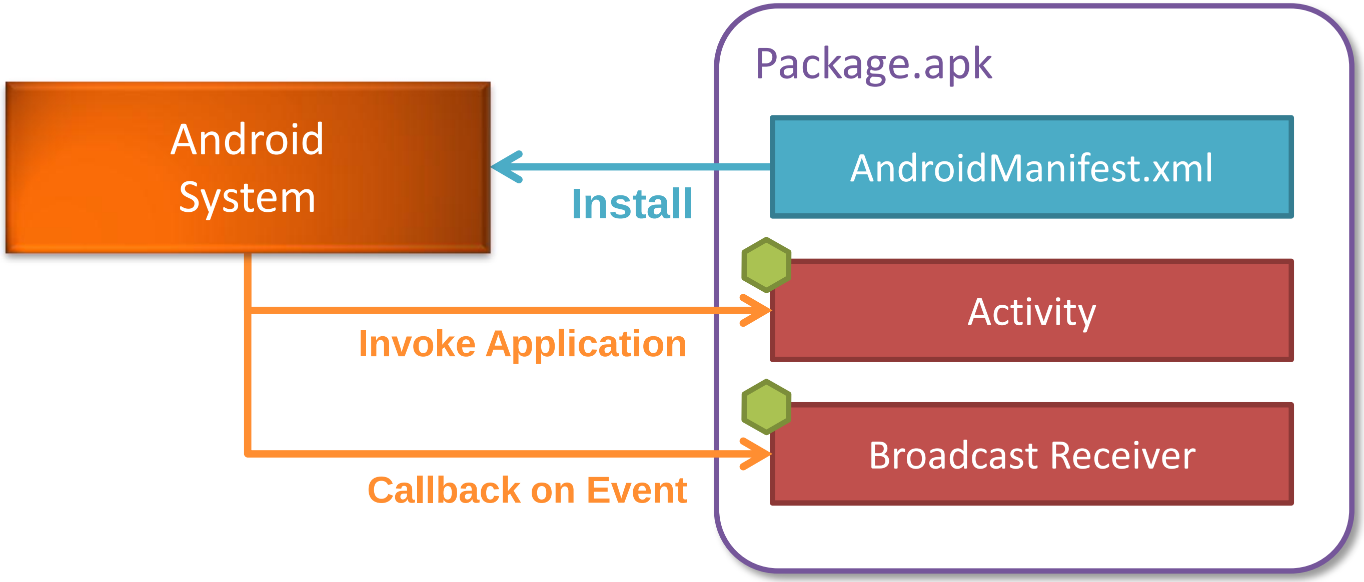
```
static int PROX_read(
    struct file *filp,
    char __user *buf,
    size_t count,
    loff_t *ppos
)
{
    *buf = atomic_read(&sensor_data);
    return 0;
}
```

The attacker write 0 or 7 (according to the sensor data) to arbitrary user memory, bypassing copy-on-write.
Modifying *setuid* function (which affects all processes) can generate root-privilege processes.

*root*ing isn't the end

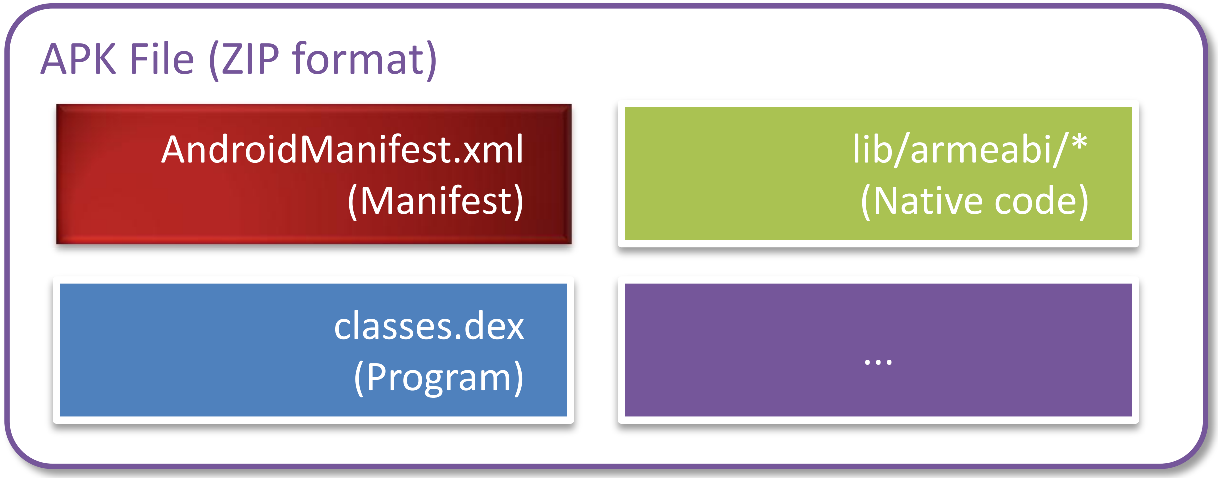
- Gaining Privileges in Android system
  - *root* user in Android system is slightly different
  - The attacker want to take over the whole system
- Vendor-Specific Protection
  - DroidDream won't work properly on some Japanese Android phones
  - /system may be Read-Only
- Is it possible to take over the system in protected smartphones?

Android Internals: App Model



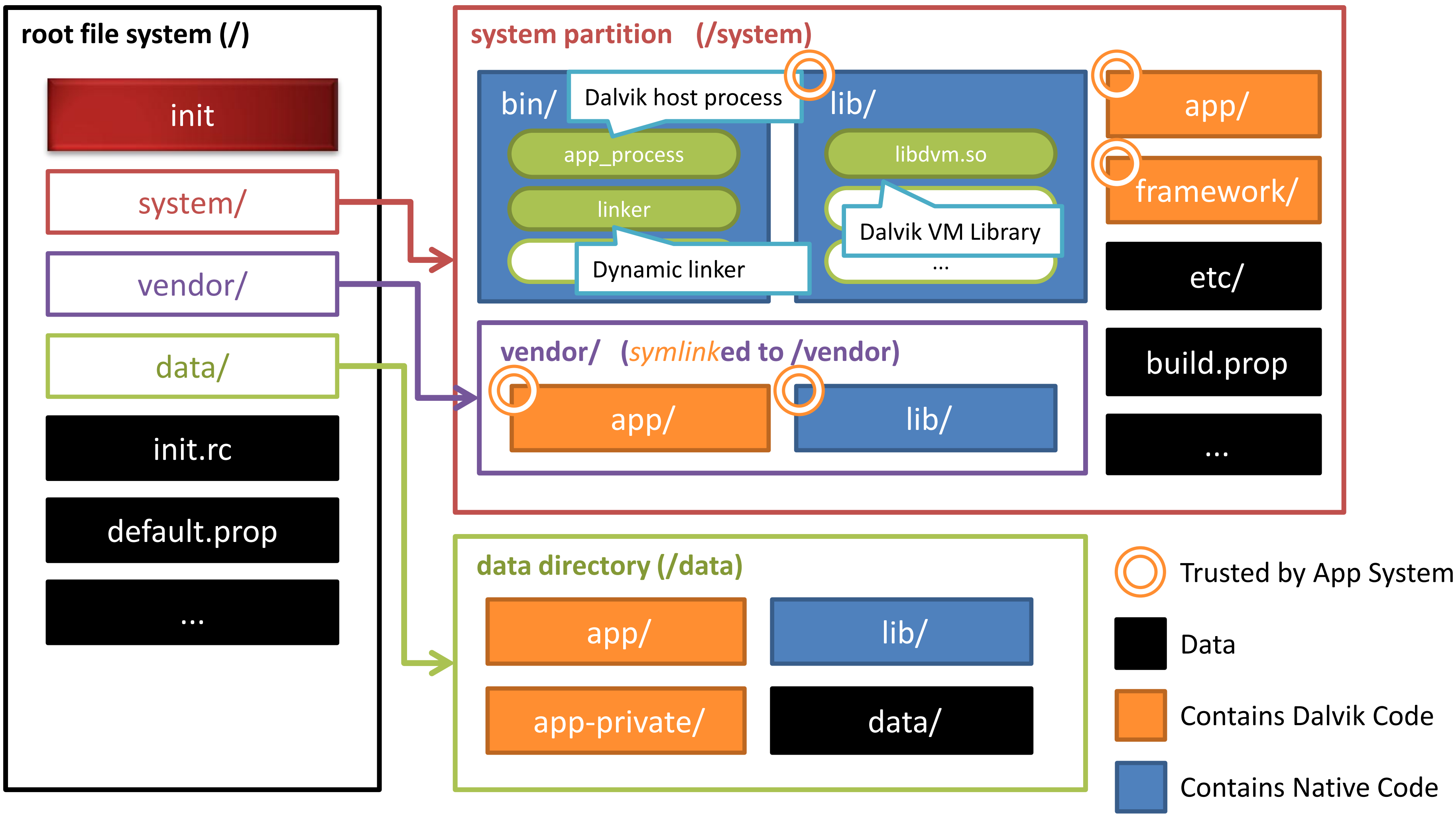
- Applications are contained in the Package
- Register how “classes” are invoked by Manifest
  - System calls application “classes” if requested
  - Activity, Broadcast, ...

