BARO SECURITY / TECHNICAL EVIDENCE

Cross-User Basket Read in OWASP Juice Shop v20.1.1

Source-backed object authorization assessment with a loopback-only reproducible proof of concept.

PUBLIC LAB - NOT CLIENT WORK

TARGET	OWASP Juice Shop
VALIDATED RELEASE	v20.1.1
FINDING	Authenticated cross-user basket read (IDOR)
EXECUTION BOUNDARY	Local loopback instance only
VALIDATION DATE	2026-07-31 KST

Assessment context. This artifact documents an authorized assessment of an intentionally vulnerable training application. It is not customer work, a paid engagement, a zero-day disclosure, a CVE claim, or evidence that a public deployment was tested.

Validated result. A fresh attacker account used only its own bearer token to retrieve a populated basket belonging to a second fresh account. The server returned HTTP 200, the victim basket ID, and its product association.

Prepared by Baro Security | Public lab capability evidence

Executive Summary

OWASP Juice Shop v20.1.1 deliberately exposes an authenticated insecure direct object reference (IDOR) in `GET /rest/basket/:id`. An ordinary authenticated user can replace the basket identifier in the path with another user's basket identifier. The server verifies that the request crosses an authentication boundary, but it does not require the requested basket to belong to the authenticated principal before returning it.

I reviewed the v20.1.1 source snapshot at commit `f915bddd82790d0f3018902d36ae9b4241a5f51f`, inspected the purpose-built PoC and its captured successful output, and confirmed the source archive's SHA-256 as `4fea62157f767b362e50a765b1ec2a54c6f2324236a3ee257bb4de3e2ad73911`. The PoC was executed against a locally built instance at `http://127.0.0.1:31001` on 2026-07-31 KST. It used one fresh user's valid bearer token to retrieve a second fresh user's populated basket and received HTTP 200. I did not test a public or live target, did not test persistence across versions, and did not establish an affected version range beyond the directly validated v20.1.1 release.

The demonstrated impact is cross-account disclosure of basket data. In the validated run, the response identified the victim's basket and contained the victim's seeded product. The source shows that the handler serializes the basket object and its associated products; this can expose basket ownership metadata, coupon state, product details, and join-record data such as quantity. This report does not claim access to passwords, bearer tokens, payment data, orders, write operations, or administrative privileges.

This behavior is part of the project's explicit “View another user's shopping basket” training challenge. The OWASP Juice Shop project describes the application as deliberately insecure, and the v20.1.1 release is the sole release assessed here. The remediation discussion therefore states the invariant a production application should enforce and gives an illustrative patch; it is not a request to remove the intentional lesson from Juice Shop.

Background

Actor and object model

The relevant actor is a normal user who can create an account and authenticate. No elevated role is required. On login, Juice Shop finds or creates the user's basket, places both the user object and the basket identifier (`bid`) in the signed authentication object, retains that object in its authenticated-user map, and returns the token and `bid` to the client:

TYPESCRIPT SNIPPET

```
// routes/login.ts, afterLogin()
BasketModel.findOrCreate({ where: { UserId: user.id } })
  .then(([basket]: [BasketModel, boolean]) => {
    const authenticatedUser = { data: user, bid: basket.id }
    const token = security.authorize(authenticatedUser)
    security.authenticatedUsers.put(token, authenticatedUser)
    res.json({ authentication: { token, bid: basket.id, umail: user.email } })
  })
```

This gives the server all the state needed to enforce ownership. For any authenticated request, `authenticatedUsers.from(req)` extracts the bearer token and resolves the corresponding authenticated object, including the original `bid`:

TYPESCRIPT SNIPPET

```
// lib/insecurity.ts, authenticatedUsers
from: function (req: Request) {
  const token = utils.jwtFrom(req)
  return token ? this.get(token) : undefined
}
```

The basket model also records the user relationship directly through `Basket.UserId`, while the basket-to-product relation uses `BasketItem` as its join model. A normal production invariant is consequently straightforward: the authenticated user may read only the basket whose `id` and `UserId` correspond to that principal.

