


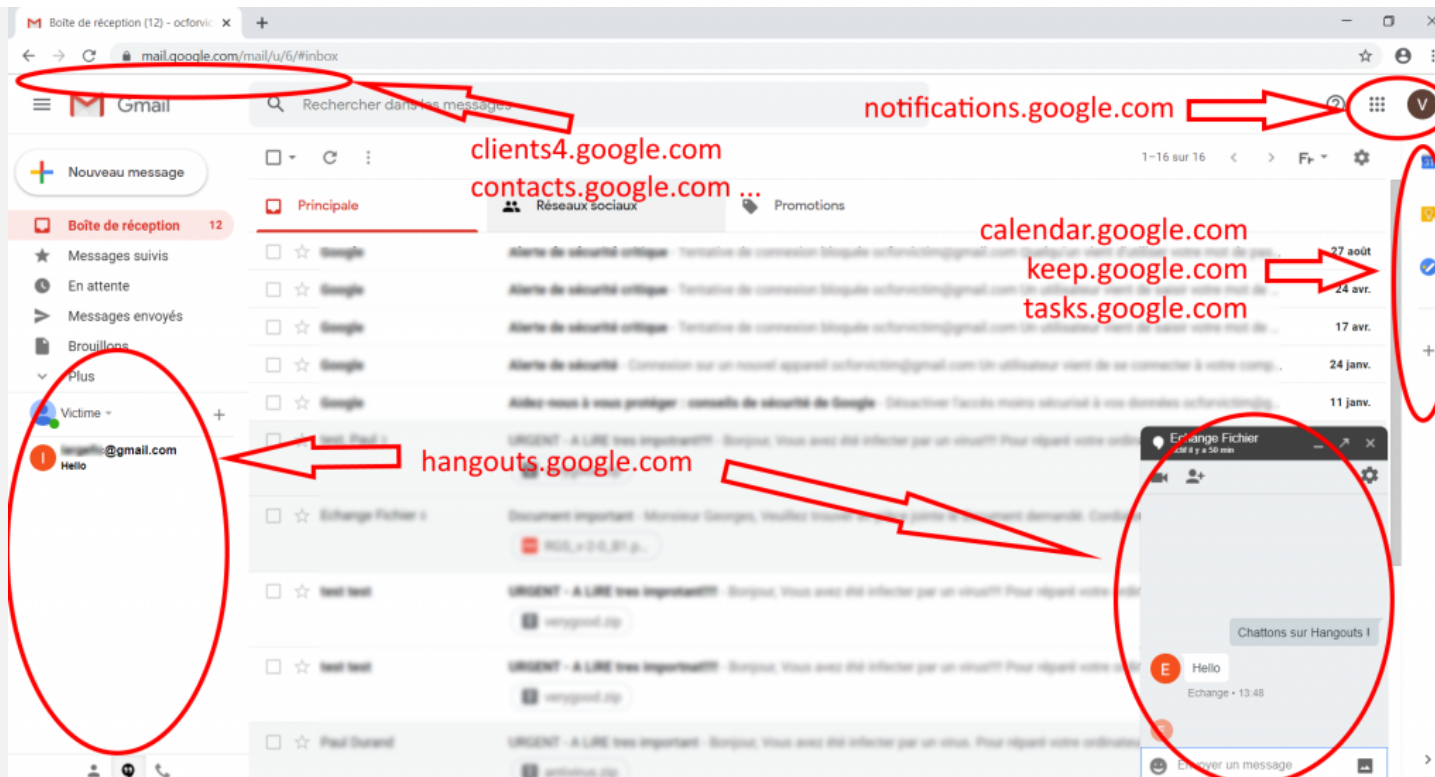
DOM XSS in Gmail with a little help from Chrome

 May 3, 2020 |  [8 Comments](#)

