Application Security Health Check

Report for **Glasshouse, a sample company**

REFERENCE	VERSION	DATE	APPLICATION
0010526	1.0 (sample)	23 May 2026	Glasshouse Store (B2C e-commerce platform)

Executive Summary

This Health Check identified seven findings. Two need attention first: any logged-in customer can read other customers' orders, and the checkout accepts whatever price the browser sends. Both are easy to find and exploit, and would carry serious financial and regulatory consequences, so we recommend fixing them as soon as possible. The remaining five are lower priority and can be addressed in turn.

FINDINGS BY ESTIMATED COST

TOTAL $710k*

1 $250k **4** $150k **2** $120k **6** $90k **Other** $100k

MOST LIKELY FINDINGS

#	FINDING	HOW LIKELY	EST. COST	WHY
1	**Any customer can read another customer's orders**	Very likely	**$250k**	UK GDPR fine up to 4% of turnover, plus breach notification and remediation
2	**Checkout trusts the price the browser sends**	Likely	**$120k**	Estimated lost margin on orders placed at manipulated prices over a quarter
3	**No limit on password guessing, and no second factor**	Likely	**$60k**	Fraud refunds and support cost across compromised accounts
4-7	Other findings	Various	**$280k**	Lower priority; excluded here for brevity, full detail in the report.

* The total is an estimated worst-case figure: the sum of the per-finding estimates, not a prediction. How we arrive at these numbers, and how to have them adjusted, is in the appendix.

What we reviewed

A Health Check looks across the whole application for the weaknesses that most often lead to real loss, so this report is as much about the areas we cleared as the ones that need work. The table below shows what we assessed and where each area stands. Where the result reads "Reviewed, no material issues", we examined that area during this engagement and found nothing worth acting on. Because a Health Check is a time-boxed review rather than an exhaustive audit (see the appendix), that is a considered, evidenced result, not a guarantee that nothing could ever be found there.

Area we assessed	What this covers	Result
Access control and authorisation	Whether users can reach data or actions that are not theirs	Findings 1 and 4
Authentication and account protection	Login, password handling, brute-force protection, second factor	Finding 3
Checkout, pricing and payments	How prices, totals, discounts and payment confirmation are handled	Finding 2
Input handling and output encoding	Cross-site scripting, injection, and how user content is rendered	Finding 5
Secrets and configuration	How credentials and environment settings are stored and managed	Finding 6
Dependencies and supply chain	Third-party libraries and known-vulnerable versions	Finding 7
Error handling and information disclosure	What the application reveals when something goes wrong	Finding 7
Database query handling	Whether database queries are safely parameterised against injection	Reviewed, no material issues
Transport security	HTTPS / TLS configuration and secure cookie handling	Reviewed, no material issues
Server-side request and file handling	Unsafe outbound requests and unrestricted file operations	Reviewed, no material issues

Overall posture. Glasshouse Store is a well-built application with sound foundations: the database layer, transport security and file handling are all in good order, and we found no sign of systemic or architectural weakness that would call for a rebuild. The issues we did find cluster in two areas, access control and trusting data sent by the browser, and they are well understood, self-contained, and fixable with focused engineering rather than redesign. Findings 1 and 2 are the two that meaningfully move the figure on page 1; with those closed and the rest worked through in turn, the application would be in materially stronger shape. In short: a few important things to put right, none of them alarming, and a clear path to a clean bill of health.

For the delivery team

Below is every finding we identified, in priority order, so the finding numbers used throughout this report are easy to follow. The two tables below explain the **How likely** and **Est. cost** columns before you meet them in the register; both are conservative, worst-case judgements, not predictions.