Reachable HTTP surface

The server installs authorization middleware over `/rest/basket` and then wires the parameterized GET route to `retrieveBasket()`:

TYPESCRIPT SNIPPET

```
// server.ts, authorization and custom REST route registration
app.use('/rest/basket', security.isAuthorized(), security.appendUserId())
app.use('/rest/basket/:id', security.isAuthorized())

// ...
app.get('/rest/basket/:id', utils.asyncHandler(retrieveBasket()))
```

The middleware establishes an authentication boundary, not an object-level authorization boundary. This distinction is visible in the upstream API tests: an unauthenticated request for basket `1` is expected to receive 401, whereas an authenticated user requesting another user's basket `2` is expected to receive 200 and the requested object:

TYPESCRIPT SNIPPET

```
// test/api/basket.test.ts
const res = await request(app).get('/rest/basket/1')
assert.equal(res.status, 401)

// ...
const res = await request(app)
  .get('/rest/basket/2')
  .set({ Authorization: 'Bearer ' + token })
assert.equal(res.status, 200)
assert.equal(res.body.data.id, 2)
```

The assessment PoC does not rely on token forgery or any other authentication weakness. We cross this boundary with a valid, server-issued token belonging to the attacker account, which isolates the object-authorization failure.

Vulnerability Details

The caller controls the lookup key

Once the request has passed middleware, `retrieveBasket()` copies the path parameter into `id` and performs a database lookup using only that value:

TYPESCRIPT SNIPPET

```
// routes/basket.ts, retrieveBasket()
const id = req.params.id
const basket = await BasketModel.findOne({
  where: { id },
  include: [{ model: ProductModel, paranoid: false, as: 'Products' }]
})
```

There is no `UserId` predicate in this query, and the ID is not replaced by the authenticated user's `bid`. If we send `GET /rest/basket/6` with the token for a user whose own `bid` is `7`, the database lookup still asks for basket `6`.

The ownership comparison scores the challenge but does not authorize

The handler next recovers the authenticated user and explicitly compares the user's `bid` with the requested identifier. That looks like the natural point for an ownership check, but the result is passed only to `solveIf()`:

TYPESCRIPT SNIPPET

```
// routes/basket.ts, retrieveBasket()
challengeUtils.solveIf(challenges.basketAccessChallenge, () => {
  const user = security.authenticatedUsers.from(req)
  return user && id && id !== 'undefined' && id !== 'null' &&
    id !== 'NaN' && user.bid &&
    user?.bid != parseInt(id, 10)
})
```

In other words, a mismatch is treated as proof that the learner solved the intentional challenge. The comparison does not return 403 or 404, does not change the query, and does not prevent response serialization. We therefore carry the cross-user `basket` object unchanged into the response:

TYPESCRIPT SNIPPET

```
// routes/basket.ts, retrieveBasket()
if (((basket?.Products) != null) && basket.Products.length > 0) {
  for (let i = 0; i < basket.Products.length; i++) {
    basket.Products[i].name = req.__(basket.Products[i].name)
  }
}

res.json(utils.queryResultToJson(basket))
```

`queryResultToJson()` simply wraps the object as `{ status, data }`; it does not filter fields or re-check ownership. Because the original query eagerly includes the `Products` association, the response contains more than an existence bit. The returned object is structured basket data with related product and basket-item state.

The full security transition is:

TEXT SNIPPET

```
valid attacker bearer token
        |
        v
authentication middleware accepts the request
        |
        v
attacker-controlled :id is used as the sole basket predicate
        |
        v
user.bid != requested id records challenge success only
        |
        v
the other user's basket and Products association are serialized
```

The missing invariant is precise: successful authentication proves who the caller is, but the lookup never proves that the requested object belongs to that caller.

Intentional lab design

The source itself identifies this path as a broken-access-control exercise. `data/static/challenges.yml` describes the challenge as “View another user's shopping basket,” and the hacking instructor tells the learner to remain logged in while changing the browser's `bid` value. This is useful corroboration that the observed behavior is intentional training functionality, not an uncoordinated vulnerability claim against the project.

