# 組み込みアプリケーションの実装における セキュリティ上の課題

日本シノプシス合同会社
松岡　正人

組み込み機器開発は、Linux や Windows などの汎用の OS を利用することでパソコン化し、開発の容易さを手に入れた一方でシステムが肥大化し品質を担保するのが困難になり、出荷後にハードウェアの統合や、ネットワーク処理で問題が明らかになることが後を絶ちません。本記事では、実際の実装上の問題によって発生した脆弱性の調査をもとに、典型的な課題について振り返ってみたいと思います。

## 序文

現在、ネットワーク接続型の組み込み機器の多くは Linux OS を採用しているか、必要なネットワーク接続のための機能を有する組み込み OS を採用しており、Windows、VxWorks、QNX など古くから使われている汎用の組み込み OS もあれば、Apple 社の iOS のように特定の端末のために開発された OS もある。さらに、データベース、ウェブ、オンライン・ゲーム、決済などを利用するためのフレームワークやライブラリ、ウェブサービスの API などを組み合わせて利用することで様々なアプリケーションを提供している。そして個々のコンポーネントが内包する脆弱性は、開発提供元の各ベンダーが対処するのが常だが、それらを統合するためのプログラムコードは利用するコンポーネントのガイドラインに則って書き、単体テストやシステムテスト、異常系テスト、場合によってはファジングやペネトレーションテストを行うことで、開発者が担保する必要がある。

ところが、なんらかの事情で適切なテストが行われることなく市場に出荷されるコードも少なくない。この記事で取り上げる事例も、個々のコンポーネント、OS やデバイスドライバー、ライブラリーやフレームワークなどの脆弱性ではなく、不適切なコードによる予期しなかった脆弱性についてである。

弊社のセキュリティ・リサーチャのロンドンチームが今年の 4 月に公開した、CVE-2020-7958 biometric data extraction in Android devices、というレポートがある。調査したメンバーと会話した結果を織り交ぜながら、ソフトウェア開発の厄介な点についてハイライトできればと思う。

## 調査対象のデバイス：

対象デバイスは 2019 年に登場した OnePlus 7 Pro、このモデル以外の調査は行っていない。少人数で短期間に実施するためだ。中国製のこの端末は純粋な Android 端末ではなく、Android をベースに OnePlus が独自に手を加えている OxgenOS を採用している。2015 年に最初のリリースが行われてから UI だけでなく様々な改良を行っているが、基本的なオリジナルの Android と構造は同じである。ちなみに、このモデルを選択したのは、最初からハイエンドの機種（機能が一番多い）の脆弱性を調べるほうがいい（何か発見できる可能性が高い）だろうと考えたからだ。

## ARM Trust Zone と Qualcomm QSEE によるシステムの保護：

ARM 社はセキュアな OS の実行環境のために Trust Zone という仕組みを提供している。これは半導体レベルでセキュアなソフトウェアの実行を実現するためのもので、ブートローダーがセキュアかどうかを判定し、セキュアな OS の起動を行い、指紋や顔認証などの生体認証などの仕組みを OS やアプリケーションか

ら論理的に分離することができる。OnePlus 7 Pro も Trust Zone を採用している。

Trust Zone ではハードウェアがリセットされると、下図の手順で順次必要なモジュールを読みだして起動していく。最初はブート ROM が Trusted Boot Firmware を起動し、TEE（Trusted Execution Environment）というセキュアなメモリ空間と REE（Rich Execution Environment）という非セキュアなメモリ空間とに分けて管理することで、安全に動作させたいコードを「Secure World」内で保護することができる。