Android Internals: Package

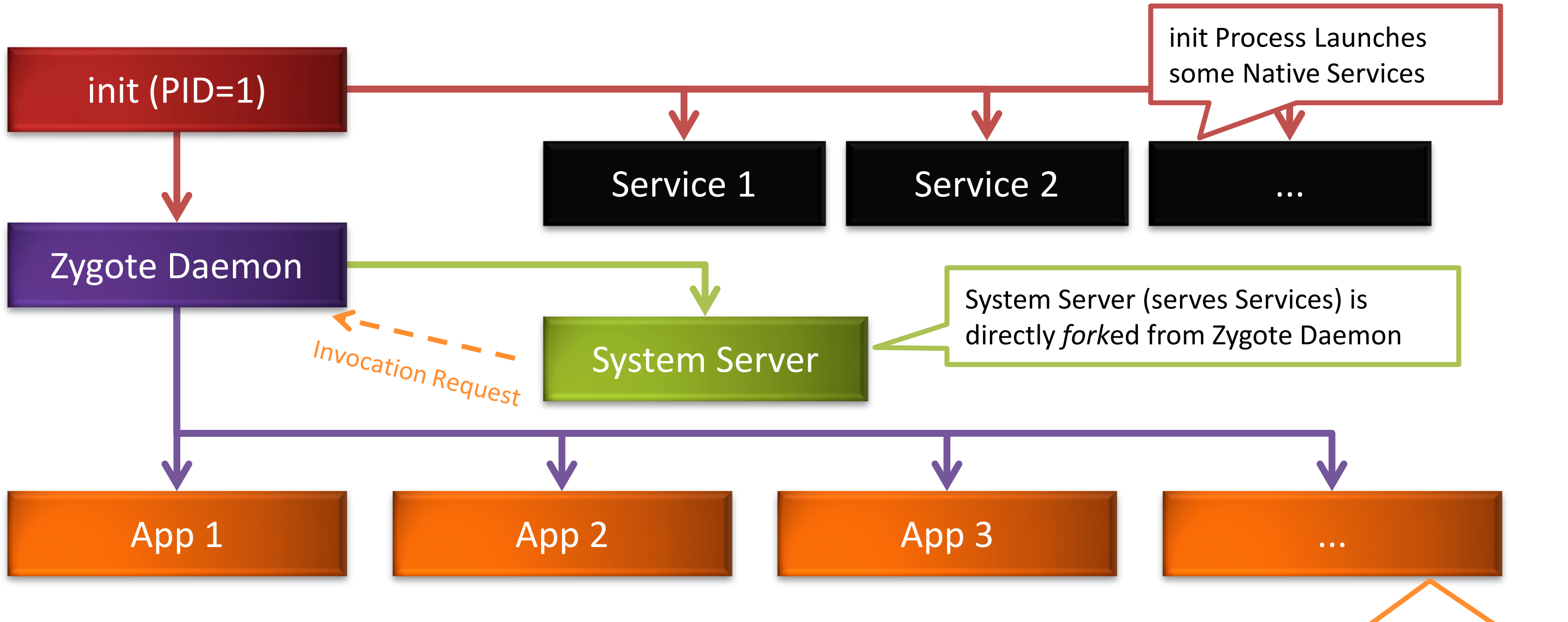


- Package itself is only a ZIP archive
- AndroidManifest.xml (Manifest)
  - Application information, permissions
  - How classes can be called (Activity, BroadcastReceiver...)

Android Internals: App Model in File System

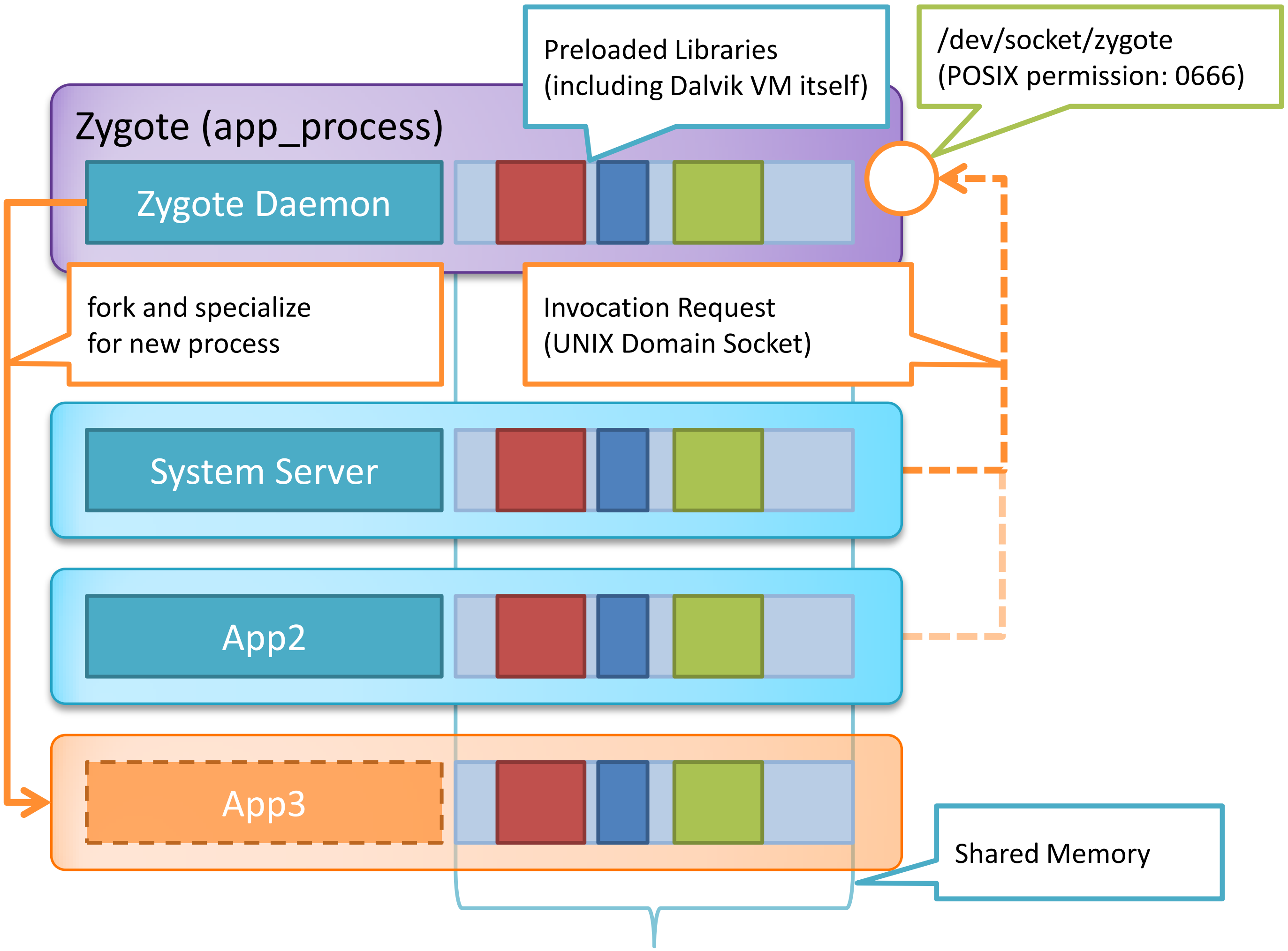


Android Internals: App Model in Lower Layer



- Important Processes are:
  - init (The root of all processes)
  - Zygote Daemon (The root of Android Apps)
  - System Server (serves many System Services)

