TERRACOTTA Android Malware: A Technical Study

whiteops.com/blog/terracotta-android-malware-a-technical-study

The White Ops Satori Threat Intelligence & Research team has been actively defending against an ad fraud botnet—which we've codenamed TERRACOTTA—since late last year. Today we are revealing the technical details of the malware campaign in an effort to broaden awareness. As we stated in our blog post, i**n a single week in June 2020, the operation generated more than two *billion* fraudulent bid requests, infected upwards of 65,000 unwitting devices, and spoofed more than 5,000 apps.**

A spokesperson from Google stated, “Due to our collaboration with White Ops investigating the TERRACOTTA ad fraud operation, their critical findings helped us connect the case to a previously found set of mobile apps and to identify additional bad apps. This allowed us to move quickly to protect users, advertisers and the broader ecosystem – when we determine policy violations, we take action.”

The TERRACOTTA malware offered Android users free goods in exchange for downloading the app—including shoes, coupons, and concert tickets—which users never received. Once the app was installed and the malware activated, the malware used the device to generate non-human advertising impressions purporting to be ads shown in legitimate Android apps. This technical report goes into detail on the TERRACOTTA malware, covering the initial application code, subsequent payload activation via its Command and Control (C2) server, and the mechanism through which the advertising fraud was executed.

Initial Download: Free goods app, with obfuscated React Native module

The initial download for TERRACOTTA apps is straightforward. The main application code (i.e. the APK) is written using the React Native cross-platform development framework and just renders a form that the user fills in to receive their ‘free’ goods. This initial part of the app contains no malicious functionality. However, the underlying maliciousness of the app is already hinted at in its permissions (which need to be specified in the APK at compile time): they include *permission.WAKE_LOCK* and *permission.FOREGROUND_SERVICE,* permissions the Satori team typically observes in ad fraud malware that runs continuously on a user device.

Another eyebrow-raising snippet of code further consolidates the assumption that the app is more than meets the eye. Most of the strings inside the app are obfuscated behind an algorithm specifically intended to thwart malware analysis. At run-time when a string is decoded, the algorithm analyzes its own call stack and is designed to fail when it identifies a modified stack - which is what happens when an analyst tries to hook the decryption routine (see below).

```java
public static String access$200(Object obj) {
    StackTraceElement stackTraceElement = new LinkageError().getStackTrace()[1];
    StringBuffer stringBuffer = new StringBuffer(stackTraceElement.getMethodName());
    stringBuffer.append(stackTraceElement.getClassName());
    String stringBuffer2 = stringBuffer.toString();
    int length = stringBuffer2.length() - 1;
    String str = (String) obj;
    int length2 = str.length();
    char[] cArr = new char[length2];
    int i = length2 - 1;
    int i2 = length;
    while (i >= 0) {
        int i3 = i - 1;
        cArr[i] = (char) ((str.charAt(i) ^ stringBuffer2.charAt(i2)) ^ '1');
        if (i3 < 0) {
            break;
        }
        int i4 = i3 - 1;
        i2--;
        cArr[i3] = (char) ((str.charAt(i3) ^ stringBuffer2.charAt(i2)) ^ 'U');
        if (i2 < 0) {
            i2 = length;
        }
        i = i4;
    }
    return new String(cArr);
}
```

(click on any image in this report to enlarge)
Figure 1. Deobfuscation routine used for one of the strings inside the initial APK. The algorithm uses an element in its own stacktrace as a pseudo-decryption-key, meaning that when the function is hooked, the stack trace changes and the resulting output of the deobfuscation routine is incorrect.
Source: White Ops Satori Threat Intelligence & Research Team

The relevant TERRACOTTA code and its hidden functionality is found in a file named ***index.android.bundle*** within the ***resources*** directory of the app. This file mostly contains benign React Native modules which are commonly found in legitimate apps and provide the core capabilities of a React Native app. However, one of the modules stored here contains heavily obfuscated code. Analysis shows that this module handles C2 communication, which is achieved through the use of the messaging capability of Firebase, a widely used mobile app platform.

