96 checks run against your site

14 vulnerabilities found across your site

4 we exploited

9 attacks our model invented for your site

7 high or urgent, the ones to fix first

This scan covers northwind-shopfront.example only. The hosts under your domain are covered by a paid full scan.

Security report for northwind-shopfront.example

RESULT **Fail**

We found 14 holes on your site. 7 are high or urgent. We exploited 4 of them.

Prepared for

The owner of **northwind-shopfront.example**, who proved control of it before we started.

The assessment

Run on **12 November 2026**. Scan **sample-0000-0000-0000-000000000000**.

This report

This report is **27** pages. Your dashboard carries every individual check behind it.

Contents

01 **Executive summary** 3
What we did on your site, the result, and where to start.

02 **What we checked** 4
Every area we tested and how each one came back.

03 **Vulnerabilities published this year** 9
Why the ones that apply to your site are the ones we test.

04 **Findings** 11
Each hole, what we proved, what it reaches and the change that closes it.

05 **Attacks our model invented** 19
The attacks written for your site, and why inventing them keeps you ahead.

06 **Fixing from your repository** 20
The changes we make in your code, and how they reach you.

07 **Invented attacks we keep and run again** 22
How the attacks we confirm on your site are run again first on your later scans.

08 **Method and terms** 24
How we work, and what potential, confirmed and exploited mean.

09 **Appendix** 25
This run's parameters and every kind of hole we look for.

How to use this report

If you own the site

The summary gives the result, how far an attacker gets and the findings to fix first. Hand the findings to whoever runs your site, or connect your repository and we write the changes your code can take as pull requests.

If you build or run it

Findings carries each hole with what we proved, what it reaches and the change that closes it. The fix list on your dashboard has the steps for your platform, and a connected repository receives each change as a pull request.

Once it is fixed

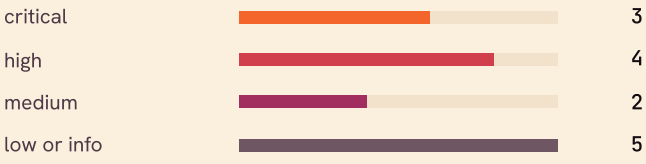
Your next scan proves each fix closed, and your dashboard records it. With scheduled scans on, FCD keeps attacking your site on your schedule and tells you what changed since the last one.

01 Summary

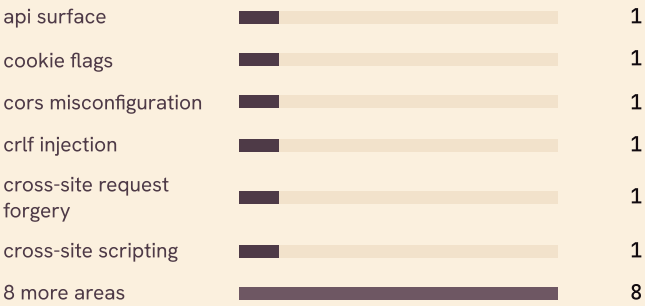
Executive summary

We attacked northwind-shopfront.example the way an attacker would on 12 November 2026: we ran 96 checks across 14 areas of your site, and our attacker model invented 9 attacks written for it, 7 of them run by us. A second model, our verifier, checks findings against the evidence before they are confirmed, and our own probes prove what can be exploited. This report is the record of that assessment and the plan for closing every hole in it.

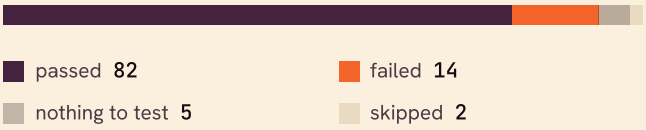
Findings by severity



Findings by area



Checks run and their results



How far an attacker gets

From this scan alone, attackers reached: recon, foothold, data and control. Your server itself stayed out of reach in this scan.

Fix these first

01 critical

Default credentials accepted at /admin/login

An account is still on its factory password. We recorded it as exploited, and it reaches control.