How likely

Rating	What it means
Very likely	Within months, or already underway
Likely	More likely than not within a year
Possible	Within two to five years
Unlikely	Not within five years, but credible
Rare	An unusual chain of events; not foreseeable

Est. cost

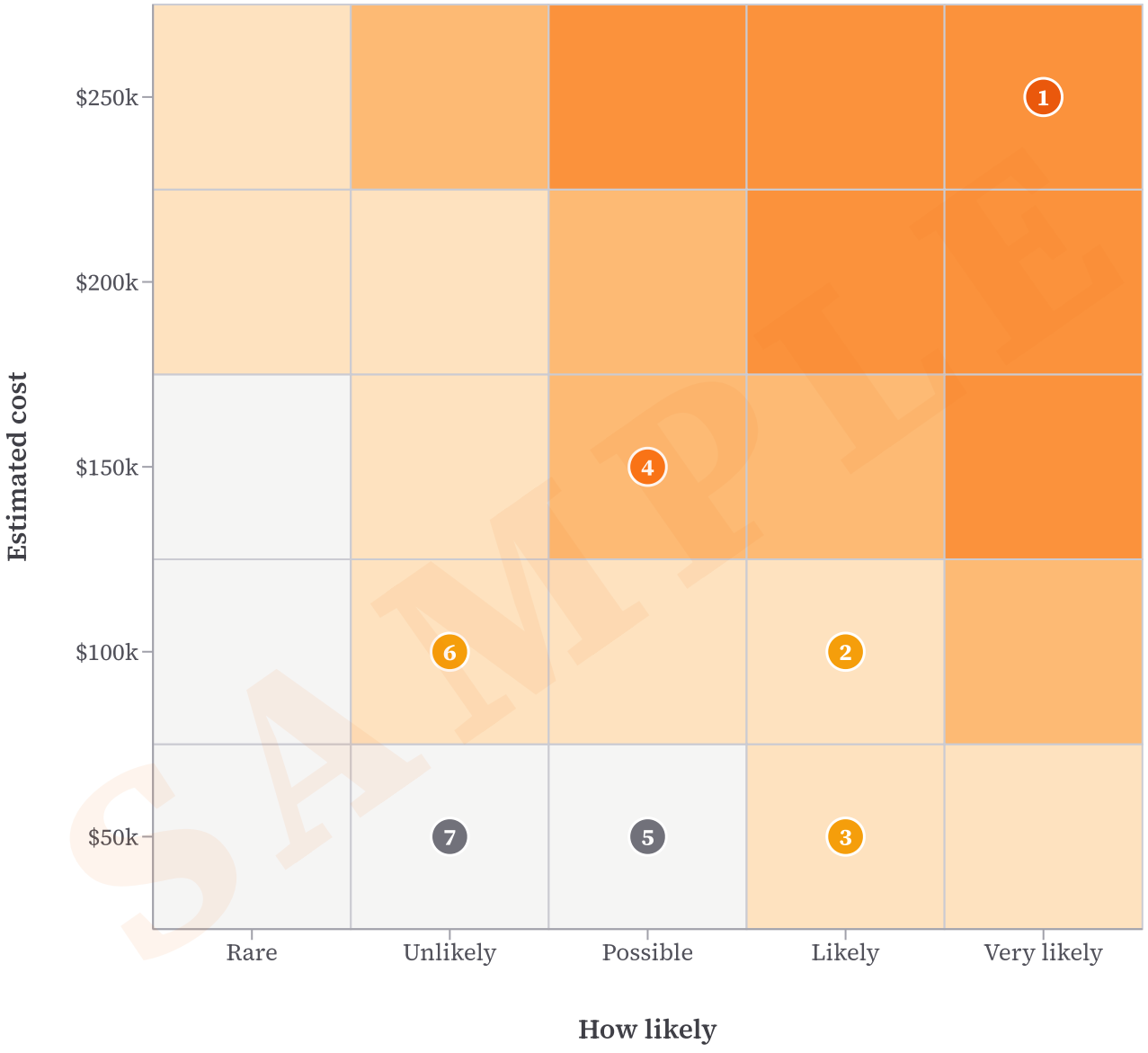
Type of loss	How we size it
Data and privacy breach	A share of turnover under UK GDPR / DPA 2018, kept below the statutory cap
Fraud and direct loss	The order volumes and margins the weakness exposes
Lost sales and reputation	A share of revenue over a short recovery period
Putting it right	Engineering time to fix and clean up

Where the company publishes figures we use them (Glasshouse's filed revenue is roughly £8m); otherwise we use sector averages. Each finding shows its own working, and we're glad to revise these with your internal numbers. We suggest starting with Findings 1 and 2, then working down the list.