The use of Firebase push-messaging as a C2 is of particular note because - as information about the device is uploaded to Firebase on installation, even for legitimate use cases - it gives the attacker a way to understand their install base (and potentially exclude certain hosts from downloading subsequent payloads) without writing bespoke code to exfiltrate information from the infected device. A push-messaging setup also means that the app doesn’t need to frequently poll the C2 to check for updates, another standard sign of malware.

```
z = function z() {
    _0x3fe2('0x22', 'Qw]2') !== _0x3fe2('0x23', '5ra8') ? new Promise(function (_r52) {
        _0x3fe2('0x24', 'X*la') === _0x3fe2('0x25', 'QvwT') ? null != _reactNativeUuid.default[_0x3fe2('0x26', 'N7R7')] ?
            _0x3fe2('0x32', 'lI*S') === _0x3fe2('0x33', 'sK$h') ? (null == _reactNative.default[_0x3fe2('0x34', '*#3n')] &
                eval(c(_0x3fe2('0x3f', 'rRwK'), m[_0x3fe2('0x40', 'C8w4')].id, 1)[_0x3fe2('0x41', 'z0mk')]())
            })(_reactNative.default, _reactNativeFirebase.default, _reactNativeUuid.default[_0x3fe2('0x42', 'SqF3')], _0x3
        })[_0x3fe2('0x4d', 'GNlV')](function (x) {
            if (_0x3fe2('0x4e', 'N7Mb') === _0x3fe2('0x4f', '*PU)')) return null;
            _r52()
        }) : _r52[_0x3fe2('0x50', 'SqF3')] > 0 && _reactNative.default[_0x3fe2('0x51', 'lI*S')][_0x3fe2('0x52', '5ra8')]
    })[_0x3fe2('0x62', 'ztXP')](function () {
        if (_0x3fe2('0x63', 'Qw]2') === _0x3fe2('0x64', 'lu)U')) return _reactNative.default[_0x3fe2('0x6c', 'lI*S')][_0x3
            if (_0x3fe2('0x6e', '*#3n') !== _0x3fe2('0x6f', 'B1sZ')) return null;
            _reactNative.default[_0x3fe2('0x70', 'N7Mb')] = {}, _reactNative.default[_0x3fe2('0x71', '8saw')].f = {}, _rea
        });
        _reactNativeUuid.default[_0x3fe2('0x65', 'v0aI')][_0x3fe2('0x66', '&b8Y')] = t, _reactNativeFirebase.default[_0x3
    })[_0x3fe2('0x7a', 'A[cJ')](function (rf) {
        if (_0x3fe2('0x7b', 'sK$h') === _0x3fe2('0x7c', 'N7Mb')) {
            if (_reactNativeUuid.default[_0x3fe2('0x7d', 'SGft')]) {
                if (_0x3fe2('0x7e', 'QvwT') === _0x3fe2('0x7f', 'B1sZ')) return null;
                _reactNativeUuid.default[_0x3fe2('0x80', 'lI*S')] = {}, _reactNativeUuid.default[_0x3fe2('0x81', 'Qw]2')]
                    _0x3fe2('0x8f', 'Qw]2') !== _0x3fe2('0x90', 'QvwT') ? (_reactNativeUuid.default[_0x3fe2('0x91', 'wrR5
```

Figure 2. Screenshot of obfuscated React Native code responsible for communicating with the malware's C2 and loading further fraud modules.
Source: White Ops Satori Threat Intelligence & Research Team

Besides establishing the C2 channel with Firebase, the most notable feature of this obfuscated module is the presence of multiple *eval* JavaScript statements. These statements —artifacts of dynamic code execution--are responsible for loading further malicious modules pushed to the device and executing them as part of the React Native app.

The ad fraud capability is activated via a push message from Firebase that contains a further React Native module—named RNVlCore—which is obfuscated similarly to the previous one and appears to do two things:

- Provide a base platform for any subsequent malicious modules that are downloaded. The main responsibility of the base platform is to ensure that any exceptions or errors thrown by subsequent modules are caught and suppressed without resulting in any notification to the device's user.
- Transfer further C2 responsibility away from Firebase to a different C2 server, used for downloading the ad-fraud-specific related modules.