02 critical

Path traversal via ?file= reads arbitrary files

A page that serves files lets a visitor step out of the intended folder. We recorded it as exploited, and it reaches data.

03 high

Sensitive file exposed: /.env

Files that were meant to be private are readable straight from the web. We recorded it as exploited, and it reaches data.

02 What we checked

What we checked

96 checks across 14 categories.

Every area we tested on your site, whether it came back clean, and the counts behind the result. Each check names a risk we went looking for, so a check that passed is one we ran and found nothing. Two examples sit under each area; your dashboard carries every individual check.

96
checks run against your site

82
passed, meaning they ran and found nothing

14
failed

How this result is decided

1. A check passes when it ran and found nothing, and fails when it found a problem. Some failed checks name a smaller problem that we list under its area and do not raise as a finding.
2. An area passes when every check in it passed. One failed check fails the area.
3. The assessment fails when a confirmed critical or high finding stands against the site. Below that it is a pass with findings. With nothing found at all it is a pass.

Fail. This site fails. 6 confirmed findings sit at high or urgent, and one such finding is enough to fail a site.

✕ security headers

`7 run, 6 passed, 1 failed`

We read the headers your server sends and check for each of the five that tell a browser what to trust.

```
the response headers on the home page at https://northwind-shopfront.example/: failed (CSP, HSTS and Referrer-Policy absent)
the response headers on the home page at https://northwind-shopfront.example/?probe=1: passed
```

✕ sensitive file exposure

19 run, 18 passed, 1 failed

We ask for private files and folders that should never be public, and check that anything returned really is that file.

a request for a dotfile at the web root at https://northwind-shopfront.example/.env: failed (served with status 200)
a request for a dotfile at the web root at https://northwind-shopfront.example/?probe=1: passed

✕ cross-site scripting

10 run, 9 passed, 1 failed

We put a harmless marker in every parameter and form field we found, and watch for it landing in the page unescaped.

a marker in the search parameter at https://northwind-shopfront.example/search?q=: failed (returned unescaped in the page)
a marker in the search parameter at https://northwind-shopfront.example/?probe=1: passed

✕ sql injection

12 run, 11 passed, 1 failed

We put a single quote in every parameter and form field we found, and watch for a database error coming back.

a quote in the product id at https://northwind-shopfront.example/product?id=: failed (database error returned)
a quote in the product id at https://northwind-shopfront.example/?probe=1: passed

✕ cors misconfiguration

5 run, 4 passed, 1 failed

We send requests to your site and its APIs from an origin we do not own, and read whether the server allows them.

an Origin we do not own, offered to the API at https://northwind-shopfront.example/api/cart: failed (reflected with credentials allowed)
an Origin we do not own, offered to the API at https://northwind-shopfront.example/?probe=1: passed

— **open redirect**

`0 run, 0 passed, 0 failed, 2 with nothing to test`

We put an external address into the parameters that redirect, and see whether your server forwards to it.

```
the area's checks at https://northwind-shopfront.example/: nothing to test (no redirect parameter found)
the area's checks at https://northwind-shopfront.example/: nothing to test (no redirect parameter found)
```

✕ **path traversal**

`6 run, 5 passed, 1 failed`

We ask the file parameters for a path outside their folder and check whether a system file comes back.

```
a parent-directory path in the file parameter at https://northwind-shopfront.example/download?file=: failed (a file outside the folder was returned)
a parent-directory path in the file parameter at https://northwind-shopfront.example/?probe=1: passed
```

✕ **cookie flags**

`4 run, 3 passed, 1 failed`

We load your home page as an anonymous visitor and read the flags on every cookie it sets.

```
the attributes on each cookie set at https://northwind-shopfront.example/: failed (session cookie without Secure and SameSite)
the attributes on each cookie set at https://northwind-shopfront.example/?probe=1: passed
```