#	Finding	How likely	Est. cost
1	Any customer can read another customer's orders	Very likely	$250k
2	Checkout trusts the price the browser sends	Likely	$120k
3	No limit on password guessing, and no second factor	Likely	$60k
4	Admin panel open to the whole internet	Possible	$150k
5	Customer reviews can run scripts in other shoppers' browsers	Possible	$30k
6	Live payment keys committed to the code history	Unlikely	$90k
7	Out-of-date dependencies and verbose error pages	Unlikely	$10k

How the findings compare

Each finding is plotted by how likely it is to be found and used (left to right) and what it would cost if it were (bottom to top), so the ones nearest the top right are the ones to deal with first. The shaded background is a simple guide: the warmer the cell, the sooner the finding warrants attention.



Findings in detail

Finding 1: Any customer can read another customer's orders

How likely. This is very easy to do. Changing a single number in the address bar returns another customer's order, and because the numbers run in sequence, a curious user or a simple script could step through every order within days. It needs no special skills or tools.

Impact if it happens. A full customer-data breach: names, delivery addresses and order history for every customer who has ever bought. It is a reportable personal data breach, requiring mandatory customer notifications, with the reputational damage that follows.

What this could cost. Glasshouse's filed accounts show revenue of roughly £8m. Under the UK GDPR and the Data Protection Act 2018, the statutory maximum for the most serious breaches is £17.5m or 4% of annual worldwide turnover, whichever is higher, so for a firm this size the formal ceiling is the £17.5m figure. In practice, though, fines are proportionate and a business of this scale is very unlikely to face anything close to that: the 4% measure works out at around £320k here, and the ICO weighs how many people were affected and whether you had taken reasonable care. We have estimated a conservative £250k, well below even the 4% figure, because fixing this and being able to show due diligence would materially reduce both the chance and the size of any penalty. On top of any fine, expect breach-notification, customer-support and clean-up costs.

What we found. The order endpoint returns an order based only on the ID in the URL; it never checks the order belongs to the person asking. Location: `app/api/orders/[id]/route.ts:18` (OWASP A01, CWE-639).

Technical rating. CVSS v4.0 6.9 (Medium), `CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:N/VA:N`: network-reachable with low complexity, needing only a logged-in customer account, for a high confidentiality impact (every customer's order data). EPSS: indicative only, this is a bespoke flaw with no published CVE.

Confirmed. Reproduced in testing: changing the ID in the URL returned another customer's order.

How to fix it. Confirm the order's owner matches the logged-in user before returning anything, and allow staff roles explicitly. Apply the same rule to every "fetch by ID" endpoint. As a second layer, swap sequential IDs for non-guessable ones.

Finding 2: Checkout trusts the price the browser sends

How likely. This is easy for anyone willing to tamper with a request. We changed the order total to GBP 0.01 and it was accepted for payment at that amount. Once the technique is known it can be repeated at scale, so it would likely be exploited within days of anyone attempting it.

Impact if it happens. Direct financial loss: goods bought for a fraction of their price and discounts set to any value. At scale this is a repeatable loss that may go unnoticed until the figures are reconciled.

What this could cost. Glasshouse trades on roughly £8m of revenue, so the loss here is the margin given away on manipulated orders plus any goods shipped for a fraction of their cost. We have taken a conservative £120k, roughly a few weeks of margin leakage before it would surface in reconciliation; the longer it remains open, the higher it climbs. Calculating prices on the server would largely eliminate it.

What we found. The server uses the prices and total the browser sends instead of working them out itself; discount codes are applied in the browser too. Location: `app/api/checkout/route.ts:41` (OWASP A04, CWE-602).

