ASIA 2015

DABiD

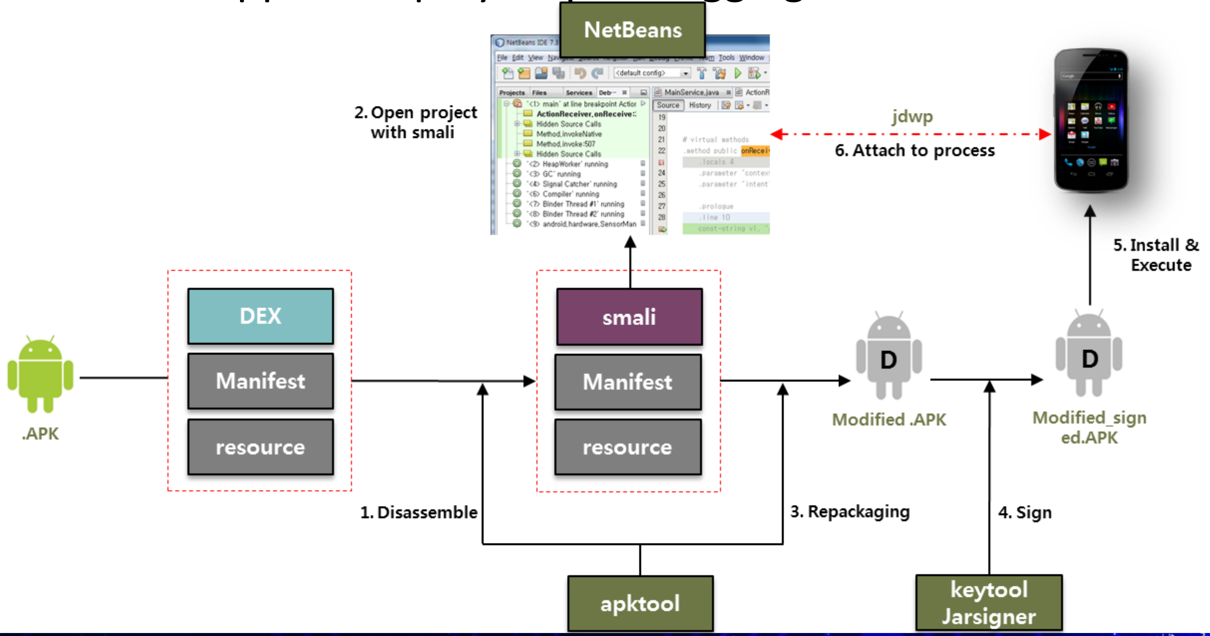
The Powerful Interactive Android Debugger for Android Malware Analysis

Reversing Android application

- Static Analysis
  - — Analyze intent of application by decoding DEX(Dalvik Executable) into readable bytecode
    - ex) Apktool, JEB
- Dynamic Analysis
  - — Monitor behavior of android application at runtime
    - ex) DroidBox, Mobile Sandbox, Anubis etc.
  - — Conduct step by step debugging with disassembled Dalvik executable code
    - ex) SmaliDebugging, IDAPro

