6.858 Lecture 21
TAINT TRACKING

What problem does the paper try to solve?
• Applications can exfiltrate a user’s private data and send it to some server.
• High-level approach: keep track of which data is sensitive, and prevent it from leaving the device!
• Why aren’t Android permissions enough?
  o Android permissions control whether application can read/write data, or access devices or resources (e.g., the Internet).
  o Using Android permissions, it’s hard to specify a policy about *particular* types of data (Ex: "Even if the app has access to the network, it should never be able to send user data over the network").

• Q: Aha! What if we never install apps that both read data *and* have network access?
• A: This would prevent some obvious leaks, but it would also break many legitimate apps! [Ex: email app]

• Information can still leak via side channels. [Ex: browser cache leaks whether an object has been fetched in the past]

• Apps can collude! [Ex: An app without network privileges can pass data to an app that does have network privileges.]

• A malicious app might trick another app into sending data. [Ex: Sending an intent to the Gmail app?]

What does Android malware actually do?
• Use location or IMEI for advertisements. [IMEI is a unique per-device identifier.]
• Credential stealing: send your contact list, IMEI, phone number to remote server.
• Turn your phone into a bot, use your contact list to send spam emails/SMS messages!

• Preventing data exfiltration is useful, but taint tracking by itself is insufficient to keep your device from getting hacked!

TaintDroid tracks sensitive information as it propagates through the system.
• TaintDroid distinguishes between information sources and information sinks
  o Sources generate sensitive data:
    • Ex: Sensors, contacts, IMEI
  o Sinks expose sensitive data:
- Ex: network.
- TaintDroid uses a 32-bit bitvector to represent taint, so there can be at most 32 distinct taint sources.
- Roughly speaking, taint flows from rhs to lhs of assignments.

```java
int lat = gps.getLatitude();
//The lat variable is now tainted!
```

Dalvik VM is a register-based machine, so taint assignment happens during the execution of Dalvik opcodes [see Table 1].

```
move_op dst src  //dst receives src's taint
binary_op dst src0 src1  //dst receives union of src0
                        //and src1's taint
```

**Interesting special case: arrays**

```java
char c = ///. . . get c somehow.
char uppercase[] = ['A', 'B', 'C', . . . ];
char upperC = uppercase[c];
    //upperC's taint is the
    //union of c and uppercase's
    //taint.
```

To minimize storage overheads, an array receives a single taint tag, and all of its elements have the same taint tag.

**Q:** Why is it safe to associate just one label with arrays or IPC messages?
**A:** It should be safe to *over*-estimate taint. This may lead to false positives, but not false negatives.

**Another special case: native methods (i.e., internal VM methods like System.arraycopy(), and native code exposed via JNI).**
- **Problem:** Native code doesn’t go through the Dalvik interpreter, so TaintDroid can’t automatically propagate taint!
- **Solution:** Manually analyze the native code, provide a summary of its taint behavior.
  - Effectively, need to specify how to copy taints from args to return values.
  - **Q:** How well does this scale?
  - **A:** Authors argue this works OK for internal VM functions (e.g., arraycopy). For "easy" calls, the analysis can be automated---if only integers or strings are passed, assign the union of the input taints to the return value.
IPC messages are like treated like arrays: each message is associated with a single taint that is the union of the taints of the constituent parts.

- Data which is extracted from an incoming message is assigned the taint of that message.

Each file is associated with a single taint flag that is stored in the file’s metadata.
- Like with arrays and IPC messages, this is a conservative scheme that may lead to false positives.

How are taint flags represented in memory?

- Five kinds of things need to have taint tags:
  1) Local variables in a method — live on stack
  2) Method arguments
  3) Object instance fields
  4) Static class fields
  5) Arrays

- Basic idea: Store the flags for a variable near the variable itself.
- Q: Why?
- A: Preserves spatial locality---this hopefully improves caching behavior.

- For method arguments and local variables that live on the stack, allocate the taint flags immediately next to the variable.

```
+------------------+
|     local0       |
+------------------+
| local0 taint tag |
+------------------+
|     local1       |
+------------------+
| local1 taint tag |
+------------------+

• TaintDroid uses a similar approach for class fields, object fields, and arrays—put the taint tag next to the associated data.

So, given all of this, the basic idea in TaintDroid is simple: taint sensitive data as it flows through the system, and raise an alarm if that data tries to leave via the network!
- The authors find various ways that apps misbehave. Ex:
Sending location data to advertisers
Sending a user's phone number to the app servers

TaintDroid’s rules for information flow might lead to counterintuitive/interesting results.
• Imagine that an application implements its own linked list class.