Notable in the RNVlCore module sent by Firebase is the occurrence of a package name, starting with **com.viking** that features consistently in all of the modules downloaded thereafter, indicating a solid link between the threat actor controlling Firebase and the developer of the subsequent modules downloaded from elsewhere.

```
    return this[_0x3b93('0x6f')]('com.viking.RNVlCoreAnalyticsPackage').catch(function (_0x28b351) {});
} [_0x3b93('0x15')](this))[_0x3b93('0x18')](function (_0x46a8bb) {
    this.coreModuleLoaded = ![];
```

Figure 3. Screenshot of source code from the RNVlCore, referencing a Java class in the same package.
Source: White Ops Satori Threat Intelligence and Research

Activation: Ad Fraud Module and supporting modules

After the initial setup mentioned above, the main module focused on performing ad fraud loads, which we call the Looper module. It consists of a main entry-point, starting a series of concurrent loops which are responsible for requesting tasks from their specific C2 endpoint and executing the tasks that are returned (visually represented in the Chart 1 Flow Chart). The table below shows the functionality of each task type as well as any conditions that are applied before initiating them:

Task	Conditions	Task Description
Pop	3 days since install Requests tasks from C2 only from 2pm to 10pm Runs once a day	If conditions allow, the received URL attempts to load. The URL can be a deep link too (i.e. to open the play store, other OS apps, or a third party app that registered a URL schema).
Push	N/A	Logic deactivated.
Web	None	An invisible custom webview of 0x0 size is spawned and configured as specified by the C2, including user agent, origin, shared resources, event listeners, and JS injections. The received task URL is loaded and runs in the webview for a task defined amount of time (30 seconds by default).

Banner	None	An invisible webview of the size specified by the C2 is requested from the webview manager (spawned as necessary) and configured as specified, including user agent, origin, spoofed app header, shared resources, event listeners, blocked domains/resources, and JS injections. The task HTML is loaded in the webview (no network requests) with a hardcoded referer (loopme[.]com), and potentially clicked on. A new cycle is started after 30 secs or the time specified by the previous task received from C2.
Video	None	An invisible webview of the size specified by the C2 is requested from the video player module and configured as specified (including user agent, origin, shared resources, event listeners (including VAST events), blocked domains/resources, and JS injections). The task URL is loaded in the webview and potentially clicked on. The Video ad is run using Loopme video SDK, which is present in the module logic.
Feed	None	An invisible webview is requested and configured as specified by the C2 (including user agent, event listeners). The feed task includes several URLs which are loaded in order: icon, image, pixel. If the task includes an additional URL, it's loaded and left to run for some time.

Table 1. Functionality of main ad fraud Looper module
Source: White Ops Satori Threat Intelligence and Research

Each task type has a set of modules on which it depends, and is responsible for checking that those dependencies are satisfied before running. For instance, the *Banner* task, responsible for the majority of the fraud, is dependent on the WebViewManager module, a module that provides functions for manipulating and controlling a series of customized webviews loaded on the device.

Modules that support the various task types above are downloaded as self-contained APK binaries and loaded by the “RNVICore” base module.