✕ **api surface**

`3 run, 2 passed, 1 failed`

We ask the API paths attackers sweep for, and note which ones answer with data.

```
the well-known API description paths at https://northwind-shopfront.example/openapi.json: failed (an API description answered)
the well-known API description paths at https://northwind-shopfront.example/?probe=1: passed
```

— graphql introspection

`0 run, 0 passed, 0 failed, 3 with nothing to test`

We send one introspection query to any GraphQL endpoint the scan found.

```
the area's checks at https://northwind-shopfront.example/: nothing to test (no GraphQL endpoint answered)
the area's checks at https://northwind-shopfront.example/: nothing to test (no GraphQL endpoint answered)
```

✕ crlf injection

`5 run, 4 passed, 1 failed`

We put an encoded line break in the redirect parameters and check whether it becomes a real response header.

```
an encoded line break in the redirect parameter at https://northwind-shopfront.example/login?next=: failed (a header we sent appeared in the response)
an encoded line break in the redirect parameter at https://northwind-shopfront.example/?probe=1: passed
```

✕ source map exposed

`2 run, 1 passed, 1 failed`

We ask for the .map file beside each script bundle and check whether it parses as a source map.

```
the source map beside each script at https://northwind-shopfront.example/js/app.js.map: failed (served with status 200)
the source map beside each script at https://northwind-shopfront.example/?probe=1: passed
```

✕ secret exposure

`3 run, 2 passed, 1 failed`

We run our key patterns over the HTML and the JavaScript your site serves to browsers.

```
key-shaped strings in the scripts the site serves at https://northwind-shopfront.example/js/app.js: failed (an AWS-style access key id pattern)
key-shaped strings in the scripts the site serves at https://northwind-shopfront.example/?probe=1: passed
```

✕ default credentials

`4 run, 3 passed, 1 failed`

We try a short list of factory passwords on any login the scan found, and confirm the session it hands back really opens a protected page.

```
a factory password on the admin login at https://northwind-shopfront.example/admin/login: failed (the login was accepted)
a factory password on the admin login at https://northwind-shopfront.example/?probe=1: passed
```

✕ cross-site request forgery

`3 run, 2 passed, 1 failed`

We read every POST form the crawl found and check it carries an anti-forgery token field.

```
the hidden fields on each POST form at https://northwind-shopfront.example/account/update: failed (no anti-forgery token field)
the hidden fields on each POST form at https://northwind-shopfront.example/?probe=1: passed
```

– TLS and certificates

`0 run, 0 passed, 0 failed, 2 skipped`

We open TLS connections to your server and read which protocol versions it accepts, what certificate it presents and which cipher it negotiates.

```
the area's checks at https://northwind-shopfront.example/: skipped (the certificate check timed out)
the area's checks at https://northwind-shopfront.example/: skipped (the certificate check timed out)
```

✕ novel frontier checks

`13 run, 12 passed, 1 failed (13 from the catalogue)`

Checks our attacker model invented for your site. We report how many ran and how many passed, and we keep the method to ourselves.

03 Published this year

Vulnerabilities published in 2026 and the ones that apply to your site

96,498

vulnerabilities published by the CVE Program in 2026, counted to 11 November 2026.

A published vulnerability names a product and the versions it affects, and it can only be used against a site that runs them. So we start where an attacker starts, by finding the software your site runs and the versions it gives away. We check those versions against the published records for WordPress, Apache, PHP and nginx, and the open-source advisories for Drupal, Magento and the JavaScript libraries your pages load. With your repository connected we also check the packages in your lockfiles: up to 50,000 from each file, and up to 20,000 of those that a published advisory names. Anything past either limit is listed as not checked. A record whose affected versions we cannot read exactly is left out, never guessed. When a version can only be narrowed down, as on a Linux server that patches without changing its version number, we tell you so.

1,779

of all the vulnerabilities ever published, from every year, are on CISA's list of ones attackers are known to be using

Where the figure comes from

We count the CVE Program's own records every day, published records only, and this is the same figure our homepage shows.

