Phone users are easily exposed to insecure Wi-Fi.

- Today major servers only allow encrypted communication.
- TLS ensures confidentiality from the middle men.
Threat Model

The attackers record communication of victims.
Later, attackers compromise targets.

Can attackers decrypt the captured communication?

Maybe, if they are lucky to find keys.
TLS Cryptosystem should resist this threat.

Various tactics are used to protect against future compromises.

- Long-term key material
- Short-term key material
TLS Cryptosystem should resist this threat.

Various tactics are used to protect against future compromises.

- Long-term key material: **Perfect forward secrecy**
- Short-term key material

---

Forward Secrecy Support\(^1\)

- **50%** in Feb/2014
- **49%** in Feb/2014
- **6%** in Feb/2018
- **94%** in Feb/2018

\(^1\) [https://www.ssllabs.com/ssl-pulse/](https://www.ssllabs.com/ssl-pulse/)
TLS Cryptosystem should resist this threat.

Various tactics are used to protect against future compromises.

- Long-term key material: **Perfect forward secrecy**.
- Short-term key material: **TLS implementations** have responsibility.
  - OpenSSL goes to great length to clean up ephemeral keys rapidly.

```c
void *OPENSSL_clear_realloc(void *p, size_t old_len, size_t num)
void OPENSSL_clear_free(void *str, size_t num)
void OPENSSL_cleanse(void *ptr, size_t len);
void *CRYPTO_clear_realloc(void *p, size_t old_len, size_t num, const char *file, int line)
void CRYPTO_clear_free(void *str, size_t num, const char *, int)
```
Research Question and Motivation

What about Android?
- Are previous communications safe under memory disclosure attack?

Motivation
1. Threat model is more practical.
Research Question and Motivation

By software exploitations

By physical techniques

- Cold-boot attack
- Nexus 5X bootloader vulnerability

Android has various attack vectors.

1. Threat model is more practical.

- BroadPwn
- Android Bug
- BlueBorne Attack
- MELTDOWN
- SPECTRE
What about Android?

- Are previous communications safe under *memory disclosure attack*?

Motivation

1. Threat model is more practical.
2. Managing secrets on memory would be more challenging.
   - Multiple software layers
   - Complex application lifecycle

*Let’s see how Android TLS deals with those issues.*
Background: Secrets on TLS

TLS Full Handshake
Background: Secrets on TLS

**TLS Abbreviated Handshake**

- **ClientHello** to **ServerHello**
- **ChangeCipherSpec**
- **Finished**

**Application Data**
- **Client Random**
- **Server Random**

**Master Secret**
- **PRF**
- **Key Block**
  - **Session Keys**
  - **MAC Secrets**
  - **IVs**

**Lifetime of Secrets**

**FULL HANDSHAKE**

**APPLICATION DATA**

**ABBR HS**

**APPLICATION DATA**
Black-Box Security Analysis

1. Establishing TLS Connections
2. Logging the keys during the handshake
3. Dumping Android’s memory
4. Searching keys from the memory dump
Black-Box Security Analysis Experiment

Repeating

- Different version: Emulators (Ver 4, Ver 5, Ver 6, Ver 8) and Nexus 5
- Performing additional actions

**Test Framework** *supporting automation*
The results are almost same for all the cases regardless of versions.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Premaster Secret</td>
<td>X</td>
</tr>
<tr>
<td>Master Secret</td>
<td>✓</td>
</tr>
<tr>
<td>Key Block (Session Key)</td>
<td>X</td>
</tr>
</tbody>
</table>

But, Why?
Is this a bug or intended?

Master secrets are found regardless of different actions.
- Moving apps to background.
- Forcing garbage collection.
- Killing apps.

Developers cannot control this retention.
In-depth Analysis
Android TLS Stack

Applications
- TLS Application
- HTTPS Application

Frameworks
- Android JSSE Interface

Libraries
- Conscrypt
- OkHttp
- BoringSSL

Java
C
Problem: Inconsistency in object management

- **Applications**
  - TLS Application
  - HTTPS Application

- **Frameworks**
  - Android JSSE Interface

- **Libraries**
  - BoringSSL
  - Conscrypt
  - OkHttp

- **Languages**
  - Java
  - C

- Issues:
  - Ref. Counting
  - Lazy Deletion
  - Eager Deletion
BoringSSL/OpenSSL: Reference Counting

- Each structure has reference count field.
- Objects are correctly freed when their reference count is zero.
- All key materials are managed within BoringSSL.
Conscrypt: Lazy Deletion

- Corresponding classes one-to-one mapped with the BoringSSL structures.
- On creation, OpenSSLSessionImpl increasing the ref. count of its underlying object.

But, no more manual reference management.

OpenSSLSocketImpl

OpenSSLSessionImpl

SSLParametersImpl

struct SSL_CTX

struct SSL

struct SSL_SESSION
Conscrypt: Lazy Deletion

Problem 1: Dependence on JVM’s Automatic Memory Management.
- Clean-up timing is undefined.