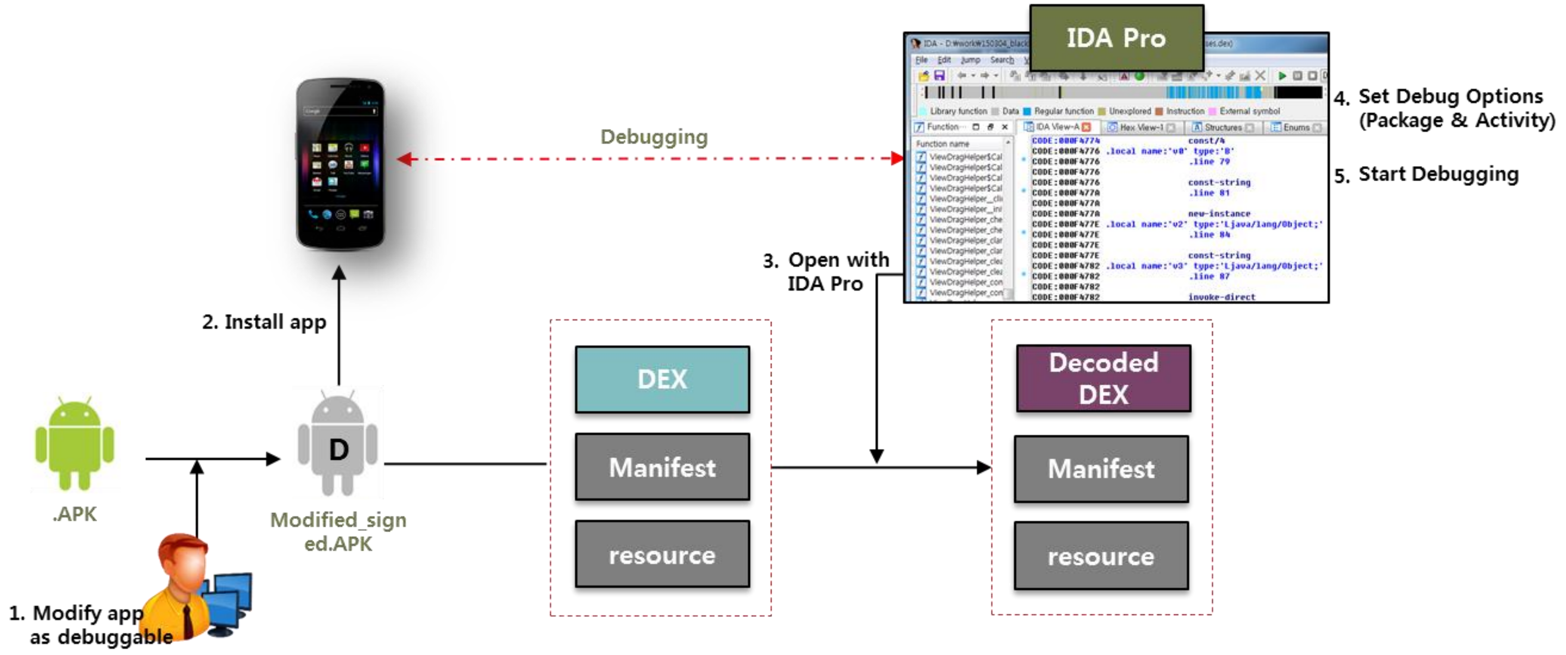
Dalvik Executable Debugging

- Smali Debugging
  - Use apktool and NetBeans(Java IDE) in combination
    - Apktool : disassemble DEX and repackage app in debug mode
    - Java IDE : support step by step debugging



Dalvik Executable Debugging

- IDA Pro Debugging
  - Supports dalvik debugging from version 6.6
  - Similar to Smali Debugging but use own DEX disassembler