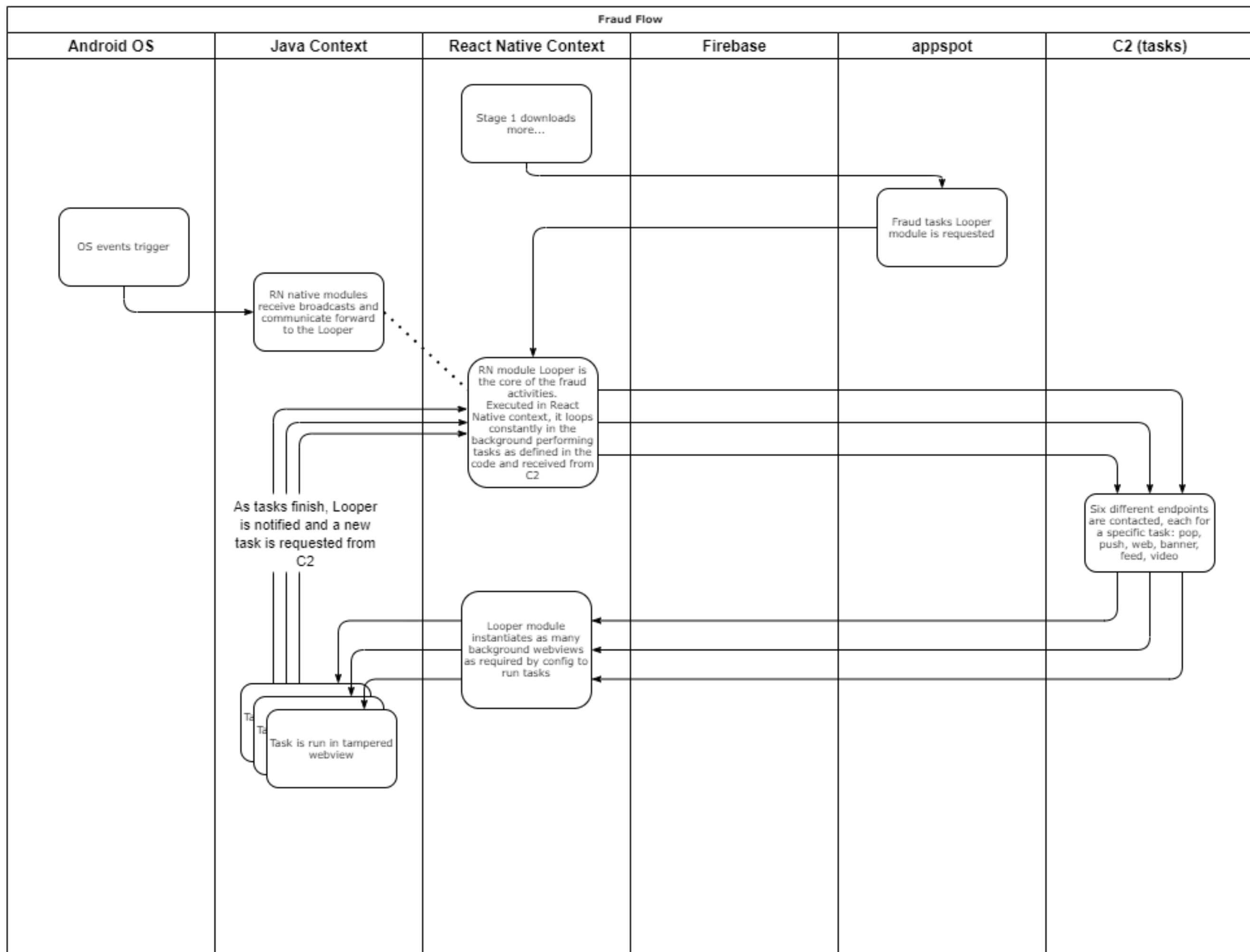

Chart 1. Infection chain flow chart.
Source: White Ops Satori Threat Intelligence and Research

Below is a full list of these modules:

Webview management	This module is extensive. It manages the download and updating of a specialized webview from a location controlled by the threat actor. (see below for In-depth: WebView)
Networking management	Broadcast listeners are installed to monitor for connection changes, and also pulls data from the isActiveNetworkMetered() system Android API.
Screen management	Broadcast listeners are installed to monitor for device locking, screen on and off, and “user present” broadcasts.

Battery management	Broadcast listeners are installed to monitor for battery level changes, charger connection, and disconnection.
Service management	Manages a foreground service using lots of reflection, implements a known hack, named meta-reflection, to access hidden API methods in Android 7+ (flipping the setHiddenApiExemptions setting to remove restrictions). Service initiation is done by hooking MESSAGING_EVENT from Firebase, and by setting timer events in React Native. The service is initiated by being hooked to a notification activity.
Intent management	This module has capabilities to collect the victim's data (including phone number, voice messaging number, and installed apps), perform actions on the device (including opening the camera app, opening device settings, and sending SMS messages) and even sharing images to social media networks (Instagram and Line - an app developed by Naver).
Push notification management	Has methods that allow it to download an icon and push a notification with it.
Video management	This package includes a full video ads SDK and the logic necessary to manage it. The video player includes logic to manage VAST video playing and tracking requests. This module deploys the video player in a webview and includes logic to interact with the webview for fraud actions. These actions include clicking, modifying outbound requests, and blocking code from certain domains from being loaded.