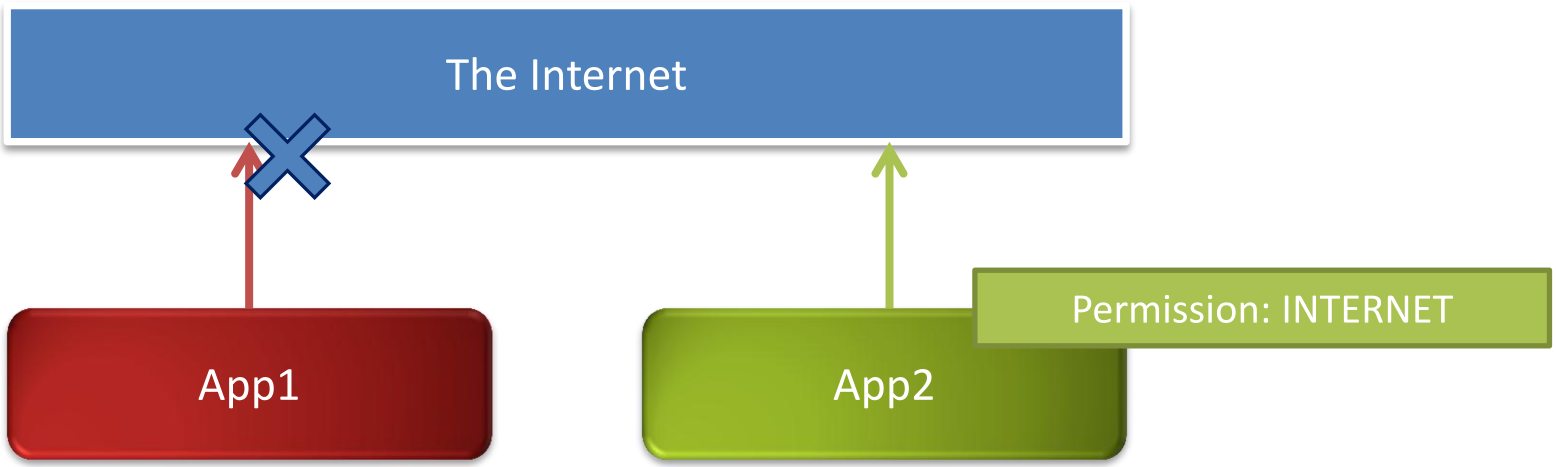
Android Internals: Zygote



Android Security: Model

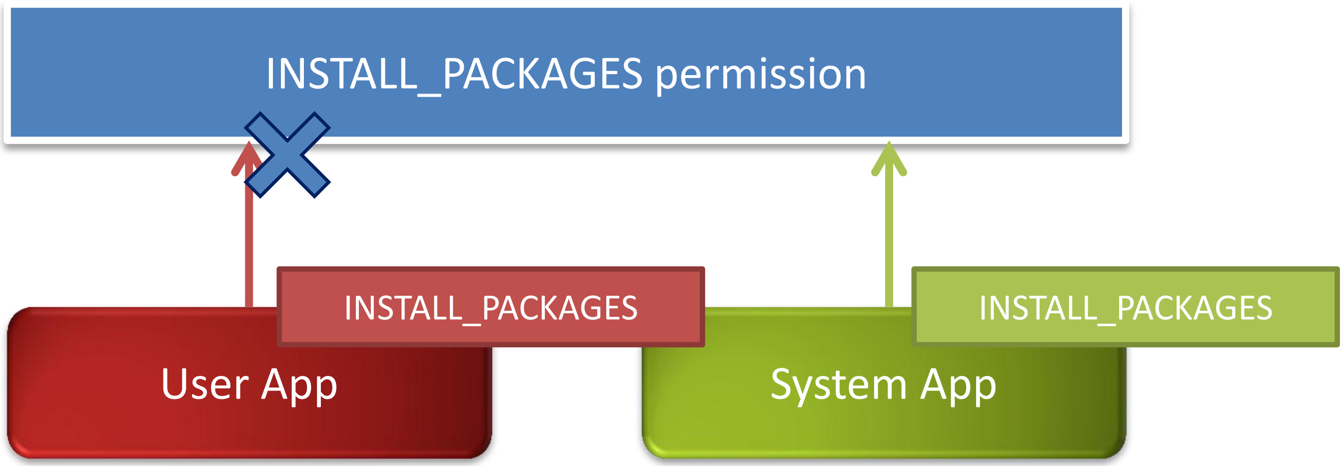
- Android Permission and Protection
  - \+ Grant by Package Information (Permission Information)
  - — Restrict by Package Location (System or User)
  - — Restrict by Package Signature
  - \+ Grant by UID/PID (Backdoor?)
- Priorities of Activity (User-Interface Element)
  - \+ Grant by Package Information (Intent Filters)
  - — Restrict by Package Location (System Only)
- Legacy Linux Security Model
  - – Grant/Restrict: UID/GID/PID...

Android Security: Permission



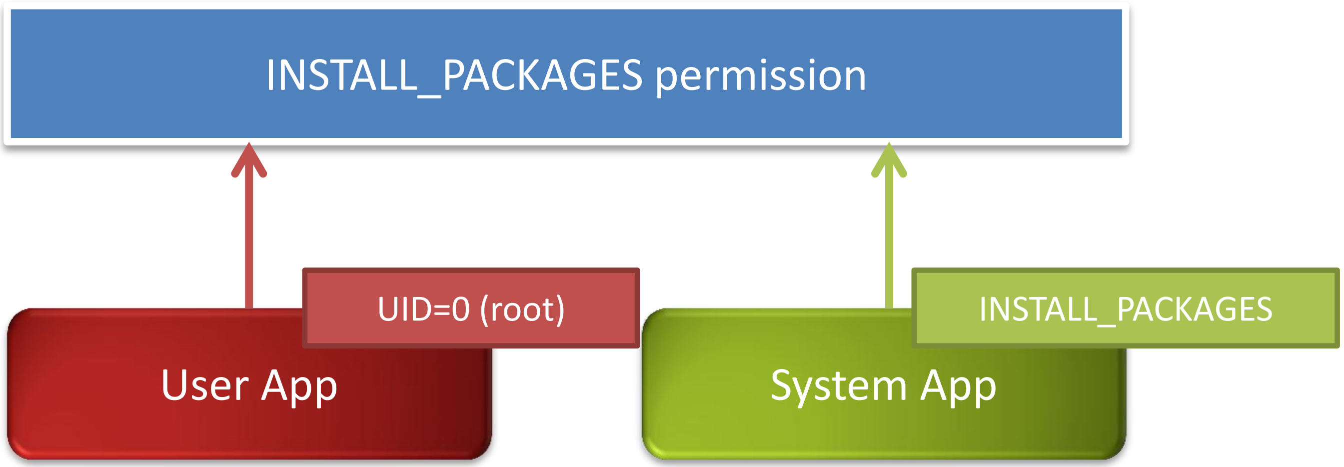
- Abstract "Capability" in Android System
  - More than 100 (Internet connection, retrieve phone number...)
- Permissions Checking
  - Software Checks
  - GID Checks (some permissions are associated with GIDs)