Dalvik Executable Debugging

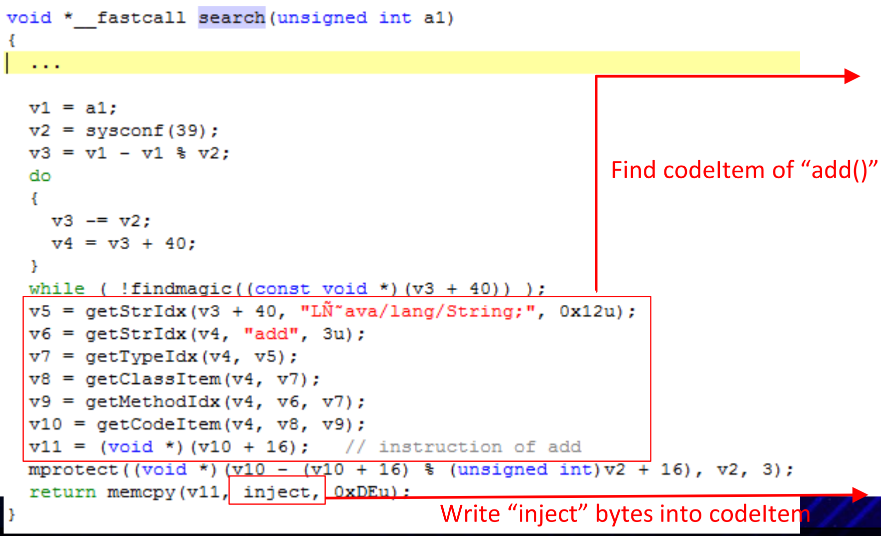
- Smali Debugging VS. IDA Pro Debugging

	Preprocessing		Dalvik Executable Disassemble		Debugging	
	Application Modification	Debugging Settings	Disassembler	Register Type	Debugging Starting Point	Dex Used In Debugging
Smali Debugging	modified as debuggable	Jdwp socket host & port	Smali	Correct Type	First BP hit after debugger attached	Extracted from apk
IDA Pro	modified as debuggable	Package & lauchable activity name	IDA Pro	All registers casted as "Object" (java.lang.Object)	Methods at launchable activity	Extracted from apk
	Should be done manually!			Bad Type Fault!	Can't debug from the start..	

What if…The dex I am debugging is not the one running??

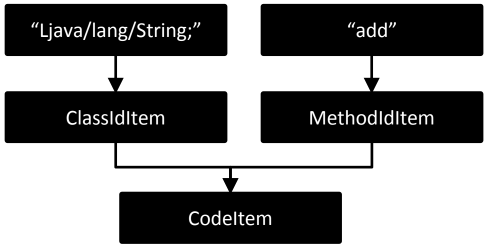
Challenges in Android Analysis
: Modification of DEX bytes at runtime

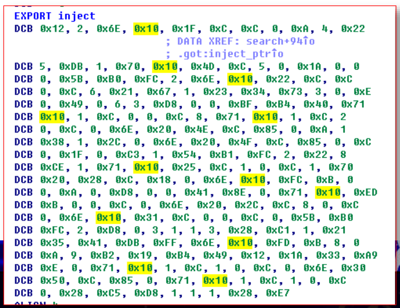
- BlueBox Security verified tempering davik bytecode during runtime is possible
  - Load library and execute function which write bytes into memory where dalvik executable is loaded
    - Find codeItem of "add()" method from DEX loaded in memory
    - Write bytes into codeItem of "add()" method



```
void *__fastcall search(unsigned int a1)
{
  ...

  v1 = a1;
  v2 = sysconf(39);
  v3 = v1 - v1 % v2;
  do
  {
    v3 -= v2;
    v4 = v3 + 40;
  }
  while ( !findmagic((const void *)(v3 + 40)) );
  v5 = getStrIdx(v3 + 40, "LÑ˜ava/lang/String;", 0x12u);
  v6 = getStrIdx(v4, "add", 3u);
  v7 = getTypeIdx(v4, v5);
  v8 = getClassItem(v4, v7);
  v9 = getMethodIdx(v4, v6, v7);
  v10 = getCodeItem(v4, v8, v9);
  v11 = (void *)(v10 + 16);    // instruction of add
  mprotect((void *)(v10 - (v10 + 16) % (unsigned int)v2 + 16), v2, 3);
  return memcpy(v11, inject, 0xDEu);
}
```