Technical rating. CVSS v4.0 7.1 (High), `CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:H/VA:N`: a logged-in user tampers with the request for a high integrity impact (prices and totals accepted as sent). EPSS: indicative only (no published CVE).

Confirmed. Reproduced in testing: we set the total to GBP 0.01 and the order was accepted for payment at that amount.

How to fix it. Look up each product's price from your own database at checkout, calculate the total on the server, validate discount codes on the server, and confirm the final amount with the payment provider before fulfilling.

Finding 3: No limit on password guessing, and no second factor

How likely. This is automated and almost certainly happening already. Attackers replay username and password pairs leaked from other breaches; with no limits in place they can attempt millions, and because people reuse passwords, a steady proportion will succeed.

Impact if it happens. Customer accounts taken over: order history, saved addresses, and any stored payment details exposed, plus fraudulent orders placed in the customer's name.

What this could cost. We have taken a conservative £60k: the fraud refunds, chargebacks and support time across a realistic number of taken-over accounts for a customer base this size, not a worst-case mass compromise. Rate limiting and an optional second factor would reduce both the number of accounts affected and the cost of each.

What we found. Login and password reset accept unlimited attempts with no rate limiting, lockout or delay, and there is no second-factor option. Location: `app/api/auth/[ … ]/route.ts:27` (OWASP A07, CWE-307).

Technical rating. CVSS v4.0 8.2 (High), `CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:L/VA:N`: no privileges or interaction needed, and credential stuffing against an unprotected login leads to account takeover (high confidentiality impact). EPSS: indicative only (no single CVE), though credential-stuffing campaigns are common in the wild.

Confirmed. By code review.

How to fix it. Add rate limiting and progressive delays per account and per IP, lock or challenge after repeated failures, and offer a second factor (require it for staff). Make password reset respond the same way whether or not the email exists.

Finding 4: Admin panel open to the whole internet

How likely. Public admin panels are typically found and targeted within hours of going live. This one is protected by a single password, so one phished or reused credential would be enough.

Impact if it happens. Full control of the business: access to every customer's data, refunds issued to an attacker's own cards, and prices changed without restriction.

What this could cost. Admin access controls the entire business, so this combines two costs: the refunds an attacker could push through, and a personal-data breach on the same UK GDPR / Data Protection Act 2018 basis as Finding 1 (up to roughly £320k at this turnover). We have taken a conservative £150k, reflecting that an attacker would more likely be caught after limited refund fraud than reach the full breach figure. A second factor for admins and restricting the panel to known networks removes most of it.

What we found. The dashboard at `/admin` is reachable from any browser with no second factor and no IP restriction. Location: `app/admin/layout.tsx:12`, `middleware.ts:20` (OWASP A01/A07, CWE-284).

Technical rating. CVSS v4.0 8.7 (High), `CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:L`: an internet-facing admin panel behind a single password gives high confidentiality and integrity impact (all data, refunds, prices). EPSS: indicative only (no published CVE).

Exploit path.

1. Find the public admin login at `/admin` (`app/admin/layout.tsx:12`).
2. Sign in with a single phished or reused password; the gate has no second factor (`middleware.ts:20`).
3. Use the admin endpoints to issue refunds, change prices, or export every customer record.

Confirmed. By code review.

How to fix it. Require a second factor for every admin, restrict `/admin` to known IP addresses or a VPN, keep admin login separate from customer login, and record every admin action.

Finding 5: Customer reviews can run scripts in other shoppers' browsers

How likely. This takes moderate effort: an attacker submits a malicious review, which then runs in the browser of every shopper who views that product page. It requires some skill, but the reach and the potential payoff make it an attractive target.

Impact if it happens. Logged-in shoppers' sessions can be stolen or their pages altered, from a product page anyone can visit. Because the malicious code is stored, it runs for every visitor to that page, not only the attacker.