```java
class ListNode{
    Object data;
    ListNode next;
}
```

• Suppose that the application assigns tainted values to the "data" field. If we calculate the length of the list, is the length value tainted?

• Adding to a linked list involves:
  1) Allocating a ListNode
  2) Assigning to the "data" field
  3) Patching up "next" pointers
• Note that Step 3 doesn't involve tainted data! So, "next" pointers are tainted, meaning that counting the number of elements in the list would not generate a tainted value for length.

What are the performance overheads of TaintDroid?
• Additional memory to store taint tags.
• Additional CPU cost to assign, propagate, check taint tags.
• Overheads seem to be moderate: ~3--5% memory overhead, 3--29% CPU overhead
• However, on phones, users are very concerned about battery life: 29% less CPU performance may be tolerable, but 29% less battery life is bad.

Q: Why not track taint at the level of x86 instructions or ARM instructions?
A: It’s too expensive, and there are too many false positives.
• Ex: If kernel data structures are improperly assigned taint, then the taint will improperly flow to user-mode processes. This results in taint explosion: it’s impossible to tell which state has *truly* been affected by sensitive data.

• One way that this might happen is if the stack pointer or the break pointer are incorrectly tainted.
• Once this happens, taint rapidly explodes:
  o Local variable accesses are specified as offsets from the break pointer.
  o Stack instructions like pop use the stack pointer.
Q: Taint tracking seems expensive—can’t we just examine inputs and outputs to look for values that are known to be sensitive?  
A: This might work as a heuristic, but it’s easy for an adversary to get around it.

- There are many ways to encode data, e.g., URL-quoting, binary versus text formats, etc.

As described, taint tracking cannot detect implicit flows.

- Implicit flows happen when a tainted value affects another variable without directly assigning to that variable.

```c
if (imei > 42){
  x = 0;
}else{
  x = 1;
}
```

- Instead of assigning to `x`, we could try to leak information about the IMEI over the network!
- Implicit flows often arise because of tainted values affecting control flow.
  - Can try to catch implicit flows by assigning a taint tag to the PC, updating it with taint of branch test, and assigning PC taint to values inside if-else clauses, but this can lead to a lot of false positives. Ex:

```c
if (imei > 42){
  x = 0;
}else{
  x = 0;
}
```

- The taint tracker thinks that `x` should be tagged with `imei`’s taint, but there is no information flow!

Interesting application of taint tracking: keeping track of data copies.

- Often want to make sure sensitive data (keys, passwords) is erased promptly.
- If we’re not worried about performance, we can use x86-level taint tracking to see how sensitive information flows through a machine.
- Basic idea: Create an x86 simulator that interprets each x86 instruction in a full system (OS + applications).
- You’ll find that software often keeps data for longer than necessary. For example, keystroke data stays around in:
- Keyboard device driver's buffers
- Kernel's random number generator
- X server's event queue
- Kernel socket/pipe buffers used to pass messages containing keystroke
- tty buffers for terminal apps
- ... etc ...

TaintDroid detects leaks of sensitive data, but requires language support for the Java VM---the VM must implement taint tags. Can we track sensitive information leaks without support from a managed runtime? What if we want to detect leaks in legacy C or C++ applications?

- One approach: use doppelganger processes as introduced by the TightLip system.
  - Ref: https://www.usenix.org/legacy/event/nsdi07/tech/full_papers/yumerefendi/yumerefendi.pdf
- Step 1: Periodically, Tightlip runs a daemon which scans a user's file system and looks for sensitive information like mail files, word processing documents, etc.
  - For each of these files, Tightlip generates a shadow version of the file. The shadow version is non-sensitive, and contains scrubbed data.
  - Tightlip associates each type of sensitive file with a specialized scrubber. Ex: email scrubber overwrites to: and from: fields with an equivalent number of dummy characters.
- Step 2: At some point later, a process starts executing. Initially, it touches no sensitive data. If it touches sensitive data, then Tightlip spawns a doppelganger process.
  - The doppelganger is a sandboxed version of the original process.
    - Inherits most state from the original process...
    - ...but reads the scrubbed data instead of sensitive data
  - Tightlip lets the two processes run in parallel, and observes the system calls that the two processes make.
  - If the doppelganger makes the same system calls with the same arguments as the original process, then with high probability, the outputs do not depend on sensitive data.
- Step 3: If the system calls diverge, and the doppelganger tries to make a network call, Tightlip flags a potential leak of sensitive data.
  - At this point, Tightlip or the user can terminate the process, fail the network write, or do something else.

- Nice things about Tightlip:
  - Works with legacy applications
  - Requires minor changes to standard OSes to compare order of system calls and their arguments
  - Low overhead (basically, the overhead of running an additional process)
• Limitations of Tightlip
  o Scrubbers are in the trusted computing base.
    ▪ They have to catch all instances of sensitive data.
    ▪ They also have to generate reasonable dummy data---otherwise, a
doppelganger might crash on ill-formed inputs!
  o If a doppelganger reads sensitive data from multiple sources, and a
  system call divergence occurs, Tightlip can't tell why.

TaintDroid and Tightlip assume no assistance from the developer ... but what if
developers were willing to explicitly add taint labels to their code?

```c
int {Alice --> Bob} x;  //Means that x is controlled
                        //by the principal Alice, who
                        //allows that data to be seen
                        //by Bob.
```

• Input channels: The read values get the label of the channel.

• Output channels: Labels on the channel must match a label on the value being
  written.

• Static (i.e., compile-time) checking can catch many bugs involving inappropriate
  data flows.
  o Loosely speaking, labels are like strong types which the compiler can
    reason about.
  o Static checks are much better than dynamic checks: runtime failures (or
    their absence) can be a covert channel!

• For more details, see the Jif paper: [http://pmg.csail.mit.edu/papers/iflow-sosp97.pdf](http://pmg.csail.mit.edu/papers/iflow-sosp97.pdf)