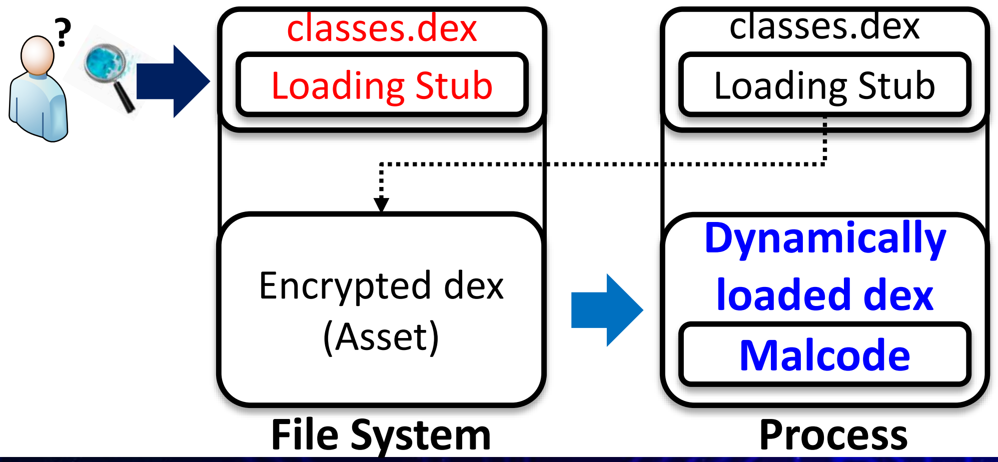




```
EXPORT inject
DCB 0x12, 2, 0x6E, 0x10, 0x1F, 0xC, 0xC, 0, 0xA, 4, 0x22
                        ; DATA XREF: search+94↑o
                        ; .got:inject_ptr↑o
DCB 5, 0xDB, 1, 0x70, 0x10, 0x4D, 0xC, 5, 0, 0x1A, 0, 0
DCB 0, 0x5B, 0xB0, 0xFC, 2, 0x6E, 0x10, 0x22, 0xC, 0xC
DCB 0, 0xC, 6, 0x21, 0x67, 1, 0x23, 0x34, 0x73, 3, 0, 0xE
DCB 0, 0x49, 0, 6, 3, 0xD8, 0, 0, 0xBF, 0xB4, 0x40, 0x71
DCB 0x10, 1, 0xC, 0, 0, 0xC, 8, 0x71, 0x10, 1, 0xC, 2
DCB 0, 0xC, 0, 0x6E, 0x20, 0x4E, 0xC, 0x85, 0, 0xA, 1
DCB 0x38, 1, 0x2C, 0, 0x6E, 0x20, 0x4F, 0xC, 0x85, 0, 0xC
DCB 0, 0x1F, 0, 0xC3, 1, 0x54, 0xB1, 0xFC, 2, 0x22, 8
DCB 0xCE, 1, 0x71, 0x10, 0x25, 0xC, 1, 0, 0xC, 1, 0x70
DCB 0x20, 0x28, 0xC, 0x18, 0, 0x6E, 0x10, 0xFC, 0xB, 0
DCB 0, 0xA, 0, 0xD8, 0, 0, 0x41, 0x8E, 0, 0x71, 0x10, 0xED
DCB 0xB, 0, 0, 0xC, 0, 0x6E, 0x20, 0x2C, 0xC, 8, 0, 0xC
DCB 0, 0x6E, 0x10, 0x31, 0xC, 0, 0, 0xC, 0, 0x5B, 0xB0
DCB 0xFC, 2, 0xD8, 0, 3, 1, 1, 3, 0x28, 0xC1, 1, 0x21
DCB 0x35, 0x41, 0xDB, 0xFF, 0x6E, 0x10, 0xFD, 0xB, 8, 0
DCB 0xA, 9, 0xB2, 0x19, 0xB4, 0x49, 0x12, 0x1A, 0x33, 0xA9
DCB 0xE, 0, 0x71, 0x10, 1, 0xC, 1, 0, 0xC, 0, 0x6E, 0x30
DCB 0x50, 0xC, 0x85, 0, 0x71, 0x10, 1, 0xC, 1, 0, 0xC
DCB 0, 0x28, 0xC5, 0xD8, 1, 1, 1, 0x28, 0xE7
```