Table 2. Various modules loaded by base module.
Source: White Ops Satori Threat Intelligence and Research

In addition to these specific modules, the Satori team observed the download of two executables as part of TERRACOTTA's activation. These are downloaded in .so native library format and are SOCKS proxies. Our team didn't observe ad fraud activity specifically from these proxies, though ad fraud activity often uses proxies. Our assessment is these proxies are registering the infected device as part of a residential / mobile proxy network which is a common monetization mechanism for developers with mobile installs.

In-depth: Webview Module & customized webview binary built for Fraud

Of all the modules downloaded as part of TERRACOTTA, the package with the most complete functionality is the **Webview manager module**, which we examine in detail in this section.

Like the other modules, the webview manager is downloaded as a React Native module, a full APK, as shown below.

Figure 4. Screenshot of the decompiled source code of the webview manager module. As with all of the modules downloaded by TERRACOTTA, it takes the form of an APK and code is located under the package com.viking.
Source: White Ops Satori Threat Intelligence and Research

In addition to the Webview manager module, a customized **chromium APK** is independently downloaded (from an independent server), unpacked and loaded for the Webview Manager module to use. This level of modularity is another indicator that multiple threat actors are probably involved in its development, and the level of customisation present in the webview binary itself showcases the sophistication of the operation.

The Webview Manager module manages a collection of webviews, which are spawned from the chromium APK. The manager will set spoofed values in those webview instances using the custom logic in the Chromium webview binary.

In order to replace the default webview on the user’s device, the Webview management module overrides internal Android API restrictions by using a hack known as meta-reflection. The module will also enable unencrypted HTTP traffic using the **setCleartextTrafficPermitted** method in the network security policy class.

The webview management module exposes many functions, which can be broadly categorized as follows:

- **Resource Control**
  browserSetMaxAmount() and *webViewSetMaxAmount()* control the number of webview instances which can be activated in parallel, helping to ensure the malware doesn’t use too much of the infected device’s resources and become unusable by the device owner. There are two functions because the number of webviews and the number of ‘browsers’ (webviews disguised as mobile browsers) are controlled independently. Also notable is *webViewSetPreset()* which allows blocking of all content from a specific domain. This allows the malware to avoid executing ad verification services’ code, referred to as tag evasion.

- **Javascript to Native Commands**
  The Javascript code running in the webviews (injected or loaded from remote servers) communicates with the native modules using a custom mechanism. The webview will listen to JavaScript alert() notifications, and look for a prefix in the text of the alert message. If the message starts with the prefix, then the rest of the message is treated as a command. The observed commands were: **close** and **tap**.
- ***close*** instructs the Looper fraud module to dispose of the webview. This action is used when something went wrong in the webview JS context and the impression did not work as expected.
- ***tap*** is used to request a click action on an element, passing the coordinates of where the element is.

```
function t(e) {
    alert("$_V1K1NG_$:" + JSON.stringify(e))
}

function o() {
    t({
        cmd: "close"
    })
}
```

Figure 5. Javascript executed in the webview including the prefix for native code to receive and act. Function wrapping the close command.
Source: White Ops Satori Threat Intelligence and Research

```java
public boolean onJsAlert(@NonNull WebView webView, @NonNull String str, @NonNull
final String str2, @NonNull final JsResult jsResult) {
  final RNVlWebView rNVlWebView = (RNVlWebView) webView;
  if (rNVlWebView.mIsMakedBrowser) {
    int randomInt = randomInt(500, 3000);
    if (str2.startsWith("$_V1K1NG_$:") || str2.equalsIgnoreCase("_G0B@byGo_")) {
      randomInt = 0;
    }
    this.mHandler.postDelayed(new Runnable() {
      public void run() {
        jsResult.confirm();
        new Handler(Looper.getMainLooper()).post(new Runnable() {
          public void run() {
            try {
              if (str2.startsWith("$_V1K1NG_$:")) {
                JSONObject jSONObject = new JSONObject(str2.substring(11));
                String string = jSONObject.getString("cmd");
                if (string.equalsIgnoreCase("tap")) {
                //... code omitted for brevity
              }
            } catch (Exception e) {
            }
          }
        });
      }
    }, (long) randomInt);
  } else {
    jsResult.confirm();
  }
  return true;
}
```