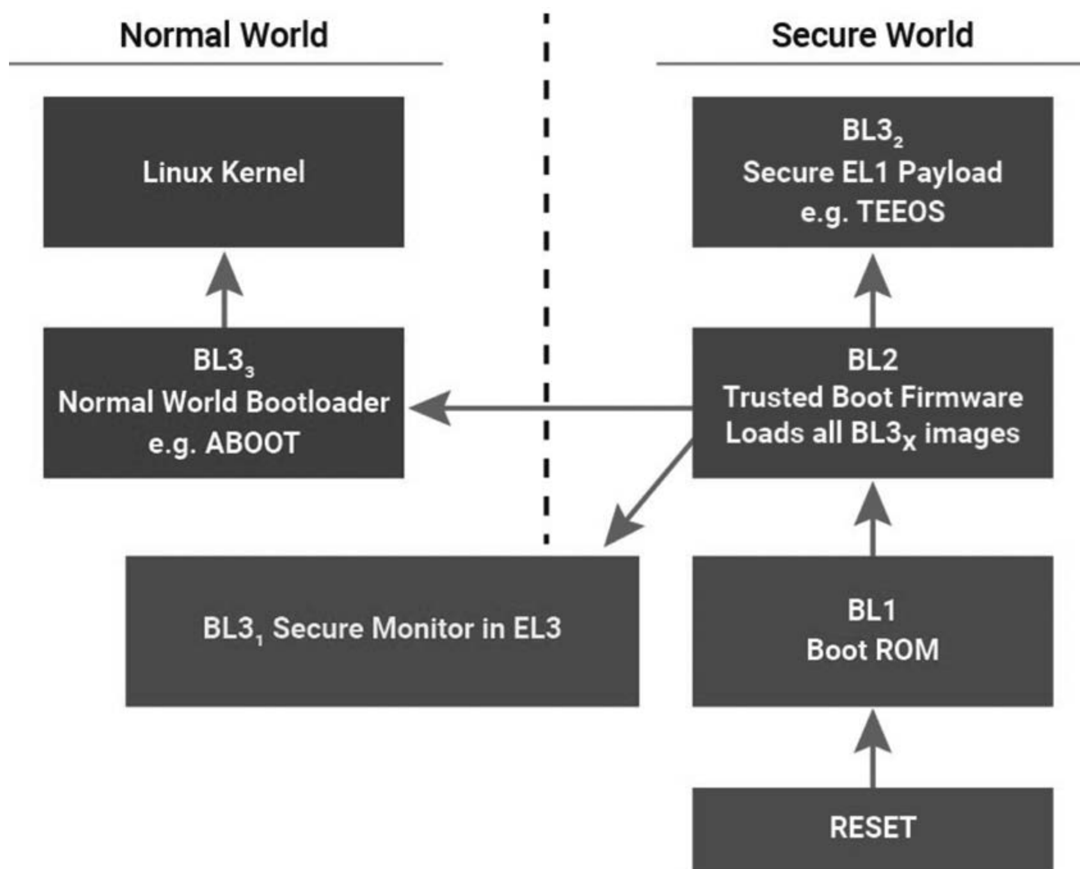


Fig.1" Trust Zone におけるブート・シーケンス"

また、Trust Zone では TEE の実装はいくつかの選択肢があるが、OnePlus 7 Pro では Qualcomm 社の QSEE（Qualcomm Secure Execution Environment）が実装されており、Trust Zone で構成されたシステムは以下のような構成になっている。