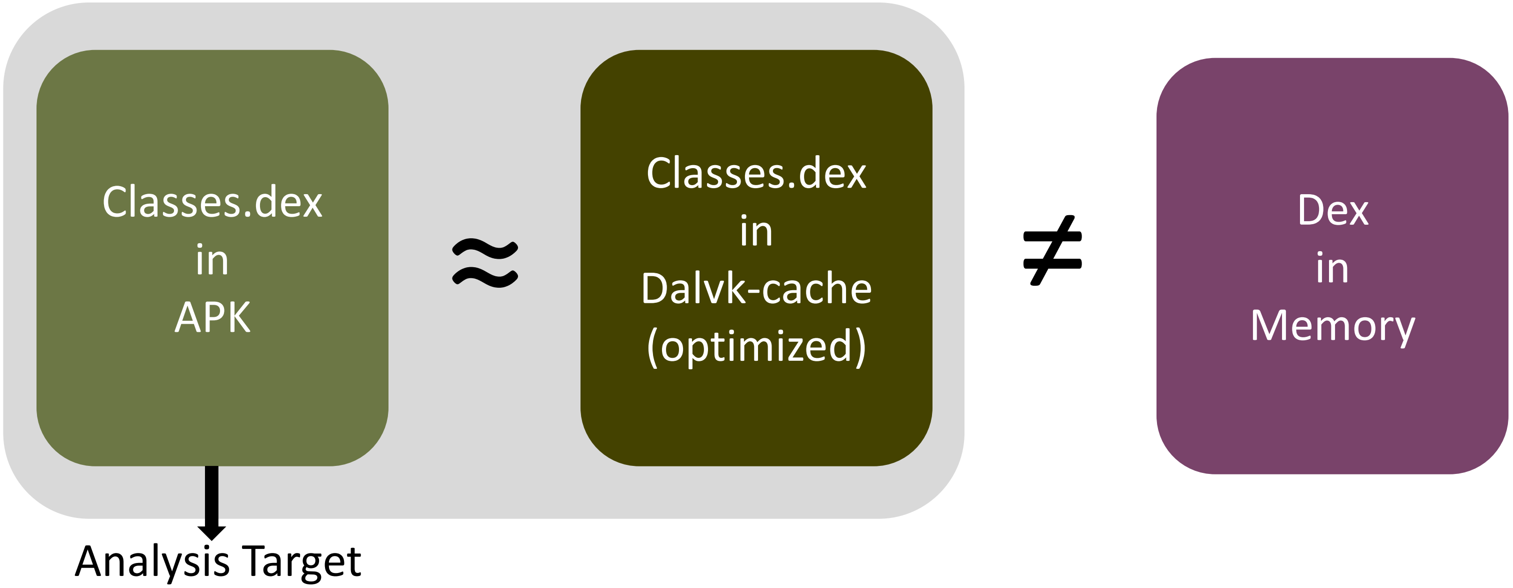
Challenges in Android Analysis
: Dynamic DEX Loading

- GoogleAppsToy malware load encrypted DEX at runtime
  - Analysts can obtain classes.dex from APK and conduct static or dynamic analysis on classes.dex
  - But, classes.dex from APK has no malicious actions
    - Only decrypt and dynamic loading routine exist in classes.dex
    - No way to debug malicious code….



Challenges in Android Debugging

- DEX(Dalvik Executable) can be different in memory



Analysis with current analyzers might be useless..

black hat ASIA 2015

DABiD :

Dynamic Android Binary Debugger

Goal

- To develop android debugger which is able to debug “the same DEX” running on memory
- To make android debugging more effective and convenient for analysts

Our Approach (1/2)

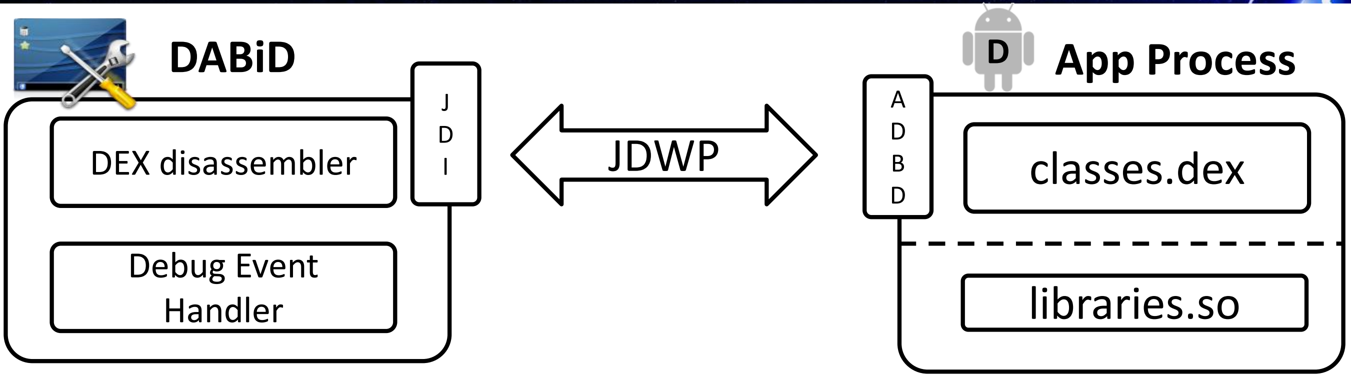
- To develop android debugger which is able to debug "the same DEX" running on memory
  - Monitor dynamic changes in memory and reflect them to debugger
    - Self modification of DEX bytes in memory
    - Dynamic DEX loading