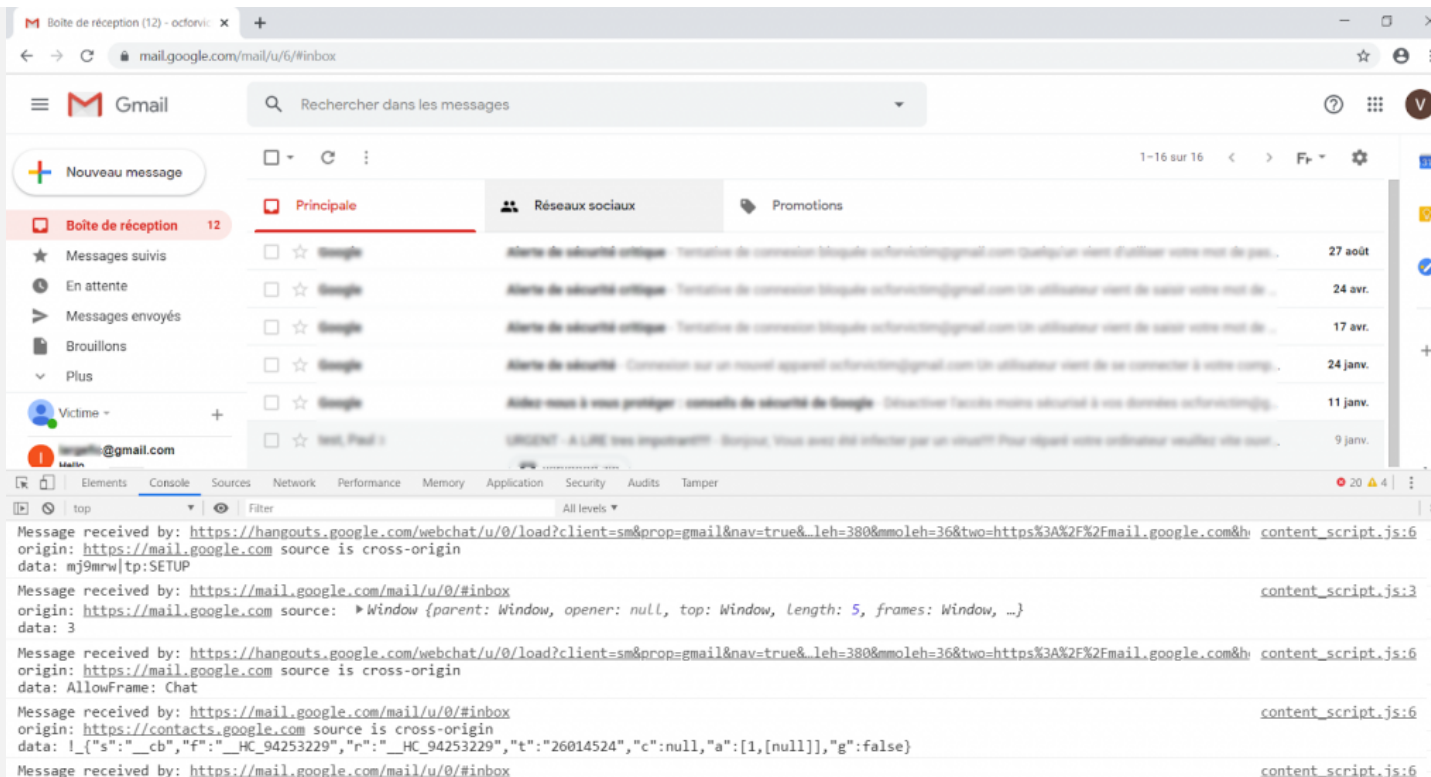
How to use browser features to help find DOM XSS.

The invisible Messages of Gmail

Last year, I looked for DOM XSS in Gmail website. Instead of using url params or the emails themselves as the source of the attack, I decided to use the much more discreet yet ubiquitous [postMessage](#) api. At first glance, the Gmail inbox seems a simple webpage, but if you go through the looking glass, it's actually a dozen of different webpages (or iframes) communicating between each others.