What applies to your site

We match the software and versions your site reveals against those records. Where a match is one of the **1,779** on CISA's list of vulnerabilities known to be exploited, the finding says so and it moves to the top of your list.

How it stays current

The records our checks match against are refreshed every day, so a vulnerability published today in those sources is tested on your first scan after the next refresh. This scan ran **96** checks in all, each listed under What we checked.

Source: CVE Program cvelistV5, release cve_2026-11-12_1400Z,
read 2026-11-12.

Frontier Cyber Defence, scan sample-0000-0000-0000-000000000000 Powered by THRead

04 Findings

Findings

We proved 10 of these 14 on your site and exploited 4. Each one carries what our check did, what it reaches and the change that closes it, with the ones we exploited first.

reaches control

CRITICAL

exploited

default credentials

Default credentials accepted at /admin/login

An account is still on its factory password.

What we proved. We tried a short list of factory passwords on your login and one was accepted. We confirmed it by carrying the session it handed us to a page an anonymous visitor cannot open, then stopped.

What it is. An account is still on its factory password: admin/admin, or a default database, router or device login that was never changed.

The risk. Anyone who tries the factory password logs in as that account and can do everything it can.

How likely. Routine. Default passwords are public and tried first.

What it reaches. Control of whatever the account runs, up to Root on a server or device login.

Why the severity. Critical on an admin or infrastructure account. High elsewhere.

How we know. We performed the attack ourselves and recorded what it gave us access to.

reaches data

CRITICAL

exploited

path traversal

Path traversal via ?file= reads arbitrary files

A page that serves files lets a visitor step out of the intended folder.

What we proved. We asked for a path outside the intended folder and the response contained the system password file, /etc/passwd. We read that one file and stopped.

What it is. A page that serves files lets a visitor step out of the intended folder by asking for ../../ paths.

The risk. They read files off the server that were never meant to be public: source code, configuration, keys, the system password file.

How likely. Routine to test. Reliable when present.

What it reaches. Data, and the makings of Control. Config and source often carry the keys to everything else.

Why the severity. Critical. It reads files off the server, and the files it reads carry the keys to everything else.

How we know. We performed the attack ourselves and recorded what it gave us access to.



reaches data

HIGH

exploited

sensitive file exposure

Sensitive file exposed: /.env

Files that were meant to be private are readable straight from the web.

What we proved. We asked for the file, having first checked that your server does not answer every unknown path with the same page. We call it read only when what came back matches the format that path claims to be.

What it is. A database backup, a .env config, a .git folder, or an open directory that lists everything in it, all readable from the open web.

The risk. An attacker downloads them and reads what is inside. A config file or a backup often holds passwords, API keys and customer records in a single file.

How likely. Routine. Automated scanners check the common paths within seconds of finding your site.

What it reaches. Data. Where the file holds a live key, that key can hand over Control or the database with no further work.

Why the severity. High. Critical when the exposed file contains a secret or personal data.

How we know. We performed the attack ourselves and recorded what it gave us access to.



reaches foothold

HIGH

exploited

crlf injection

HTTP response splitting via ?next= parameter

Visitor input lands in a response header without being cleaned.

What we proved. We put an encoded line break in a parameter and a header of our own came back in your server's response.

What it is. Input from a visitor lands in a server response header without being cleaned, letting an attacker insert new lines and forge extra headers.

The risk. They can plant a poisoned response in a shared cache that is then served to other visitors, set their own cookies, or trigger a cross-site script.

How likely. Situational to find. Effective when present.

What it reaches. Foothold, and Control of a visitor's session through the chained script.

Why the severity. High. We watched our own header come back, so this one is proven, and a shared cache spreads whatever is injected next.

How we know. We performed the attack ourselves and recorded what it gave us access to.



reaches data

CRITICAL

confirmed

sql injection

Database error on injected quote in "id" at /product

Your site builds a database query out of something a visitor typed.

What we proved. We added a single quote to a parameter and your page came back with a database error. That is the signature of an injection; we did not push on it and we read no data.