- To make android debugging more effective and convenient for analysts

Our Approach (2/2)

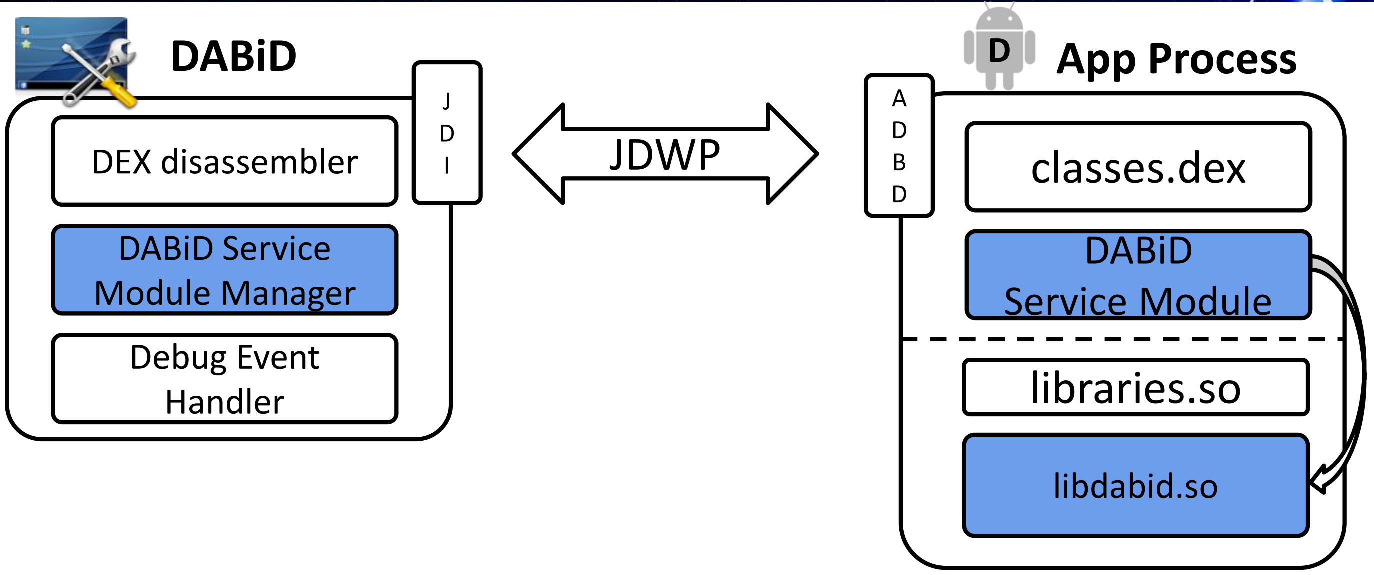
- To develop android debugger which is able to debug "the same DEX" running on memory
- To make android debugging more effective and convenient for analysts
  - Provide advanced debugging features
    - Code update by analyst
    - Register value acquisition
  - Automate bothersome settings for android debugging

DABiD - Overview



- Resembles java debugger structure
  - DEX disassembler : disassemble DEX from both apk file and memory
  - Debug Event Handler : create and handle debugging event from JDWP
- But, JDWP has limitations…