My first task was to make the cross-frames messages visible. This is not a native feature in DevTools yet. Instead, you can use this simple [postMessage logger extension](#). After reloading, the console is now overwhelmed with frame to frame messages going back and forth.



Each message has a **target** (the frame which receives the message), a **source** (the frame which sent the message), an **origin** (the domain of the source) and the **data**, which can be a string or a JSON object (or any kind of object that **will be cloned** in the process). Messages can be sent by any frame to another, even from different domain and even from different tab if the source has a reference to the target, with `window.opener`, `window.open()` and `window.frames`.

The target receives the message using

```
addEventListener("message", function(message) { /* handle message here */ })
```

like in the extension.

If there are too many messages, you can customize the extension to filter messages by any of their property. I was looking for interesting messages and one message in particular contained an url in the data:

```
Message received by: https://hangouts.google.com/webchat/u/0/load?client=sm&prop=gmail&nav=tr content_script.js:6
origin: https://mail.google.com source is cross-origin
data: AllowFrame: Chat

Message received by: https://mail.google.com/mail/u/0/#inbox content_script.js:3
origin: https://mail.google.com source: ▶Window {parent: Window, opener: null, top: Window,
data: 3

Message received by: https://mail.google.com/mail/u/0/#inbox content_script.js:6
origin: https://hangouts.google.com source is cross-origin
data: mj9mrw|c:[["capi:fsc","https://hangouts.google.com/webchat/u/0/","1588101636","/_sc GNGyv13e4Fkjhwut0gEH-
5Y6fYllkcyqg",0,0,380,"",36,"https://mail.google.com", "AMP3uWafC7qxo5VgOD4gGE33-OiYIoF -iframe-id"]],3]

Message received by: https://mail.google.com/mail/u/0/#inbox content_script.js:6
origin: https://hangouts.google.com source is cross-origin
data: mj9mrw|cp:[["h_sru","https://hangouts.google.com/webchat/u/0/frame?v=1588101636&hl=f #e","https://hangouts.goog
le.com/webchat/u/0/frame?v=1588101636&hl=fr&pvt=AMP3uW...ATwHmCp535NZ_XvjDWgJQczHKU7PNNyiT6M0

Message received by: https://mail.google.com/mail/u/0/#inbox content_script.js:6
origin: https://hangouts.google.com source is cross-origin
data: mj9mrw|c:[["h_sru","https://hangouts.google.com/webchat/u/0/frame?v=1588101636&hl=fr e","https://hangouts.googl
e.com/webchat/u/0/frame?v=1588101636&hl=fr&pvt=AMP3uW...ATwHmCp535NZ_XvjDWgJQczHKU7PNNyiT6M0Q
```