Android Security: Permission Protection



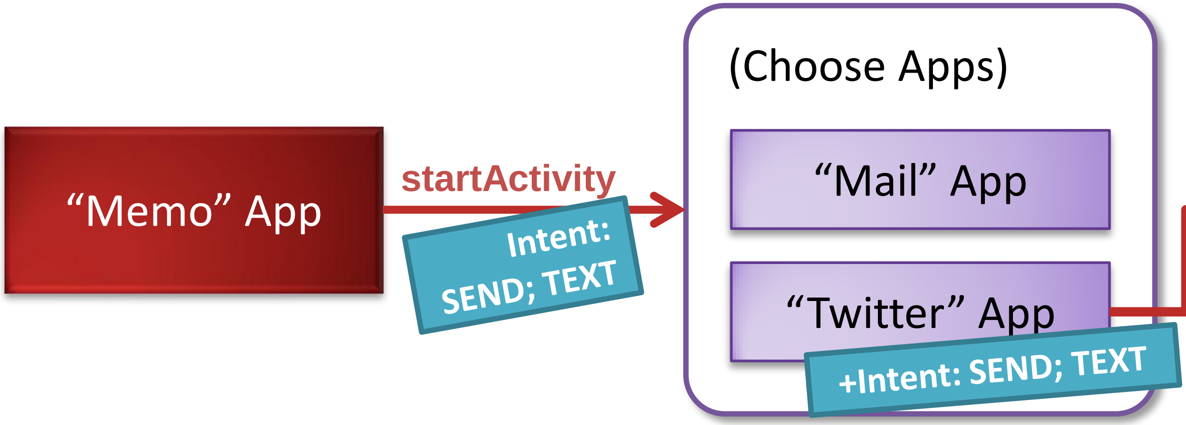
- Permission for User App is Restricted
  - Some permissions are “protected”
- Protection Level
  - Package Location (signatureOrSystem)
  - Package Signature (signature, signatureOrSystem)

Android Security: Permission Protection



- All Permissions are granted for *root* process
  - Permission Checks are not really Performed
- GingerMaster (malware) utilizes this behavior
  - GingerMaster calls pm command via root shell script
  - pm is actually a Dalvik program

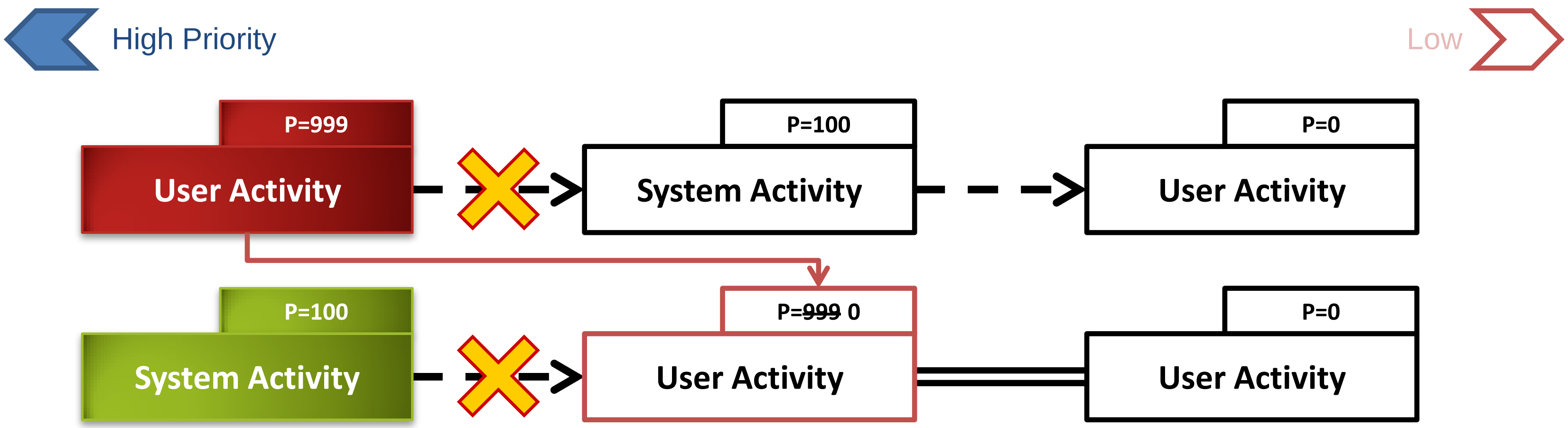
Android Internals: Activity



Intent and multiple applications (Activities)

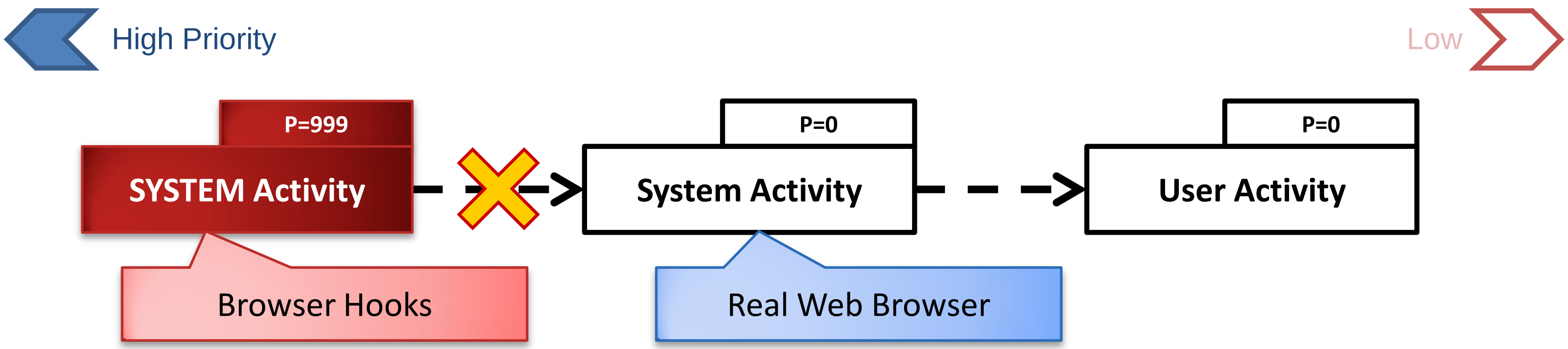
- Activity = Unit of "Action" with User Interface
  - Specifying object type (target) and action, Activity is called by the system automatically

Android Security: Activity Priorities



- Prevent Activity Hooking
  - High-priority Activity can hide lower Activities
- Only System Packages can use Higher Priority
  - e.g. Android Market (Vending.apk)

Bypassing Security: Activity Priorities



- Simply need to write System Locations
  - /system/app, /vendor/app... (Normally write-protected)
- DEMO

Breaking Security: *root* can simply...

- Write System Partition
  - Overwrite Framework, Applications
- Use *chroot*
  - Make fake root and make system partition virtually
- Use *ptrace*
  - Inject Malicious Hooks

- root can spoil Android security mechanism.
  - Or is it?

AOSP is not the everything.

VENDOR-SPECIFIC PROTECTION

Vendor-Specific Protection

- Some Android devices have Additional Security Feature
  - Restrict *root* privileges to prevent devices to be overwritten
- Modification to the Kernel
  - NAND Lock
  - Secure [Authenticated] Boot
  - Integrity Checking
  - Linux Security Modules (LSM)

Vendor-Specific: NAND Lock

- Reject all WRITE requests to important regions
  - Boot Loader
  - System Partition
  - Recovery Partition
- Implemented as a NAND driver feature

- pros. Strong
  - Prohibits ALL illegal writes in kernel mode
- cons. Does not Protect Memory
  - Still can use *ptrace*

Vendor-Specific: Secure Boot

- Prevent Unsigned Boot Loader / Kernel to be Executed
  - Hardware Implementation:
    - e.g. nVidia Tegra
  - Software (Boot Loader) Implementation:
    - e.g. HTC Vision (Qualcomm's Implementation)

- pros. Hard to Defeat
  - Haven't defeated directly
- cons. Only Protects Boot Loader / Kernel
  - Does not Protect On-Memory Boot Loader / Kernel
  - Most implementations does not Protect System Partition

Vendor-Specific: Integrity Verification

- Verify loaded packages / programs are legitimate
  - Restrict some features if untrusted packages / programs are loaded
- Sharp Corp. : Sphinx (Digest Manager)
  - Protected Storage in Kernel Mode
  - Digest Verifier in User-mode (dgstmgrd)
    - Exports Content Provider
- pros. Ability to use Digital Signatures
- cons. Easy to avoid if processes can be compromised
  - e.g. *ptrace*

Vendor-Specific: Linux Security Modules (1)

- Security Framework in Linux Kernel
  - Used by SELinux (for example)
- LSM to Protect Android System

- Sharp Corp. : Deckard LSM / Miyabi LSM
  - Protect Mount Point (/system)
  - Prohibit *ptrace*
  - Prohibit *chroot*, *pivot_root*...
- Fujitsu Toshiba Mobile Communications : fjsec
  - Protect Mount Point (/system) and the FeliCa [subset of NFC] device
  - Prohibit *pivot_root*
  - Path-based / Policy-based Restrictions

Vendor-Specific: Linux Security Modules (2)

- LSM (and NAND lock) Stops DroidDream
  - DroidDream tries to remount /system read-write but it is prohibited by the LSM
- pros. Mandatory and Strong
  - Difficult to Defeat
  - Capable to Hook System Calls
- cons. Difficult to Protect "Everything"
  - ...unless you know all about Android Internals
  - That could lead to LSM bypassing
    - Some holes were fixed though...