What if TLS apps are going to background?
What if other objects hold this object unnecessarily long?

When GC is triggered

What if TLS apps are going to background?
What if other objects hold this object unnecessarily long?
Problem 2: **Session Cache**’s LRU replacement policy

- No explicit eviction routine. Expired OpenSSLSessions are still in the cache.

---

What if TLS apps are no longer used?

When GC is triggered

---

Master Secret
Problem 3: **Static Singleton objects** are connected to them.

- Their lifetime is same as the application. No way to release them.
OkHttp: Eager Deletion

OkHttp manages Singleton Connection Pool

- Good thing: eagerly delete with **Cleanup Thread**.

*But, its effort is useless in removing master keys.*

When GC is triggered
What is the consequence of the problem?

- Each TLS application holds some number of master secrets whether they are expired or not.
Evaluation of Attack Feasibility
Can attackers exploit this problem in practice?

1. Is an attacker able to find 48 bytes of keys in a reasonable time?
   - Yes. We found the pattern.
   - Simple tool finds master secrets in several seconds.

2. How long does master keys live in memory with real-world apps?
   - Additional experiment with Chrome application.
## Evaluation of Attack Feasibility

How long does master key live in memory?

### Result with Chrome application

<table>
<thead>
<tr>
<th>Time (Hour)</th>
<th>Event</th>
<th># of Found Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Access five web sites</td>
<td>51</td>
</tr>
<tr>
<td>1</td>
<td>Move the app to background</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>Run YouTube application</td>
<td>42</td>
</tr>
<tr>
<td>…</td>
<td>Keep playing movies</td>
<td>…</td>
</tr>
<tr>
<td>2</td>
<td>After 2 days</td>
<td>38</td>
</tr>
</tbody>
</table>
Evaluation of Attack Feasibility
How long does master key live in memory?

Result with Chrome application

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th># of Found Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Access five web sites</td>
<td>51</td>
</tr>
<tr>
<td>1</td>
<td>Move the app to background</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>Run YouTube application</td>
<td>42</td>
</tr>
<tr>
<td>...</td>
<td>Keep playing movies</td>
<td>...</td>
</tr>
<tr>
<td>51</td>
<td>After 2 days</td>
<td>38</td>
</tr>
</tbody>
</table>

Most of master secrets are preserved as long as the app is alive.
What if attackers access Android memory of the targeted victim?
We implemented two solutions.

1. Hooking Android lifecycle
   - Clean up expired keys when applications are going to background.

2. Eager Deletion: Sync with OkHttp
   - Run secondary thread to evict expired TLS sessions.

Two modest patches can mitigate this problem.
Reporting to Google

- Reported the issue with the patches in Nov 2017.
- Recently, we received the feedback.

<table>
<thead>
<tr>
<th>status:</th>
<th>Assigned → Infeasible</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASR Severity:</td>
<td>Moderate → NSBC</td>
</tr>
</tbody>
</table>

...we don't consider deleting information from the application's memory fast enough to be a security issue ...

But, we believe expired master secrets should be deleted.
Conclusion

We first investigate Android TLS in terms of managing ephemeral keys.

Android retains master secrets because of conflicting memory models.

- Impact on all applications using standard TLS APIs.
- Impact on all Android versions we examined from Android 4 to 8.
- Our forensics tools show that it is exploitable practically.

We suggest the practical solutions.
Thank you!

Jaeho Lee
PhD student, Rice University

Contact: Jaeho.Lee@rice.edu
Web: https://cs.rice.edu/~jl128
Analysis Framework
SSL_SESSION Structure

```c
struct ssl_session_st {
    int ssl_version;  // 0x0301~0303
    int master_key_length;  // 0x30
    uint8_t master_key[SSL_MAX_MASTER_KEY_LENGTH];
    unsigned int session_id_length;  // 0x20
    uint8_t session_id[SSL_MAX_SSL_SESSION_ID_LENGTH];
    ...
}
```
Discussion

Conscrypt (Java) vs BoringSSL (C)
- Conscrypt: effective Java coding
- BoringSSL: isolated secret management

Conscrypt (TLS Session Cache) vs OkHttp (HTTP Connection Pool)
- Different perspective dealing with underlying objects
  - OkHttp: Eagerly eviction with Timer
  - Conscrypt: No explicit eviction

Bad Programming Pattern: Singleton object + Dependence on GC
- Singleton object + Dependence on GC for critical routines
Methodology

Black-Box Security Analysis → White-Box Security Analysis
Research Question and Motivation

Android has various attack vectors.

By software exploitations

- BroadPwn
- BlueBorne Attack
- MELTDOWN
- SPECTRE

By physical techniques

- Cold-boot attack
- Nexus 5X bootloader vulnerability