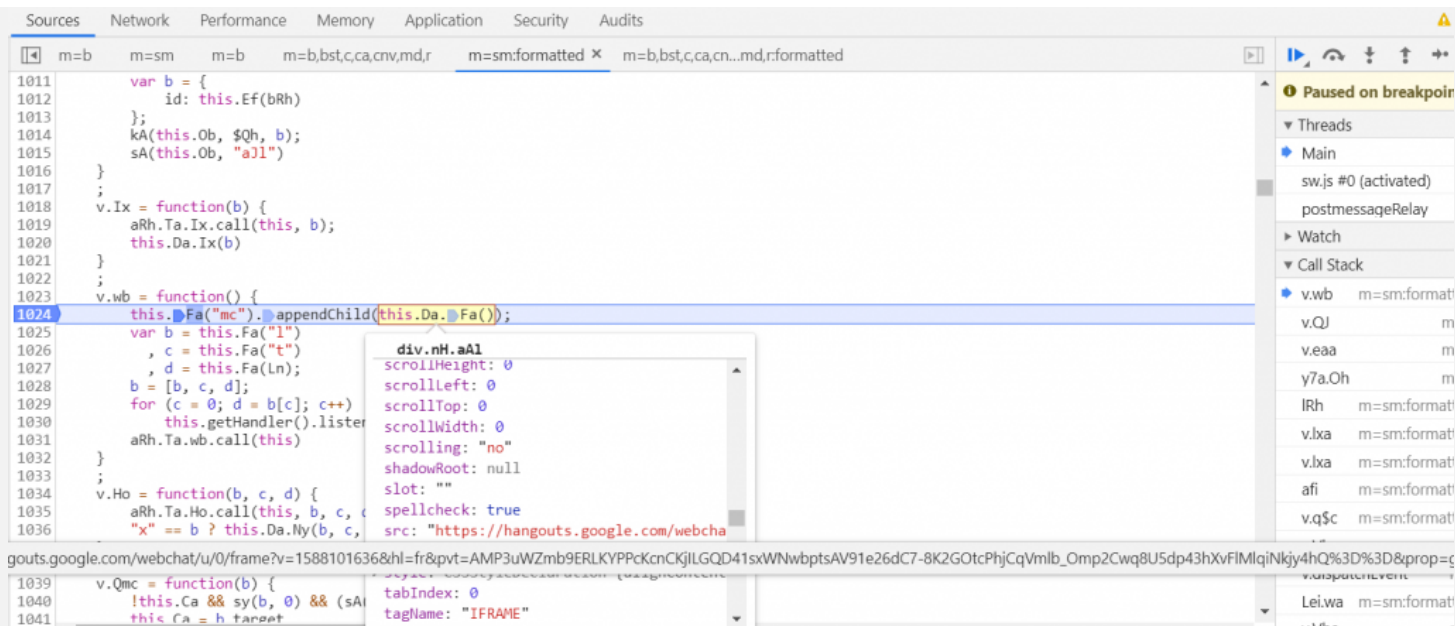
This message is sent by “hangouts.google.com” to “mail.google.com”. Not only is there a url in the message data, but the url contains the word “frame”. interesting...

The browser Toolbox

I went to the network tab of DevTools and filtered the requests by type “doc” which means “top window url and iframes src”. The same url was there:

Name	Method	Status	Protocol	Domain	Type	Initiator
0/	GET	200	http/1.1	mail.google.com	document	Other
data?sw=2&token=%5B%22cftp%22,%2279d...	GET	200	h2	mail.google.com	document	(index)
postmessageRelay?parent=https%3A%2F%2F...	GET	200	http/2+quic/46	accounts.google.com	document	cb=gapi.loaded_2
2?origin=https://mail.google.com&usegapi=...	GET	200	http/2+quic/46	contacts.google.com	document	contacts.google.c
load?client=sm&prop=gmail&nav=true&fid...	GET	200	http/2+quic/46	hangouts.google.com	document	m=sm:319
hscv?pvt=AMP3uWaFcGizC7qxro5VgOD4gGE...	GET	200	http/2+quic/46	hangouts.google.com	document	m=b:208
frame2?v=1588101636&pvt=AMP3uWaFcGiz...	GET	200	http/2+quic/46	hangouts.google.com	document	m=sm:428
frame2?v=1588101636&pvt=AMP3uWaFcGiz...	GET	200	http/2+quic/46	hangouts.google.com	document	m=sm:428
frame?v=1588101636&hl=fr&pvt=AMP3uWa...	GET	200	http/2+quic/46	hangouts.google.com	document	m=sm:73
client:clg=%7B%22	GET	200	http/2+quic/46	hangouts.google.com	document	m=b:208
client:xpc=%7B%22	GET	200	http/2+quic/46	hangouts.google.com	document	m=b:208
hscv?pvt=AMP3uWaFcGizC7qxro5VgOD4gGE...	GET	200	http/2+quic/46	hangouts.google.com	document	m=b:bst.c.ca.cnv.i
app?origin=https%3A%2F%2Fmail.google.co...	GET	200	http/2+quic/46	ogs.google.com	document	rs=AA2YrTv_Vh6v

I could also confirm that the request referrer was “mail.google.com” which was a good sign since it’s the domain that received the Message. The value in red circle is the initiator of the request, the JS code that loaded the iframe. You can click on it and it brings you directly to the corresponding code in the source tab. Unminify the code, set a breakpoint, reload and the breakpoint is hit:



And voila! The function that triggers the request is `appendChild()`. This is pretty much all we can understand from the code which is unreadable. But with the magic of the debugger we can confirm that the argument contains an `iframe` with the `src` set to the url. If you click on the functions in the Call Stack on the right, you can navigate through the “control flow” of the program and understand its logic.