Bypassing All Protections

- Restrictions
  - No Kernel-Mode
  - No /proc/*/mem, /dev/*mem
  - No *ptrace*
  - No *chroot*, *pivot_root*
  - No writes to system partitions (/system)
- But Assume if the attacker can gain *root* Privileges
  - Possibility to take over whole system
- User-Mode Rootkit

/protecting/system/is/not/enough/

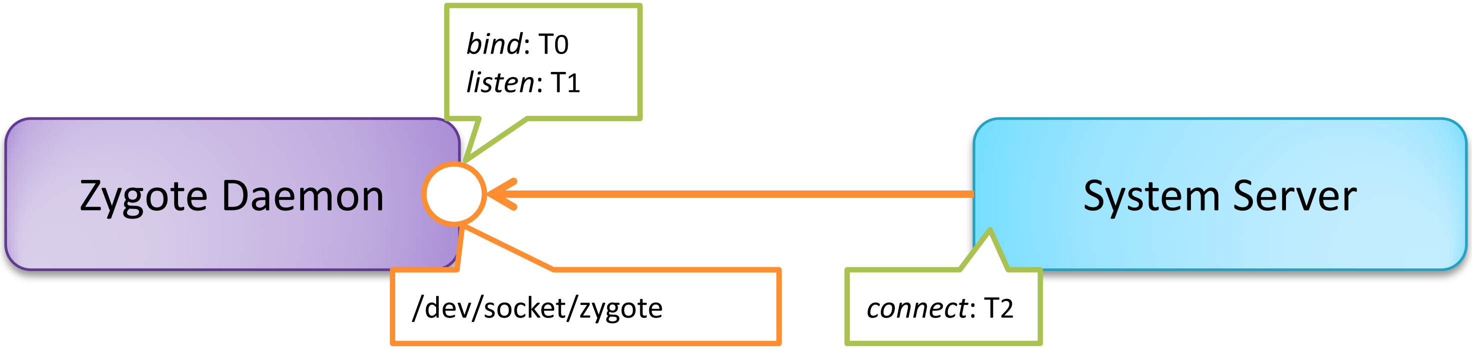
YET ANOTHER ANDROID ROOTKIT

Injecting Hooks: 0 out of 3

Injecting Hooks: Taint Zygote (1)

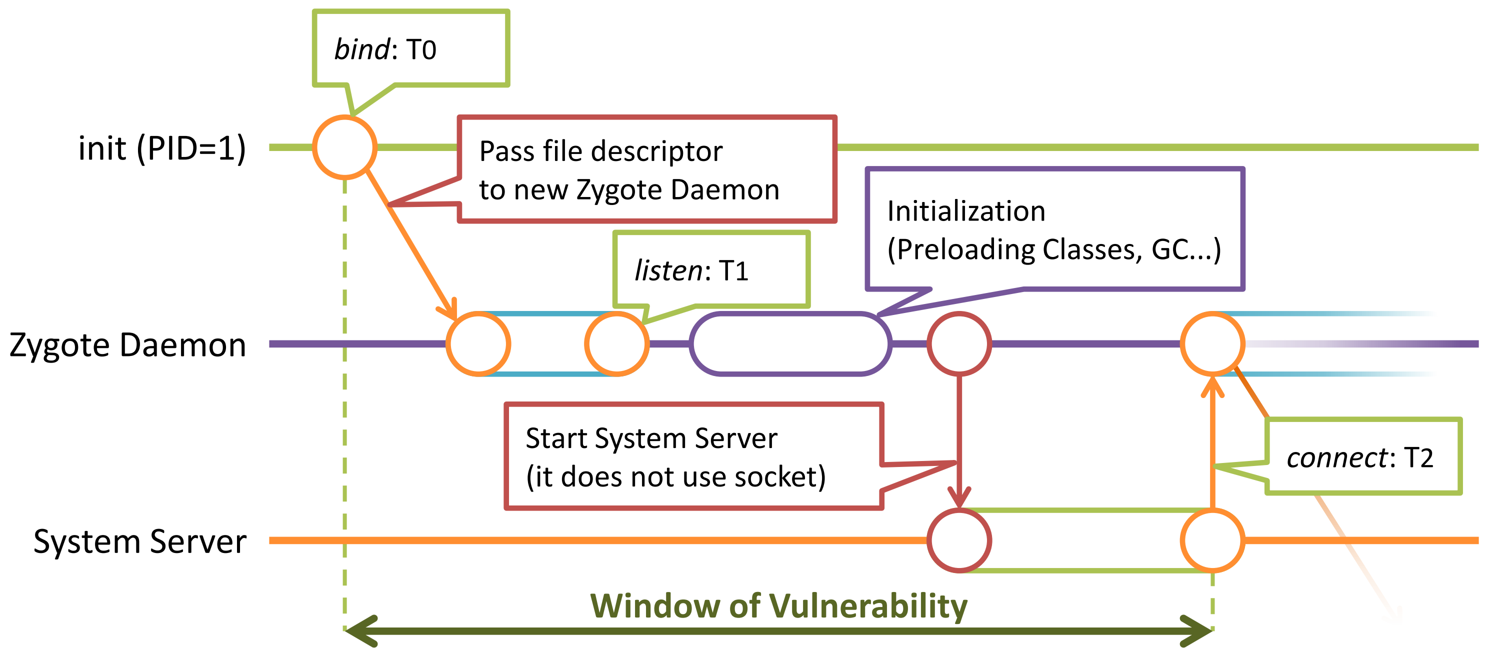
- Facts:
  - All normal Android Apps are *fork*ed from Zygote Daemon
  - Zygote Daemon *fork*s child on request through UNIX-domain socket
- Two plans:
  - Plan A: Hooking UNIX-domain Socket
    - Stealthy
  - Plan B: Generating two Zygote processes
    - Easy to implement
    - Flexible

Injecting Hooks: Taint Zygote (Plan A - 1)