Figure 6. Native code in Webview Module receiving the onAlert messages, checking for the prefix and triggering the requested commands (close and tap).
Source: White Ops Satori Threat Intelligence and Research

- **User interaction and Navigation**
  The module contains various ways of opening URLs, such as *webViewNavigateByUrl()* and *webViewNavigateByData()* as well as faking clicks using *webViewClick(),* which sends click actions (complete with X,Y coordinates) received from the main module's fraud code to the actual webview instance.
- **Parameter modification to simulate multiple devices**
  The module exposes functions to get and set the webview's user agent parameter (which is often used by third parties to determine the type of device) as well as to clear cookies browsing data. This means TERRACOTTA has fine-grained control of each of the webviews created and can ensure that they appear to be from a variety of devices as opposed to all loaded on a single device.

Operation: Generating Fraudulent Ad Traffic

While much of the traffic generated from an infected device is directed towards legitimate advertising networks for the purposes of monetization, some network traffic was suspected by White Ops analysts to have purposes other than acquiring and manipulating advertising networks.

A domain was observed through the network traffic of the infected devices with behaviors that suggested it was the C2/task server for TERRACOTTA's ad fraud module including hosting the customized Webview and requesting the /banner endpoint.

In addition to using parameters specified directly by the C2, the Webview management module selects certain parameters for the fake traffic it creates. For instance the HTTP User-Agent is generated from a template, with one of a set of specified Chrome versions selected at random and inserted into the user-agent for each ad request.

```
    this.chrVer       = ['80.0.3987.132', '79.0.3945.147', '78.0.3904.130'];

    this.emitter.on('moduleLoaded', function(moduleInfo) {
        switch (moduleInfo.name) {
--
module.prototype.init = function() {
    if (this.pushModule && this.inited == 0) {
        this.inited = 1;
        return this.pushModule.getData().then(function(initData) {
            this.data.device.wvUserAgent = initData.ua;
            this.data.device.userAgent   = initData.ua.replace(/\;\ wv/g,'').replace(/ Version\/4\.0/g,'');
            this.data.device.locale      = initData.locale;
            this.data.device.packageName = initData.pn;
            this.data.device.tag         = this.coreContext.react.vCore.core.hashMD5(this.data.device.packageName);
        }.bind(this)).then(function() {
            return Promise.all(Object.values(MODULE_TYPE).map(function(type) {
--
    }
};

module.prototype.getUserAgent = function(isBrowser, changeDevice) {
    if (this.realUA) {
        var userAgent = this.realUA;

        if (changeDevice == true) {
            var versionData = this.randomItem(this.data.deviceList.version);
            if (versionData && versionData.os && versionData.build) {
                userAgent = userAgent.replace(this.aVerRegExp, versionData.os);
                userAgent = userAgent.replace(this.bVerRegExp, versionData.build);
            }
            var modelData = this.randomItem(this.data.deviceList.model);
            if (modelData)
                userAgent = userAgent.replace(this.modelRegExp, modelData);

            var chrVer = this.randomItem(this.chrVer);
            if (chrVer)
                userAgent = userAgent.replace(/80\.0\.3987\.132/g, chrVer);
        }

        if (isBrowser == true) {
            userAgent = userAgent.replace(/\ Version\/4\.0/g, '');
            userAgent = userAgent.replace(/\; wv/g, '');
        }

        return Promise.resolve(userAgent);
```

Figure 7. Screenshot of the source code from the malware payload responsible for launching the webview with a specific configuration. Note the logic in this snippet that modifies the user agent string.
Source: White Ops Satori Threat Intelligence and Research

The randomization of the Chrome version supplied in the user agent is likely done with the intention of *device magnification*, that is, an attempt to make a single infected device look like many different devices. However, in aggregate this randomization becomes easily identifiable and can be used as an identifying characteristic (as shown in the next section).

Global impact: 2 billion fraudulent requests in a week

The scale and global impact of TERRACOTTA was impressive. In a single week in June 2020, the malware's ad fraud operation was responsible for more than 2 billion fraudulent bid requests, had upwards of 65,000 unwitting participating devices and spoofed more than 5,000 apps**.**

Faking Chrome versions with outdated values

The original lead that led the Satori team to the identification of the TERRACOTTA campaign was a highly uniform browser distribution and the presence of outdated Chrome mobile sessions.