Relevant upstream source is available in the official repository:

- server.ts
- routes/basket.ts
- routes/login.ts
- lib/insecurity.ts
- test/api/basket.test.ts
- data/static/challenges.yml
- frontend/src/hacking-instructor/challenges/viewBasket.ts

Exploitability Analysis

Strongest demonstrated route

The strongest clean proof uses two fresh accounts so that object ownership is unambiguous. We log in as each user, record their different server-issued basket IDs, place one product in the victim's own basket, and confirm the victim can read it. We then carry only the attacker's token into `GET /rest/basket/<victim_bid>`. A 200 response whose `data.id` equals the victim's ID and whose `Products` array is non-empty proves all three relevant properties at once:

1. the principal is the attacker;
2. the selected object is the victim's basket; and
3. the response discloses the victim's associated product data.

This route is reliable when the target basket ID is known because there is no race, cache dependency, or multi-step state corruption. The request path is a direct primary-key lookup, and the ownership mismatch does not influence control flow after challenge scoring.

Identifier discovery and bounded enumeration

Basket identifiers are integer, auto-incrementing primary keys. The client also stores its own `bid` in session storage, and the official instructor suggests changing that value. These facts make nearby identifiers a plausible discovery route in a lab.

The response behavior provides a coarse object-existence oracle: the upstream tests expect an authenticated request for a nonexistent basket to return HTTP 200 with null or an empty object, while an existing basket returns a populated object. An authenticated learner could therefore try candidate IDs and distinguish a populated basket from a missing one. I did not perform a range scan or measure rate limiting, identifier density, or production-scale enumeration. The PoC instead uses basket IDs issued to two accounts that it created itself, which proves the authorization defect without broad probing.

Constraints and non-claims

This primitive has meaningful boundaries:

- It requires a bearer token accepted by the application. The demonstrated path is not anonymous.
- The reader must supply or discover an existing basket ID. An invalid ID returns no basket data.
- The assessed endpoint is a read path. This PoC does not use checkout, coupon, basket-item modification, or any other write operation.
- The source path returns a basket and its product association; it does not by itself establish access to account credentials, tokens, payment instruments, or historical orders.
- The demonstration used two lab-created accounts and one seeded product. It did not access pre-existing user data.

A browser-only route exists because the frontend builds the request from the session-storage `bid`, but it is less controlled as evidence: UI state, navigation, and pre-existing data can obscure which principal and object were used. The direct HTTP PoC is preferable because it pins the two identities and asserts the returned object. Token forgery would also add no value to this finding; a valid low-privilege token is sufficient, so introducing a separate authentication weakness would make the causal proof weaker.

Proof of Concept

The accompanying `poc/view-basket-idor.mjs` is a dependency-free Node.js script. It refuses non-loopback targets, creates random credentials for two fresh users, never prints credentials or bearer tokens, and exits successfully only when the attacker-authenticated response contains the victim's basket ID and at least one product.

Requirements and safe run

- OWASP Juice Shop v20.1.1 running on a loopback address
- Node.js 22 or later
- A disposable lab database, because the PoC creates two users and one basket item

From the report directory:

SH SNIPPET

```
cd poc
node view-basket-idor.mjs --target http://127.0.0.1:3000
```

For the validated instance, the same relative command used port `31001`:

SH SNIPPET

```
cd poc
node view-basket-idor.mjs --target http://127.0.0.1:31001
```

The script's loopback check is a deliberate safety boundary; it rejects non-loopback hosts and HTTPS URLs. Run it only against a locally controlled Juice Shop lab. It does not delete data, but it does leave the two test users and seeded basket item in the lab database. Stop and restart a default Juice Shop instance to reset its ephemeral lab state, or reset the disposable database using the operator's normal procedure.