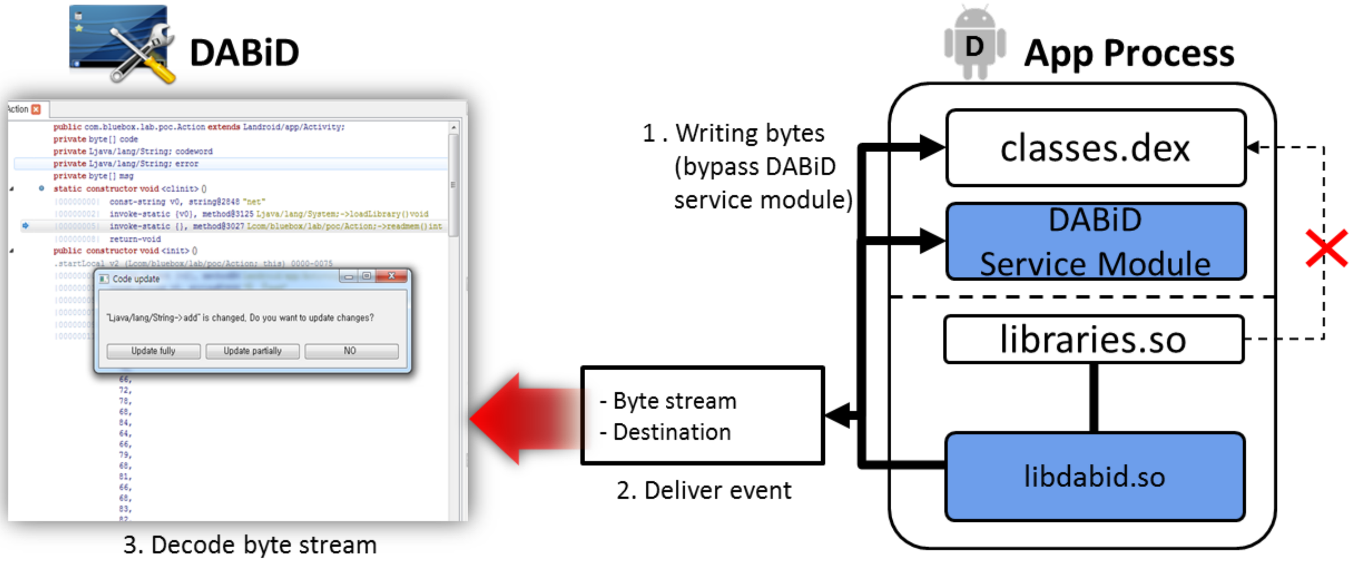
DABiD - Overview



- Make our service module reside in application process
  - Notify supervision results of dynamic changes in memory
  - Give a control over the application

Monitoring Dynamic Changes
: Self modification of DEX bytes

- DABiD Service module detects that memory write function call is made and alarm debugger when event happens



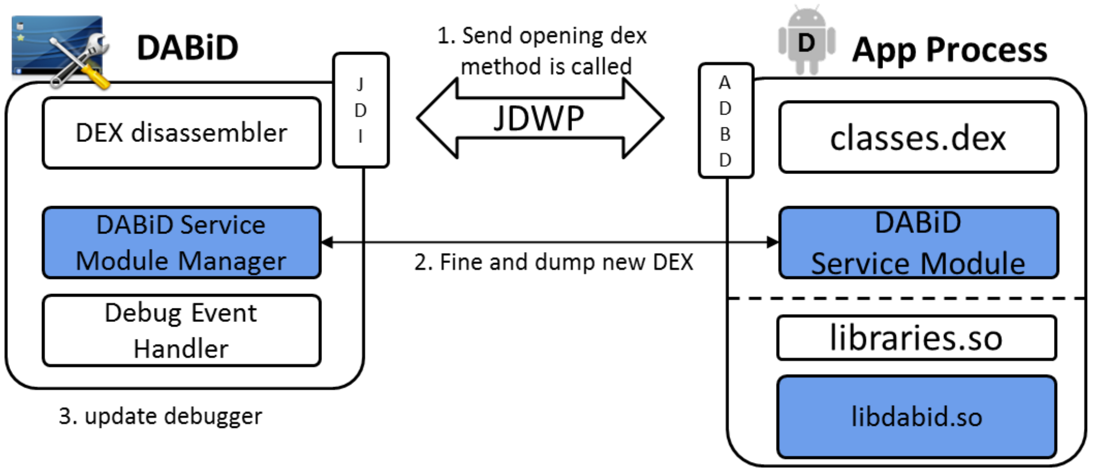
```
public com.bluebox.lab.poc.Action extends Landroid/app/Activity;
private byte[] code
private Ljava/lang/String; codeword
private Ljava/lang/String; error
private byte[] msg
static constructor void <clinit> ()
|00000000|   const-string v0, string@2848 "net"
|00000002|   invoke-static {v0}, method@3125 Ljava/lang/System;->loadLibrary()void
|00000005|   invoke-static {}, method@3027 Lcom/bluebox/lab/poc/Action;->readmem()int
|00000008|   return-void
public constructor void <init> ()
.startLocal v2 (Lcom/bluebox/lab/poc/Action; this) 0000-0075
66,
72,
78,
68,
84,
64,
66,
79,
68,
81,
66,
68,
83,
```

"Ljava/lang/String->add" is changed, Do you want to update changes?