White Ops identified rotation of different versions of Chrome for Android across several older versions of Chrome, and while older versions of Chrome for Android certainly remain in the ecosystem, it was evident that TERRACOTTA was lying about its browser engine. The figure below shows an example of TERRACOTTA adapting its user agent spoofing strategy for a period in February 2020 in an effort to evade detection.

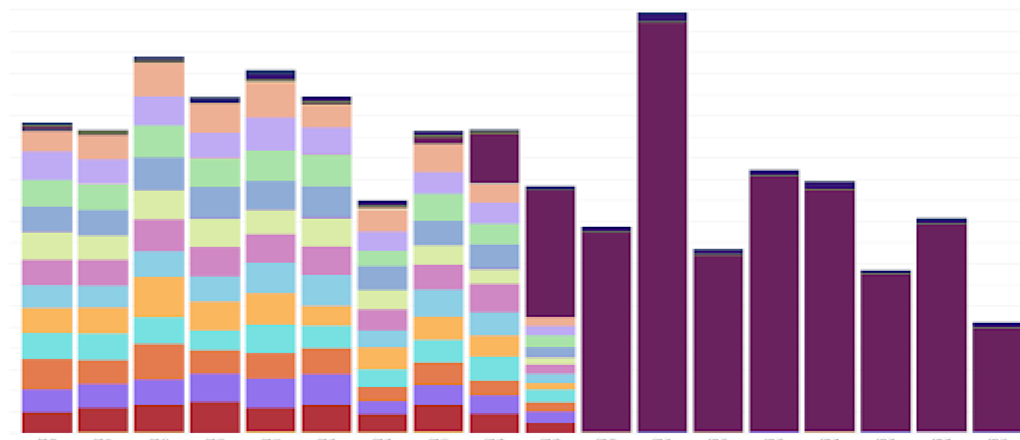

Figure 8: A time series chart showing Terracotta faking 10 different browser version, before updating to report only one more recent version.
Source: White Ops Satori Threat Intelligence and Research

Spoofed app and referrer values

One of the initial challenges in isolating TERRACOTTA traffic was that White Ops assessed early on that many of the values in the suspected TERRACOTTA traffic were "spoofed," meaning threat actors supplied false values for several fields. Although the scale and sophistication were considerable, the spoofed values appeared intentionally placed. Accepting this, White Ops surmised that somewhere in the threat actor's technical stack (malware binary or in the configuration of the malware), could be a presence of hardcoded values required for spoofing the fields.

A big portion of the TERRACOTTA traffic possessed a single value in the “Referrer” field of the HTTP GET headers to attribute the traffic to loopme[.]com. (We do not believe that LoopMe is involved in this scheme. However the threat actors' intentions behind using the loopme referrer domain are not clear at this time.) This spoofed referrer was not consistent with the app ID values provided in the session, even though the app ID values were also being spoofed. Armed with this knowledge, the Satori team identified both the traffic and malware binaries for the TERRACOTTA attack.

Use of residential and cellular IP space in the United States

The traffic White Ops Satori identified as TERRACOTTA originated almost exclusively from residential and cellular IP addresses. This observation supported the White Ops theory that mobile phone users were the victims of the botnet, contributing to the attack likely without their knowledge. While we observed TERRACOTTA traffic originating from several countries, the majority of traffic originated from IP addresses in the United States (US).

Indicators of Compromise

All of the TERRACOTTA apps, which have been removed from the Google Play Store, are based on React Native (RN). Indicators can be identified to discriminate RN apps using Firebase, adding more specific indicators to this threat in particular:

1. A registered service with the following identifier: *io.invertase.firebase.messaging.RNFirebaseBackgroundMessagingService*
2. The following permissions specified in the manifest:
    1. android.permission.FOREGROUND_SERVICE
    2. android.permission.WAKE_LOCK
3. The presence of the following file: *assets/index.android.bundle*
4. Multiple occurrences of the string “eval(c(“ within *assets/index.android.bundle*

Appendix A: All Identified Terracotta Apps

All of the following TERRACOTTA apps have been removed from the Google Play Store.
The list of all identified TERRACOTTA apps is available as a PDF here.

Previous Post
Next Post