What it is. Your site builds a database query out of something a visitor typed, so an attacker can rewrite the query and ask the database whatever they like.

The risk. Where the injection is real, an attacker reads and changes the database directly: customers, orders, password hashes.

How likely. Routine to test. Where it is present it is usually fully exploitable.

What it reaches. Data, meaning the database sitting behind that page.

Why the severity. Critical when it is real, because it is direct access to everything you store.

How we know. We proved it is real and reachable.



reaches data

HIGH

confirmed

cross-site scripting

Reflected, unescaped form input "q" at /search

Your page shows text from a visitor without cleaning it first.

What we proved. We put a harmless marker in a parameter and found it echoed back into the page unescaped. No script of ours ran, and this check only finds the reflected kind.

What it is. Your page shows text from a visitor without cleaning it, so an attacker can make the page run their code inside someone else's browser.

The risk. Their code runs as your visitor. It can take the session cookie and use the account, capture what they type into a form, or quietly change what the page does. Aimed at an admin, it takes over the admin.

How likely. Likely. One of the most common holes on the web and easy to test for. This kind needs the victim to open a link the attacker built.

What it reaches. Data belonging to the visitor who opens that link: their session, and whatever their account can see. It reaches Control where the visitor who opens it is an admin.

Why the severity. High where the reflection lands on a page your customers use. Critical on an admin console or a payment page.

How we know. We proved it is real and reachable.



reaches recon

LOW

confirmed

novel frontier checks

A check our attacker invented for your site flagged /account/export

Checks our own attacker model invented, on top of the standard set.

What we proved. Our attacker model invented this check out of what your site gave it, and ran it non-destructively. We report the count and the result, and we keep the method to ourselves.

What it is. A check our attacker model wrote for your site specifically, from the evidence this scan collected. It runs under the same rules as everything else: read-only, non-destructive, and only against the server you proved you own.

What it reaches. Whatever the check finds is reported as its own finding, in the category it belongs to. This entry exists so you can see how much of that work ran.

Why the severity. Low, and confirmed: our own check fired on your site. We keep the method to ourselves, so we report it at the lowest severity rather than claim a reach this report cannot show you.

How we know. We proved it is real and reachable.



reaches recon

LOW

confirmed

cookie flags

A cookie is set without Secure, SameSite

A cookie is set without its safety flags.

What we proved. We loaded your home page as an anonymous visitor and read the flags on the cookies it set. We never logged in, so we cannot tell you which of your cookies carries a session.

What it is. A cookie your site sets is missing its safety flags (Secure, HttpOnly, SameSite), so a script can read it or it travels over plain HTTP.

The risk. Where the cookie in question carries a login, an attacker who can run a script on your page or watch the network takes it and uses that account.

How likely. Situational. It needs a second hole, an injected script or an insecure link, to cash in.

What it reaches. Nothing on its own. It reaches Control of an account where the cookie missing the flag is the one carrying a login and something else can read it.

Why the severity. Low. High on the session cookie of an admin.

How we know. We proved it is real and reachable.

---

reaches recon

LOW

confirmed

security headers

Missing security headers: CSP, HSTS, Referrer-Policy

Your server does not send the browser its safety instructions.

What we proved. We read the response headers your server sends and listed the ones that are absent.

What it is. Content-Security-Policy, HSTS, X-Frame-Options and their neighbours are missing. These headers tell the browser what to trust.

The risk. An attacker has more room to run a script in your visitor's browser, to frame your site inside theirs and trick a click, or to drop a visitor onto plain HTTP and read the traffic.

How likely. Routine to spot. On its own it rarely lets anyone in. What it does is make every other hole easier and more reliable.

What it reaches. Nothing on its own. It widens an injected script or a session theft once another hole opens the door.

Why the severity. Low alone. We raise it when a real script-injection hole or a login sits behind the missing header.

How we know. We proved it is real and reachable.

---

reaches recon

LOW

confirmed

source map exposed

Source map exposed: /js/app.js.map

The source maps for your minified JavaScript are published alongside it.