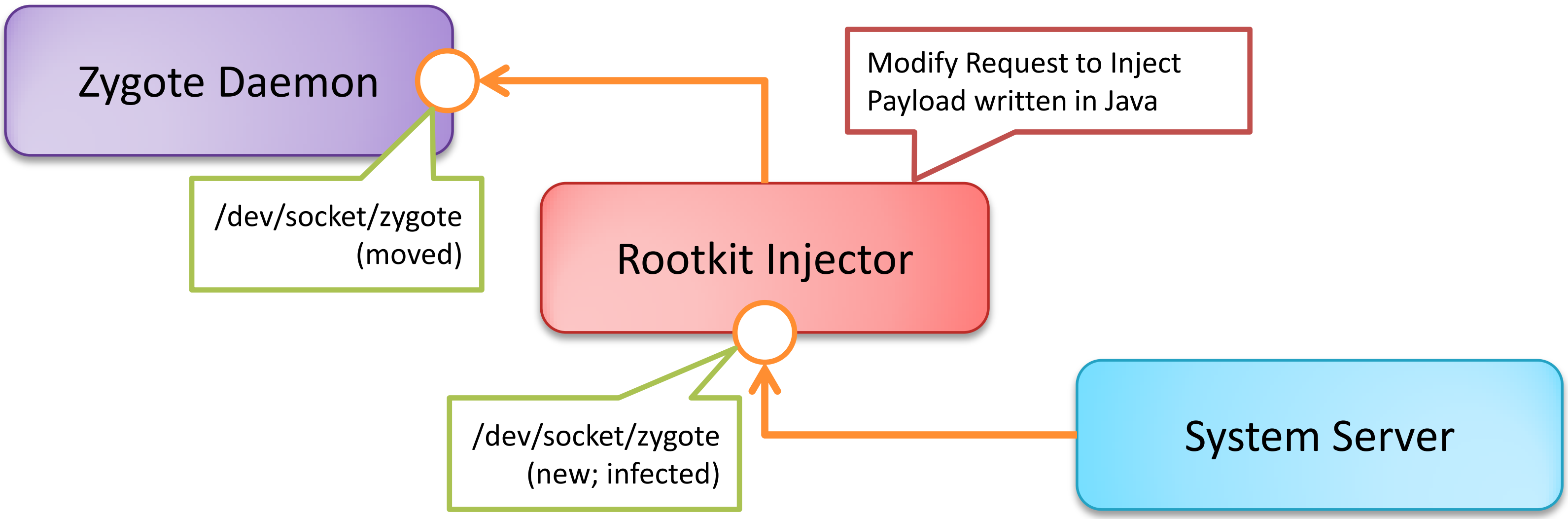
- Exploit race-condition during Initialization of Zygote Daemon
  - Time until the first process is requested
  - Window of Vulnerability is very wide (almost 2~3 seconds)

Injecting Hooks: Taint Zygote (Plan A - 2)



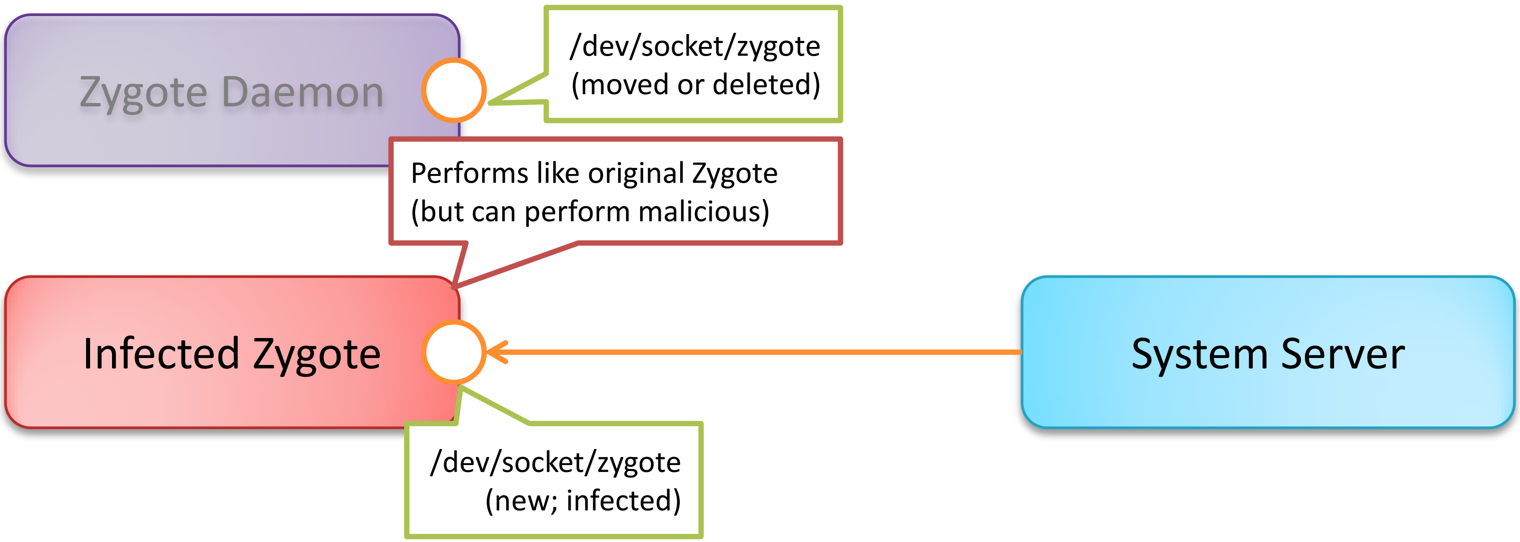
- Exploit race-condition during Initialization of Zygote Daemon
  - Time until the first process is requested
  - Window of Vulnerability is very wide (almost 2~3 seconds)

Injecting Hooks: Taint Zygote (Plan A - 3)



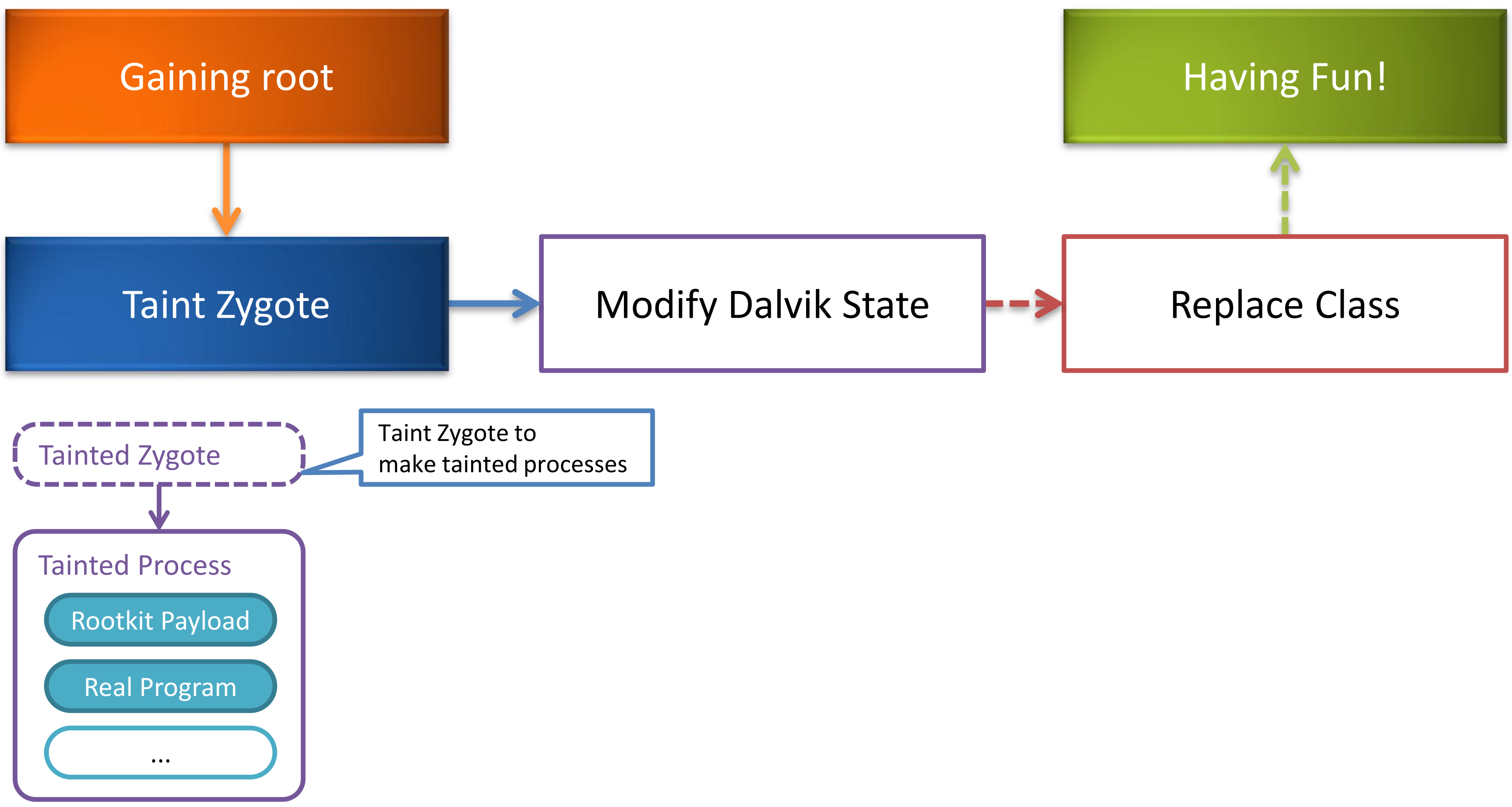
- Perform Man-in-the-Middle Attack
  - System Server refers Rootkit's Socket
- Rootkit Injector can restore original Socket to make it stealth
  - New Apps are requested from one connection between System Server