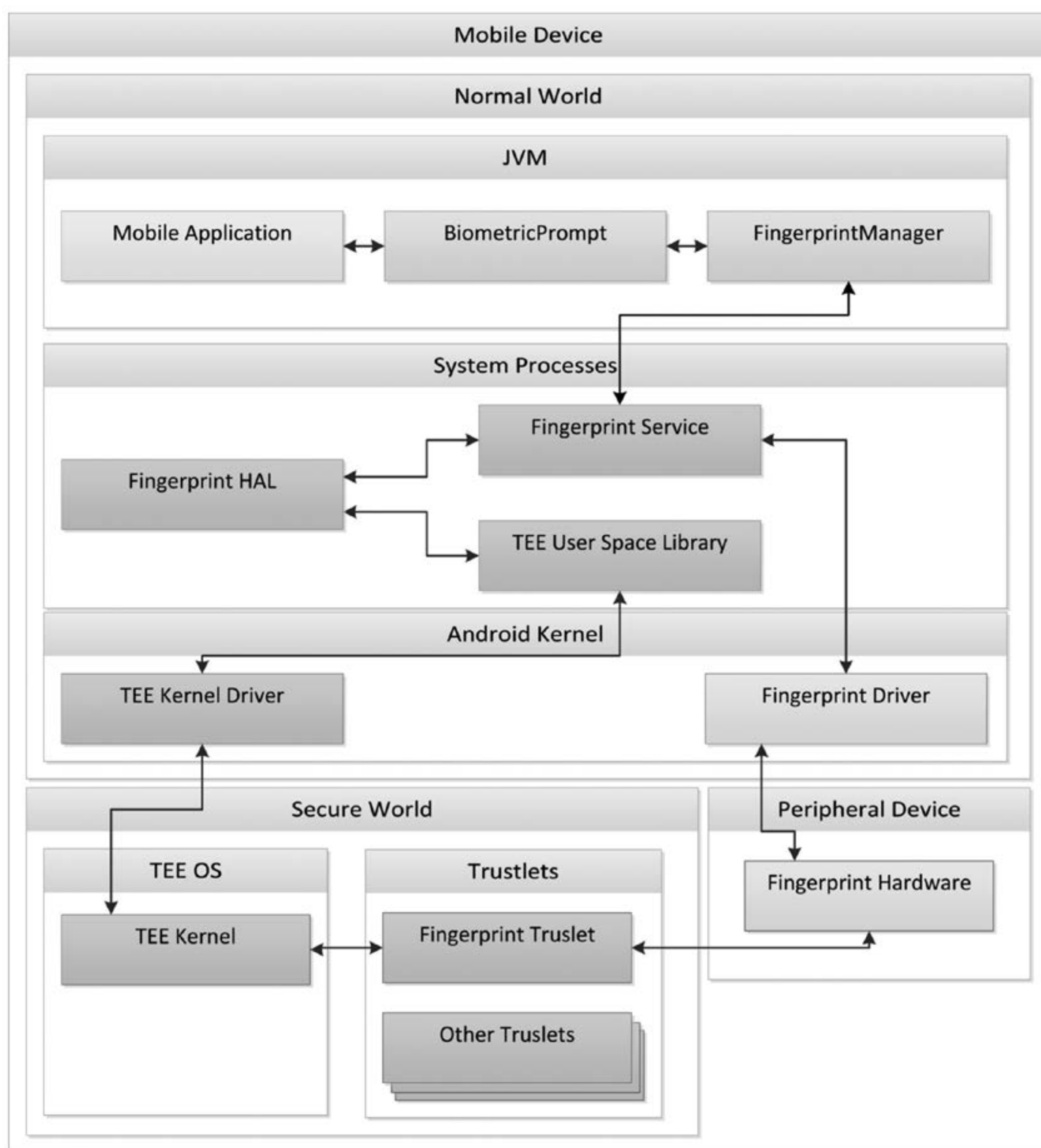


Fig.2:” Trust Zone によるシステム構成図”

TEE は“Secure World”と書かれた部分で、Android（OxygenOS）とは論理的に分離されているのがわかる。
さて、本稿で述べる脆弱性は、Secure World 内で指紋センサー（Fingerprint Hardware）で取得した指紋の画像データを認証するために TEE 内の Fingerprint Trustlet で処理する際のものであるが、指紋の画像データは REE（Fig.2 の Android Kernel も含む Normal World のブロック）内に渡されることがないようにしなければならない。これは Android の開発者向けのガイド（https://source.android.com/security/authentication/fingerprint-hal）にも”指紋認証 HIDL”の項目で次のように注意書きがある。

- ベンダー固有の HAL 実装では、TEE で必要な通信プロトコルを使用する必要があり、未処理の画像と処理済みの指紋の特徴は、信頼できないメモリに渡さないこと
- このような生体認証データはすべて、TEE などのセキュア ハードウェアに保存する必要がある
- 従って、ルート権限取得によって生体認証データが侵害されないようにする必要がある