What this could cost. This is harder to price because the loss is largely reputational: hijacked shopper sessions, a period with the affected pages taken down, and the loss of goodwill. We have taken a conservative £30k, a short run of lost sales and clean-up for a store this size. Showing user content as plain text, with a Content-Security-Policy behind it, removes the problem.

What we found. Reviews are rendered as raw HTML with no sanitising, and there is no Content-Security-Policy as a backstop. Location: `components/ReviewList.tsx:44` (OWASP A03, CWE-79).

Technical rating. CVSS v4.0 5.9 (Medium), `CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:A/VC:L/VI:L/VA:N`: a stored script that runs in a victim shopper's browser (user interaction required), with limited impact on their session. EPSS: indicative only (no published CVE).

Exploit path.

1. Submit a product review containing a script; it is stored without sanitising.
2. Any shopper opens that product page and the script runs in their browser (`components/ReviewList.tsx:44`), with no CSP to stop it.
3. The script steals the shopper's logged-in session or alters what they see.

Confirmed. By code review.

How to fix it. Show user content as plain text, or sanitise it with a vetted library if you need formatting, and add a Content-Security-Policy as a second layer.

Finding 6: Live payment keys committed to the code history

How likely. This is lower, because it requires access to the repository. However, anyone who has ever cloned it, such as on an old laptop, by a former contractor, or in a careless fork, already holds the values.

Impact if it happens. A leaked payment key can move money and read transaction data; a leaked database connection string can expose the whole customer database.

What this could cost. It depends on who holds the repository and whether the keys are still live. We have taken a conservative £90k: fraudulent charges on a leaked payment key before it is noticed, plus the work to rotate every secret and clean the history. Rotating the secrets today and moving them into a manager caps the exposure quickly.

What we found. A live payment secret and the production database URL are committed at `.env.example:3–4` and remain in earlier commits. (OWASP A05, CWE-798)

Technical rating. CVSS v4.0 5.8 (Medium), `CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:N/VC:H/VI:H/VA:N`: high confidentiality and integrity impact if the secrets are used, but it requires access to the repository (an attack requirement), which holds the score down. EPSS: indicative only (no published CVE).

Exploit path.

1. Obtain the repository (a fork, an old clone, or a former contractor's laptop).
2. Read the live payment key at `.env.example:3` and charge or refund through the payment API.
3. Read the database URL at `.env.example:4`, connect directly to production, and read every customer record.

Confirmed. By code review.

How to fix it. Rotate every secret that has touched the repo today, remove them from history and force-push, move secrets into a manager, and add automated secret scanning to your pipeline.

Finding 7: Out-of-date dependencies and verbose error pages

How likely. This is opportunistic. Known-vulnerable libraries are well-documented targets, and detailed error pages reveal how the application is built.

Impact if it happens. On their own these are minor, but together they make every other finding here easier to exploit.

What this could cost. Low on its own, mostly the engineering time to patch the flagged packages and switch off detailed error pages, which we have put at a conservative £10k. Its real cost is making the other findings easier, so it is best cleared alongside them.

What we found. Several dependencies are behind on security updates, and unhandled errors in production return full stack traces with file paths and library versions. Location: `package.json:28`, `next.config.js:9` (OWASP A06/A05, CWE-1104/209).

Technical rating. CVSS v4.0 6.1 (Medium), `CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:L/VI:L/VA:L`: the flagged packages carry known CVEs reachable over the network. EPSS 0.55: taken from those CVEs, indicating a meaningful real-world exploitation likelihood (this is the one finding here with a genuine EPSS).

Confirmed. By code review.

How to fix it. Patch the flagged packages, turn on automated dependency alerts, and show users a plain error while logging the detail server-side only.

Recommended next steps

The findings are already in priority order. We suggest working through them in three waves; the timing is yours, but this is the sequence that removes the most exposure for the least effort first.