Injecting Hooks: Taint Zygote (Plan B)

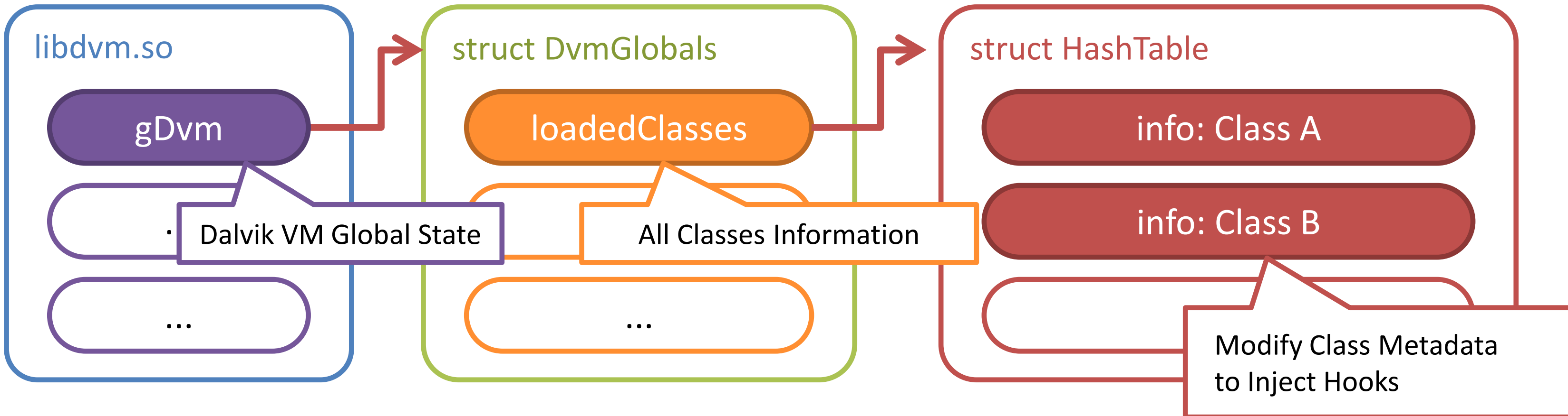


- Pause original Zygote Daemon
- Launch Tainted instance of Zygote
  - Many ways to launch tainted Zygote
- Replace socket with rootkit’s one