もちろん、OnePlus 7 Pro の指紋センサーと認証の仕組みは上記に則って実装されていた。

解析を始め、最初に取っつきやすい REE 側の指紋センサーのコンポーネントを調べてみた。libgf_ud_hal.so という共有オブジェクトが REE の指紋認証用のサブシステムに含まれており、ルート化された端末の /vendor/lib64/ に見つけることができた。そこで、OnePlus のダウンロードサイトから img ファイルを入手して調べてみると /vendor/lib64/ とファイル群をみつけることができた。libgf_ud_hal.so には goodix::SZCustomizedProductTest::factoryCaptureImage() というメソッドが含まれていた。ここから先は根気のいるリバースエンジニアリングを伴う作業の繰り返しとなり、ようやく擬似コード（Pseudocode）を生成するところまでたどり着いた。なお、これらの作業は root 化し、USB デバッグモードでホスト PC のデバッグ環境と接続してあったことを付け加えておく。

```
__int64 __fastcall goodix::SZCustomizedProductTest::factoryCaptureImage(goodix::SZCustomizedProduct
{
  goodix::command::FactoryCaptureImage *Command; // x0 MAPDST
  unsigned int rv; // w21
  const char *errfmt; // x0

  if ( raw_data_out )
  {
    Command = malloc(0x1502Cu);
    if ( Command )
    {
      memset(Command, 0, sizeof(goodix::command::FactoryCaptureImage));
      Command->ae_expo_start_time = ae_expo_start_time;
      Command->field_1C = uchar;
      Command->field_1E = ushort2;
      Command->Parent.target = 1003;
      Command->Parent.cmd_id = 17;
      rv = goodix::HalBase::invokeCommand(&this->HalBase, &Command->Parent, 0x1502C);
      if ( !rv )
      {
        memcpy(raw_data_out, &Command->captureImageResponseBuffer, sizeof(GF_SZ_TEST_RAWDATA));
        free(Command);
        return rv;
      }
      free(Command);
    }
    else
    {
      __android_log_print(6, "[GF_HAL][SZCustomizedProductTest]", "[%s] out of memory, cmd", "facto
      rv = 1001;
    }
  }
  else
  {
    __android_log_print(6, "[GF_HAL][SZCustomizedProductTest]", "[%s] param is erro", "factoryCaptu
    rv = 1004;
  }
  errfmt = gf_strerror(rv);
  __android_log_print(
    6,
    "[GF_HAL][SZCustomizedProductTest]",
    "[%s] exit. err=%s, errno=%d",
    "factoryCaptureImage",
    errfmt,
    rv);
  return rv;
}
```

Fig.3:” 擬似コード goodix::factoryCaptureImage()”

得られた擬似コードから、Target ID（擬似コード内の target）が 1003、Command ID（同じく cmd_id）が 17 とわかったことで、これを使って TEE 内の Fingerprint Trustlet の機能を呼び出すことができるのではないかと想像してみる。このようなコマンドの構造を解き明かしていくと、goodix::HalBase::invokeCommand() を呼び出し、QSEE のライブラリ libQSEEComAPI.so と通信し、指紋画像のデータを格納するメモリーを確保するだろうことが想像できる。

また、SZCustomizedProductTest のインスタンスを生成するには HalContext インスタンスに有効なリファレンスを渡さなければならないが、運のいいことに goodix::HalBase::invokeCommand() が何度も繰り返し呼び出されていたことから指紋センサーのデータを処理に関わっていることは明白で、goodix::HalBase のインスタンスには HalContext への有効なポインターが含まれていることがわかった。これらの作業が厄介なのは、root 化したデバイスでの root のアクセス可能な領域を思い出してもらえればわかると思うが、指紋センサーの画像データを処理して認証する仕組みは TEE（Fig.1 の Secure World）内の Trustlet として実装されているため REE（Fig.1 の Normal World）から直接アクセスすることができず、TEE 内の実装がどのようになっているのかは、リバースエンジニアによって得た少ない証拠を積み重ねていくことしかできないからである。

しかし、擬似コードを得ることができた為、あとは Trustlet の機能と、初期の解析で獲得した target=1003、cmd_id=17 のパラメータを利用した Trustlet をエクスプロイトするコードの作成ということになる。

詳細は本文最後に記載した URL にあるブログ記事本文を参照願うとして、エクスプロイトコードは完成し、機能した。REE 内の指定したメモリー領域に指紋センサーのデータをコピーすることができたのである。指紋センサーと指紋認証が動作することをテストするためのコードが Trustlet 内に残っていたためである。



## 対策と振り返り：

テスト用のコードを削除する、つまり cmd_id=17 を受け取ってテストするコマンドハンドラーを取り去ることで解決したが、#ifdef によって開発時の指紋センサーの試験時にはこのコマンドハンドラーを活かし、出荷用のコードはコマンドハンドラーを省けば良いだけである。これがたんなる不注意によるものなのか、開発者にとっては留意すべきはこの一点だろう。現時点で有効な高度なセキュリティ技術だが、やはりシステム全体、各コンポーネントや基盤技術をサポートした安全なソフトウェアの開発の脆弱性はコードの中に潜んでいたのである。

なお、本調査に際して、速やかに対応していただいた OnePlus 社の関係者には謝意を申し上げる。

参照先 URL
英　語：https://www.synopsys.com/blogs/software-security/cve-2020-7958-trustlet-tee-attack/
日本語：https://www.synopsys.com/blogs/software-security/ja-jp/cve-2020-7958-trustlet-tee-attack/