Representative successful output

The following is the captured output from the validated v20.1.1 run:

TEXT SNIPPET

```
[+] target=http://127.0.0.1:31001
[+] creating two fresh lab users
[+] victim_bid=6 attacker_bid=7
[+] victim own read returned 1 product(s)
[+] cross-user request status=200
[+] returned_basket_id=6
[+] returned_product_count=1
[VULNERABLE] attacker read the victim basket with the attacker's token
```

Here we can trace the decisive state directly: the bearer token belongs to basket `7`, the requested and returned basket is `6`, and the returned association contains the product seeded by the victim. A production-style fixed handler should instead return a denial or an indistinguishable not-found response; in that case the PoC prints `[NOT REPRODUCED]` and exits with status 1.

The release built successfully on macOS arm64 with Node.js v24.14.0. The first startup attempt encountered a macOS code-signing mismatch between the Node runtime and the native SQLite module. The local lab was started after ad-hoc-signing a disposable copy of the runtime and constraining the generated listener to `127.0.0.1`. Neither change altered the TypeScript authorization logic assessed here.

Remediation

Production invariant

In a production application with this route shape, a caller-supplied basket identifier must be treated only as an object selector, never as authorization. The server must bind the lookup to the authenticated principal. It can derive the basket ID entirely from authenticated state, or query on both the requested `id` and the authenticated user's `UserId`. A mismatch must stop execution before any basket or product data is serialized.

Because Juice Shop intentionally teaches this failure mode, the following is a lab-safe illustration of the production pattern rather than a proposed upstream change:

TYPESCRIPT SNIPPET

```
// Illustrative production handler shape
export function retrieveBasket () {
  return async (req: Request, res: Response, next: NextFunction) => {
    try {
      const principal = security.authenticatedUsers.from(req)
      const requestedId = Number(req.params.id)

      if (
        principal?.data?.id == null ||
        principal.bid == null ||
        !Number.isInteger(requestedId) ||
        requestedId !== principal.bid
      ) {
        return res.status(404).json(utils.queryResultToJson(null))
      }

      const basket = await BasketModel.findOne({
        where: {
          id: requestedId,
          UserId: principal.data.id
        },
        include: [{
          model: ProductModel,
          paranoid: false,
          as: 'Products'
        }]
      })

      return res.json(utils.queryResultToJson(basket))
    } catch (error) {
      next(error)
    }
  }
}
```

Checking both values is intentionally defense in depth: the `bid` comparison binds the route parameter to authenticated session state, while the `UserId` predicate binds the database result to the user. In a service that permits multiple baskets per user, the `bid` equality can be omitted, but the principal-derived ownership predicate must remain.

Returning the same external shape for an unauthorized ID and a nonexistent ID also avoids turning the route into an ownership oracle. The application should log the distinction internally without exposing another user's existence to the caller.

Regression coverage for a production analogue should include:

1. an unauthenticated request is rejected;
2. user A can read user A's populated basket;
3. user B cannot read user A's basket by ID, and no user A fields appear;
4. malformed and nonexistent IDs reveal no object data;
5. unauthorized and nonexistent IDs have the chosen indistinguishable external response shape; and

6 both empty and populated cross-user baskets are denied before association serialization.

The critical regression is the two-principal case. A test that checks only authentication or only an owner's successful request cannot detect this class of horizontal authorization failure.

Summary

OWASP Juice Shop v20.1.1 intentionally demonstrates a compact horizontal authorization failure. The route authenticates the caller, accepts a caller-controlled basket ID, queries on that ID alone, and uses the one available ownership comparison only to mark the learning challenge as solved. It then returns the selected basket and associated product data.

In the authorized local lab, the executed PoC created two accounts, populated the victim's basket, and used only the attacker's valid bearer token to retrieve that basket with HTTP 200. The result proves cross-user read access without depending on token forgery, public-target testing, broad enumeration, or a write primitive.

For production systems, the durable lesson is to centralize object-level authorization at the data-access boundary: derive ownership from the authenticated principal, include it in the query, and deny before related data is materialized or serialized. Useful variant analysis would review sibling basket, order, coupon, and basket-item handlers for the same “authenticated but not owner-bound” query shape. Those paths were not dynamically tested in this assessment and are not findings in this report.