Injecting Hooks: 1 out of 3

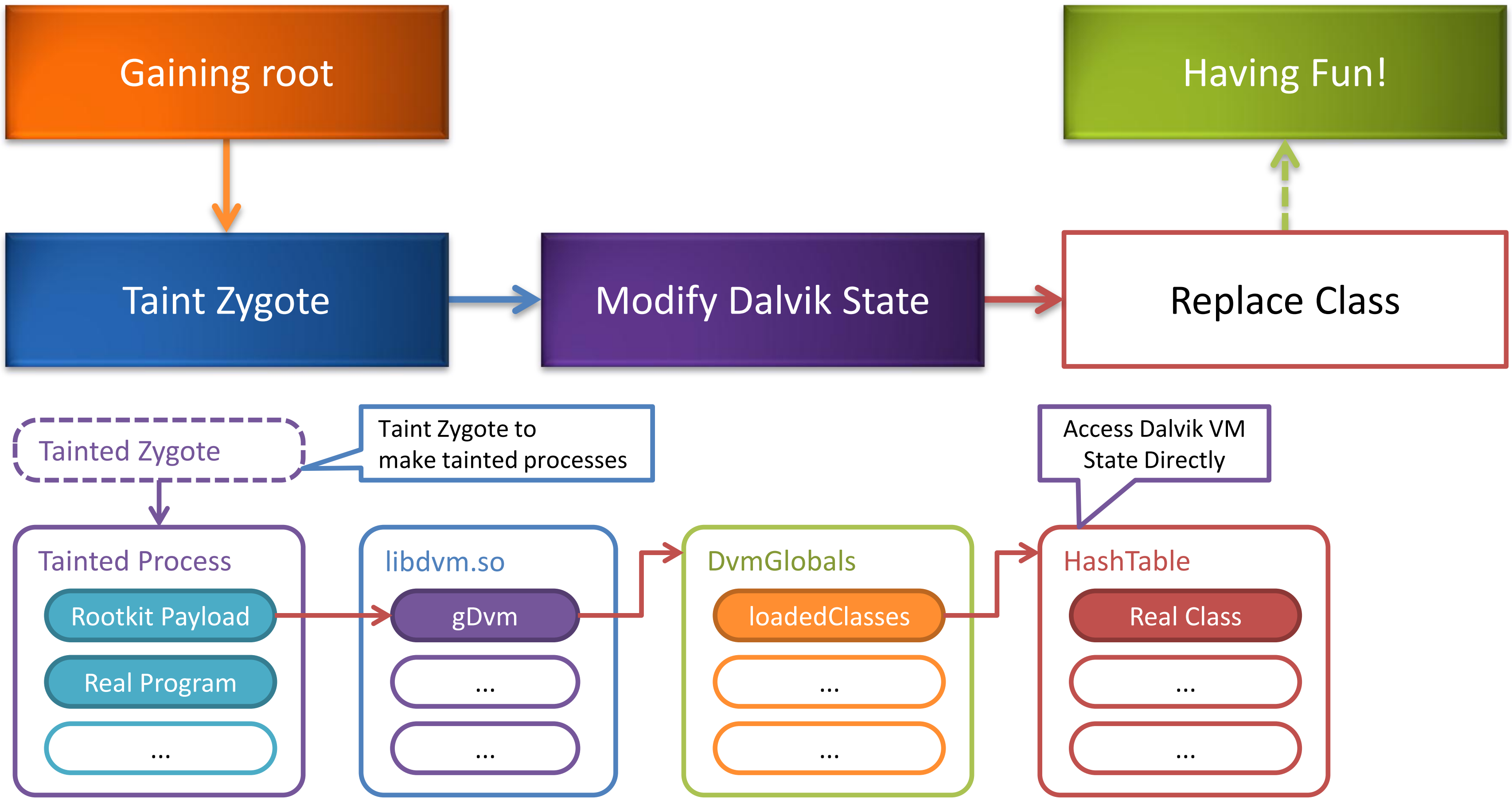


Injecting Hooks: Modify Dalvik State

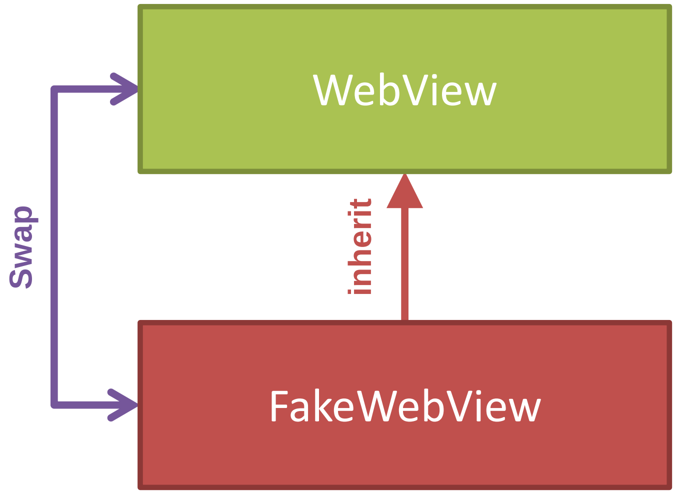


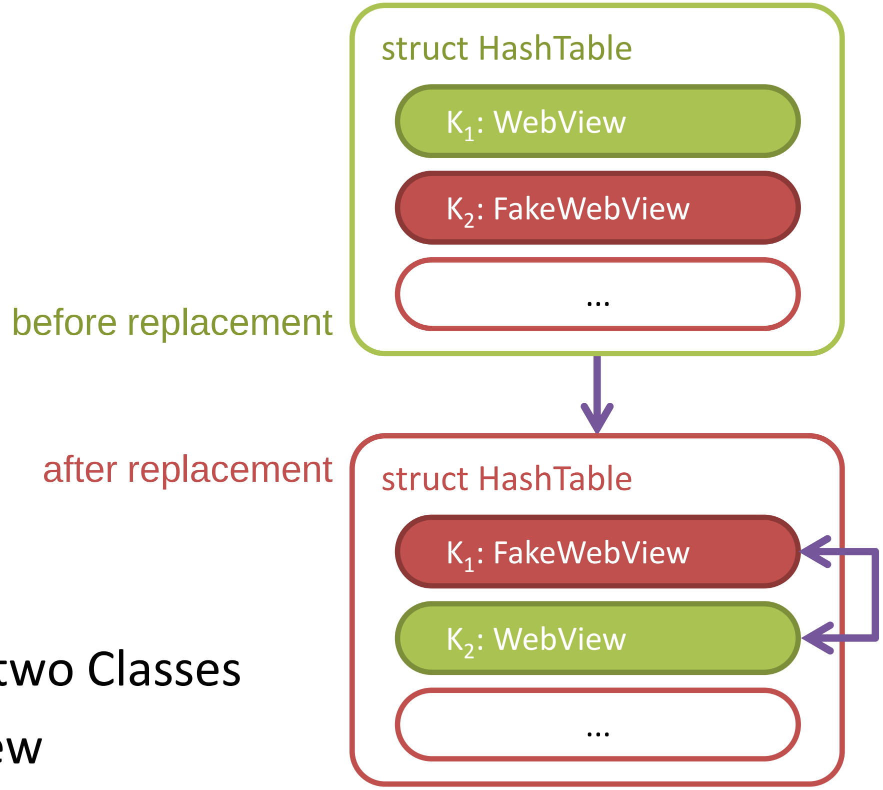
- Assume: The attacker can execute malicious Java class
- Modify Dalvik VM state to inject hooks
  - Read/Write arbitrary memory required
  - sun.misc.Unsafe class
- Dalvik VM (libdvm.so) exports many symbols
  - Including its Global State (gDvm)
  - Modifying gDvm enables hook injection

Injecting Hooks: 2 out of 3



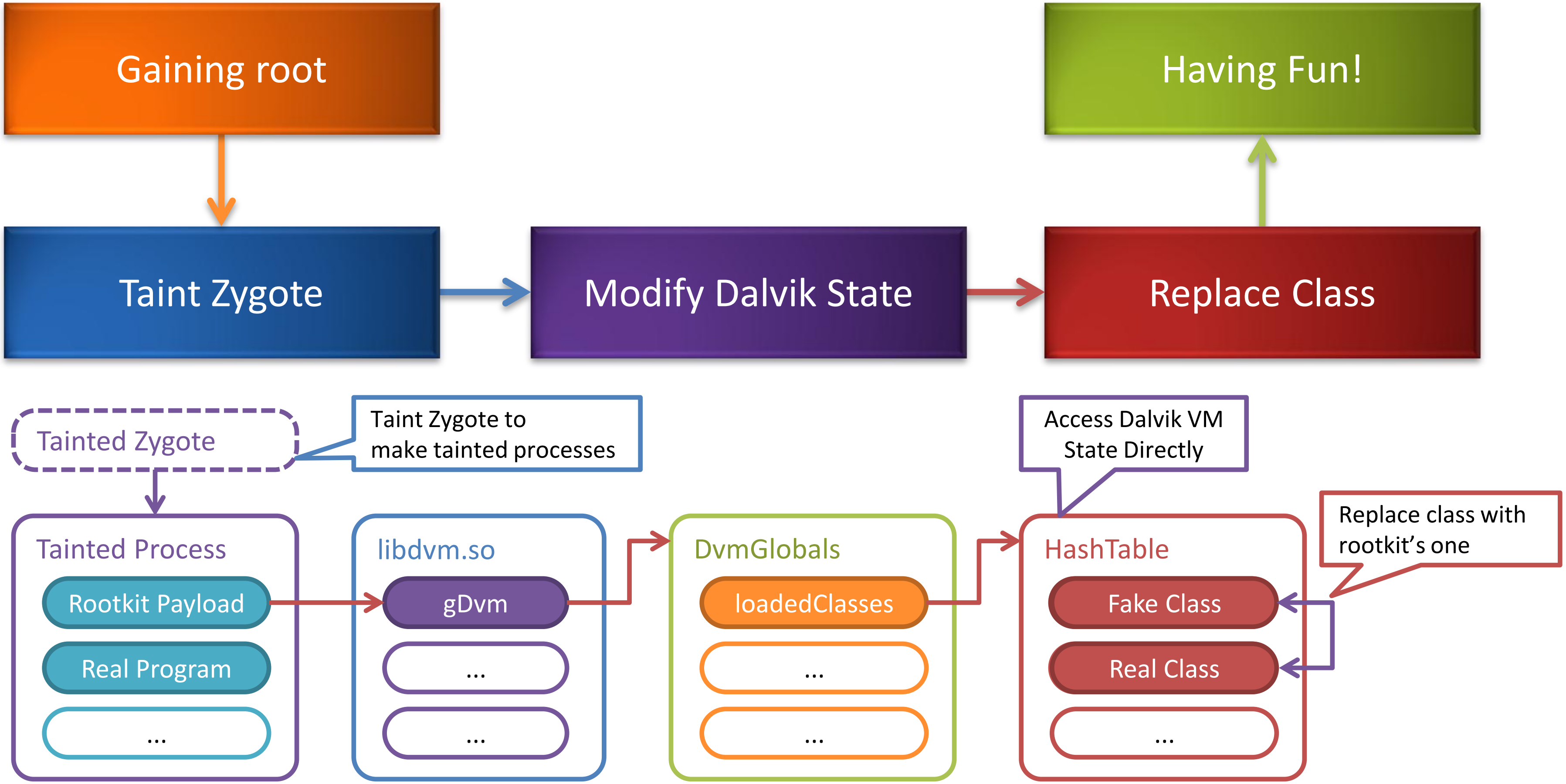
Injecting Hooks: Class Replacement/Swapping





- Easy Implementation Plan: Swap two Classes
  – e.g. WebView ⇔ FakeWebView
  – Target = gDvm->loadedClasses
  – Replacing classes must have exactly same methods

Injecting Hooks: Complete!



Conclusion

- By tainting Zygote,
  we can hook many of activities including method calls
  - Rootkit Payload can be implemented in Pure Java
- Most of implementation are not so difficult
  - Be aware of these kind of attacks

On-memory modification gives attackers ultimate flexibility.

DEMO

Protecting system is not so easy.

BOTTOM LINE

This is not...

- This Android “weakness” is not a vulnerability alone
- This malware is not a really advanced rootkit
  - Easy to detect, Easy to defeat

- But it’s not the point.

So, what was wrong?

- Protection: LSM...
  - Need to know Android Internals
- Difference: Security Requirements
  - Some Japanese smartphones had higher security requirements
  - Different than Google expects

Android: Open source, Closed platform

- Low Open Governance Index[1]
  - Not everything is shared
- Vendor have to implement its own LSM and/or protection
  - Compatibility Issues
  - e.g. Deckard / Miyabi LSM prohibits **all** native debugging
- Can Google provide additional information to implement LSM?
  - To Defeat Compatibility Issues
  - To Make implementing Additional Security Easier

(1) http://www.visionmobile.com/research.php#OGI

Suggestions / Conclusions

- Suggestion: Make policy guidelines to protect Android devices
- Suggestion: Understand what's happening inside the Android system

- If the attacker can gain *root* privileges, the attacker can inject rootkit hooks and monitor App activities

- This is easy to protect, but it implies many of other possibilities
  - Advanced Android malware?

- Share the knowledge to protect Android devices!

Thank You!

Fourteenforty Research Institute, Inc.
http://www.fourteenforty.jp

Research Engineer – Tsukasa Oi
<oi@fourteenforty.jp>