- Analysts is now able to analyze hidden bytes

Monitoring Dynamic Changes
: Dynamic DEX loading

- DABiD Service module find and dump new DEX bytes in memory

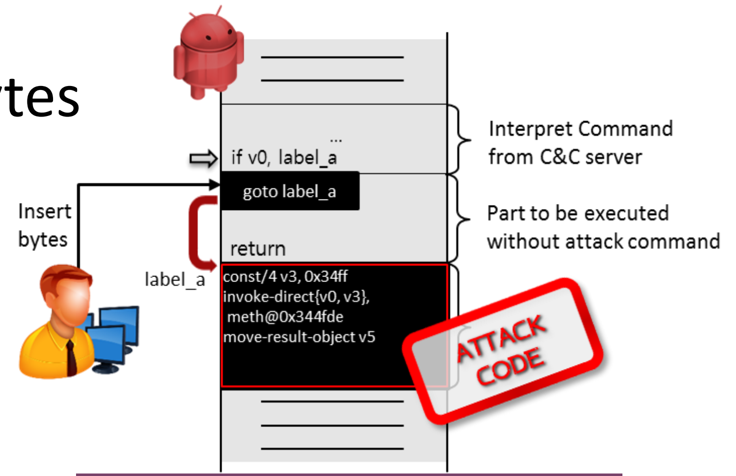


- With DABiD, Analysts no longer need to
  - Find the location of hidden or newly downloaded DEXs
  - Conduct static analysis for new DEXs by pulling them to local

Advanced Debugging Feature
: Code update by Analyst

- Analysts can modify bytecodes on the fly
  - Analysts can input bytes from DABiD
  - DABiD service module writes input bytes to proper location in memory
  - Debuggee runs with modified bytes

- Analysts are able to
  - Force to execute code
  - Skip code part should not be executed to continue analysis



...
if v0, label_a
goto label_a
return
Insert bytes
label_a
const/4 v3, 0x34ff
invoke-direct{v0, v3}, meth@0x344fde
move-result-object v5
Interpret Command from C&C server
Part to be executed without attack command
ATTACK CODE

Force to execute code before the time of event

Advanced Debugging Feature
: Register value acquisition

- JDI provides register values only with debug symbols
  - But, Not all registers have debug symbols

Java	Bytecode	DebugSymbol

int a = 3;
Log.d("Info", "a :" +a);
	.local name :'v0' type: int
	const/4 v0, 3
	const-string v1, "info"
	new-instance v2, Ljava/lang/StringBuilder;
	const-string v3 "a : "
	Invoke-direct {v2, v3} StringBuilder.init()
	Invoke-virtual {v2, v0} StringBuilder.append()
	...
		'v0' – int

- Modify JDI to get values of registers
  - Get register value using slot number
  - Eliminate evaluation check whether the register is visible variable or not
  - Cast value with type information by emulating bytecodes in DEX disassembler

Java	Bytecode	DebugSymbol

int a = 3;
Log.d("Info", "a :" +a);
	.local name :'v0' type: int
	const/4 v0, 3
	const-string v1, "info"
	new-instance v2, Ljava/lang/StringBuilder;
	const-string v3 "a : "
	Invoke-direct {v2, v3} StringBuilder.init()
	Invoke-virtual {v2, v0} StringBuilder.append()
	...
		'v0' – int
		'v1' – Ljava/lang/String;
		'v2' – Ljava/lang/StringBuilder;
		'v3' – Ljava/lang/String;

Analyzed type info

black hat ASIA 2015

Automation of debugging setting

- DABiD automates followings to aid debugging
  - Transform application into debuggable
  - Install and start application
  - Set jdwp socket connection
  - Set breakpoints at the starting point of application

DEMO

Future work

- Debugging for Android Runtime (ART)
  - — Support ART features
  - — Resolve Code Protections on ART

- Code coverage
  - — Support native code debug included in APK

- Anti-Debugging
  - — Counter anti-debugging techniques