For example, here is how the message is received by the frame:

```

else
    throw Error("z");
Tg++;
return c
}
, Zg = function() {
    var a = ah
    , b = Cg ? function(c) {
        return a.call(b.src, b.listener, c)
    }

```

```

ah      m=b:formatted:5114
Zg.b    m=b:formatted:5046
postMessage (async)
(anonymous)
    m=b,bst,c,ca,cn...ormatted:2918
c
    m=b,bst,c,ca,cn...ormatted:1474
setTimeout (async)

```

You can even see the code where the source of the message sent it:

```

.send = function(a, b) {
    var c = this.ea.ik;
    c && (this.send = function(d, e) {
        var f = this
        , h = this.ea.name;
        this.Rw = Wh(function() {
            f.Rw = 0;
            try {
                var k = c.postMessage ? c : c.document;
                k.postMessage && k.postMessage(h + "|" + d + ":" + e, f.Ia)
            } catch (n) {}
        }, 0)
    }, 0)
    , this.send(a, b))

```

```

ah      m=b:212
Zg.b    m=b:209
postMessage (async)
(anonymous)
    m=b,bst,c,ca,cn...ormatted:2918
c
    m=b,bst,c,ca,cn...ormatted:1474
setTimeout (async)
c.<computed>
    m=b,bst,c,ca,cn...ormatted:1477
Wh
    m=b,bst,c,ca,cn...ormatted:5871

```

There are dozens of functions between the listener and the loading of the url, across multiple files and thousands of lines of code. It is not uncommon that the browser freezes, breaks or skips breakpoint when you debug such heavy webpages, it's a matter of trial and error! 😊

I tried to send a message to the frame from the console with the same data but with “javascript:alert(1)” instead of the url. I didn’t got an alertbox, but a CSP error message.

```
✖ ▶ Refused to run the JavaScript URL because it violates the following Content Security Policy directive: "script-src 'report-sample' 'nonce-uajIdze2UP4f1G/EeYw3Ig' 'unsafe-inline' 'strict-dynamic' https: http: 'unsafe-eval'". Note that 'unsafe-inline' is ignored if either a hash or nonce value is present in the source list.
```

Thankfully, this CSP rule wasn’t enforced by IE11 and Edge (at the time) so I was able to trigger the alertbox on those browsers. There was no check on the origin of the message. A simple attack scenario is to start from the attacker webpage, open a new tab to Gmail and inject the payload in the Gmail tab using postMessage. The Gmail tab loads the javascript iframe and the attacker has arbitrary code execution on the victim’s Gmail page, which means it can read and send emails, change password of accounts registered to this email etc...

The random Channel Name

There was still one issue: with so much communication between so many frames, it is easy to get confused, so messages usually have a channel name. The name is a random 6 char generated by “mail.google.com” and transmitted in the first message to “hangouts.google.com”. In all following messages that it receives, “hangouts.google.com” checks if it contains the correct channel name, and if not it doesn’t process the message.

The attacker doesn’t know the channel name, and 6 alphanumeric is too much possibilities to try all.

The random generator is “Math.random()” which is not secure and [has been exploited](#) in the past by a Google engineer to find an XSS in Facebook! 😊 However the technique required the state of the random generator to be shared between cross-domain tabs which is not the case anymore.