1. **First: Findings 1 and 2.** These two carry the largest combined exposure and are the easiest to exploit. Confirm the owner of every record before returning it (Finding 1) and calculate every price and total on the server (Finding 2). Between them they remove most of the figure on page 1.
2. **Next: Findings 3 and 4.** Add rate limiting and a second factor to login, and restrict the admin panel to a second factor and known networks. These close the two routes to account and business takeover.
3. **Then: Findings 5, 6 and 7.** Show user content as plain text behind a Content-Security-Policy, rotate the committed secrets and move them into a manager, and patch the flagged dependencies while switching off verbose error pages.

Each finding's own **How to fix it** note has the specifics. None of this requires a redesign; these are focused changes to existing code and configuration.

Threat model

Alongside this report we provide a threat model for Glasshouse Store as a `.justappsec` file. It is a living inventory of the threats we identified, each with a status, the area it affects, and the path an attacker would take. The tables below are a snapshot taken when this report was issued; the open items correspond to the findings above, and we have kept the issues already resolved during the engagement for the record.

The file is yours to keep. Open it at https://justappsec.com/threat-model to see the threats and their attack paths laid out visually, update each one's status as you address it, and add your own as the application changes. It is a plain text file that lives happily in version control next to your code, and any AI coding assistant can read and update it directly, so the model stays current between Health Checks.

Open and in progress

Threat	Area	Status
Order endpoint returns any customer's order by ID	Authorisation	Open
Checkout trusts client-supplied prices and totals	Business Logic	Open
Admin panel reachable from the public internet	Authorisation	Open
Stored XSS in product reviews	Input Handling	Open
Outdated dependencies and verbose error pages	Dependencies	Open
No rate limiting or MFA on login and reset	Authentication	In progress
Live payment and database secrets in git	Configuration	In progress

Closed

Threat	Area	Status
Stripe webhook signature not verified	Business Logic	Closed (mitigated)
Session cookie missing Secure and HttpOnly flags	Session Management	Closed (mitigated)
Missing HSTS allows TLS downgrade on first visit	Configuration	Closed (mitigated)

Closed items are kept so the record stays complete.

Appendix: scope, approach and about this assessment

What we looked at. The Glasshouse Store codebase (`github.com/glasshouse/store`, `main`) and the staging environment at `https://staging.glasshouse.example`.

How we did it. AI-assisted tooling reads the full codebase for breadth, performs the first-pass analysis, and drafts the findings and this report. A senior application security engineer then reviews that work in full: confirming each issue, discarding false positives, weighing the real business impact and how easily each would be found, and revising the wording where needed. Nothing reaches you without that human review and sign-off. This is a focused, time-boxed Health Check, not a penetration test, a full line-by-line audit, or a certification.

How we judge likelihood and cost. Likelihood combines how easily a weakness can be found and exploited with how exposed it is, summarised as a single rating from Rare to Very likely; the timeframes behind those words are set out on the delivery-team page. The estimated cost is what it would cost you if the weakness were used, expressed in money, with the reasoning shown in each finding. The matrix plots the two together.

How we estimated the financial exposure. The figure beside each finding is our estimate of the worst-case cost to you if that weakness were exploited: the kind of loss it would cause (for example regulatory fines, fraud, lost sales, or the cost of putting it right) rather than a prediction that it will. We base each estimate on the nature of the finding and conservative assumptions, and we set out the reasoning in that finding's write-up. The headline total is simply the sum of those estimates.

These figures are a starting point for a conversation, not a verdict. If you have internal numbers, revenue at risk, contract values, real breach or downtime costs, that would sharpen them, tell us and we'll revise the estimates with you.

Who did it. This Health Check was carried out by Davy Rogers, Senior Application Security Engineer at JustAppSec, who holds the Certified Information Systems Security Professional (CISSP) and Offensive Security Certified Professional (OSCP) certifications. He reviewed the analysis, made the final judgement calls, approved every finding, and signed off this report, and is accountable for its conclusions.

Version history. Every issued version of this report, taken from our release log.

Version	Date	Notes
v1.0	23 May 2026	Initial Health Check
v1.1	23 May 2026	Minor wording corrections; findings unchanged

This report is confidential and intended for the named client. Questions: contact@justappsec.com