The third solution is to load an iframe controlled by the attacker in the hierarchy of frames in

the Gmail tab. Because of the way cross-domain redirection of iframes works in the browser, the fact that Gmail X-Frame-Options header is "SAMEORIGIN" and that the messages were sent with the argument targetOrigin "*", it would then be possible to intercept the channel name and execute the XSS.

Conclusion

I couldn't find an easy way to load an iframe inside Gmail, but with all this I was confident enough to send a report to Google VRP and after a few days I received the "Nice Catch" answer and reward. Google fixed it by adding check on the origin of the message containing the url. The XSS doesn't work anymore, but the message is still sent if you want to check.

Browsers have all the cool features to navigate complex code, and for the features that are still missing, you can build you own extensions easily. With that, good hunting! 😊



8 comments on "DOM XSS in Gmail with a little help from Chrome"



Sergio

May 4, 2020 at 4:07 am

Really appreciated how clear you explained all the steps. Congrats!

Reply



Binit Ghimire

May 6, 2020 at 4:59 am

This is really an AWESOME research! Thank you for sharing, and congratulations!

Reply



JCVD

May 7, 2020 at 6:36 pm

Could it be possible to apply the same technique with the others messages if the page doesn't check the origin?

Reply



Enguerran Gillier

May 7, 2020 at 7:21 pm

Thx all! Yes I encourage you to check other messages on Gmail and on other websites. In this example the workflow is simple: the message contains a url that is loaded as iframe src, without url sanitation nor origin check.

In other cases the workflow might be way more complex: message encoding, partial sanitation, complex payload, etc...

For example, what if google.com trusts *.doubleclick.net? 😊

Reply



Nani

May 9, 2020 at 4:53 pm

could u do exploit on url only? or any other parameters?

Reply



Just a nOob

May 13, 2020 at 10:55 pm

Thanks for sharing! I really believe that the Dev tools its really a great tool to hack, we just need to know how to dominate this tool! Congrats for the achievement men! I already trying to read Google source code and I it was like WOW! The way how they randomly create the functions and vars and other code its mind blowing! So I really admire what u did! Keep Up!!

[Reply](#)



vahhitiu

May 14, 2020 at 9:03 am

Great write up. But I don't really understand how you get this random prefix. I just saw that you introduced two ways to get this random channel code, one is to crack `math.random()`, and you said that this method is not suitable for this scenario, and the other is to load a self-controlled iframe in gmail Tab, and in the next paragraph you said you didn't find an easy way to load it. Could you tell me more clearly how to get this random prefix?

[Reply](#)



Enguerran Gillier

May 18, 2020 at 7:05 pm

Good question, I didn't find a straight way to predict this random prefix! I found ways to load an arbitrary iframe in Gmail but it requires user interaction so it's not practical. I reported the vulnerability to Google with a PoC where the Google engineer had to manually copy and paste the random prefix in an input to trigger the XSS. Google VRP

validated the bug, maybe because a non secure random generator is not a proper security defense mechanism, so you have to suppose an attacker can predict the value.

[Reply](#)

Leave a Reply

Your email address will not be published. Required fields are marked *

Name *

Email *

Website

Recent Posts

[DOM XSS in Gmail with a little help from Chrome](#)

[Into the Borg – SSRF inside Google production network](#)

[Stored XSS on Facebook](#)

[FlashME! – WordPress vulnerability disclosure \[CVE-2016-9263\]](#)

[\[CVE-2016-9263\] XSF vulnerability in WordPress \[UPDATED\]](#)

Recent Comments

[Enguerran Gillier](#) on [DOM XSS in Gmail with a little help from Chrome](#)

[vahhitu](#) on [DOM XSS in Gmail with a little help from Chrome](#)

[Just a nOob](#) on [DOM XSS in Gmail with a little help from Chrome](#)

[Nani](#) on [DOM XSS in Gmail with a little help from Chrome](#)

[Enguerran Gillier](#) on [DOM XSS in Gmail with a little help from Chrome](#)

[^ Top](#) | [Home](#)

© 2020 [OpnSec](#). Theme by [XtremelySocial](#).