What we proved. We downloaded the .map file next to your bundle and it parsed as a version 3 source map.

What it is. The source maps that translate your minified JavaScript back into readable code are published alongside it.

The risk. An attacker downloads your original front-end source, comments and all, and reads it for endpoints, keys and logic to attack.

How likely. Routine to pull when present.

What it reaches. Recon, occasionally Data when the recovered source carries a secret.

Why the severity. Low. Raised when the recovered code leaks a key.

How we know. We proved it is real and reachable.

---

would reach data

HIGH

potential

secret exposure

Possible AWS access key id exposed in client code (/js/app.js)

A key is sitting in the code your site sends to browsers, where anyone can read it.

What we saw. We matched a key pattern in the code your site serves to browsers. We never tried the key against the service it belongs to, so we cannot tell you whether it still works.

What it is. A key, such as an API key, a token or a password, is baked into the JavaScript your site sends to browsers.

The risk. Whatever the key opens is open to whoever reads the page. A payment key, a cloud key or a database token in the page is a direct line to money, infrastructure or data.

How likely. Routine. The file is public and scanners read it automatically.

What it would reach. Data, Control or Root, set entirely by what the key opens. Some keys belong in the page. A browser key for maps or analytics is designed to ship, and a restriction on which domain may use it keeps it safe.

Why the severity. High for a server-side credential. Low for a browser key that is meant to be public, where the thing to check is its domain restriction. Critical once a key is confirmed live.

How we know. We saw the signature of the hole. We did not push on it.

---

would reach foothold

MEDIUM

potential

cors misconfiguration

Permissive CORS (reflects arbitrary Origin)

Your server tells browsers that any website may read its responses.

What we saw. We asked with an Origin header naming a site we do not own, and your server told the browser that site may read the response. We read no private data through it.

What it is. Your server echoes back whatever Origin a page claims and tells the browser that page is allowed to read the response, sometimes with the visitor's login attached.

The risk. A page the attacker controls can call your site as your logged-in customer. Where an endpoint behind that header returns private data, their page reads it back: account details, order history, saved addresses.

How likely. Likely where the header is wrong. It needs the victim to visit the attacker's page first.

What it would reach. Foothold: the attacker's page gets to talk to your site as your visitor. It reaches Data where an endpoint behind that header returns personal records, which we have not tested.

Why the severity. Medium. High when the response is allowed to carry the visitor's credentials.

How we know. We saw the signature of the hole. We did not push on it.

---

would reach
recon

MEDIUM

potential

cross-site request forgery

POST form without an anti-CSRF token at /account/update

A form accepts an action without proving the request came from your own page.

What we saw. We read the HTML of a POST form and saw no token field in it. This one is a heuristic. An app can carry the token in a header or a cookie our crawl cannot see, so check the form before you act on it.

What it is. A form or action runs without a token proving the request came from your own site, so another page can submit it on a logged-in visitor's behalf.

The risk. An attacker's page can make your visitor change their email, their password or their saved details without meaning to.

How likely. Situational. It needs the victim to be logged in and to open the attacker's page.

What it would reach. Nothing on its own, on this evidence. It reaches Control of the account where the unprotected action is sensitive and the token really is absent.

Why the severity. Medium, set by what the unprotected action does.

How we know. We saw the signature of the hole. We did not push on it.

---

would reach
recon

INFO

potential

api surface

API endpoints discovered: /api, /openapi.json

Endpoints answer requests directly, outside the pages of your site.

What we saw. We requested the path and it answered with JSON. We did not test whether it checks who is asking, and several of these are meant to be public.

What it is. API endpoints answer requests directly. This one may be documented, an old version, a health route or a debug route.

The risk. Each one is somewhere an attacker can aim without going through your pages. Where an endpoint answers without checking who is asking, they read or change data the website itself would never let them touch.

How likely. Likely to be found. Automated tools sweep the common paths in seconds.

What it would reach. Recon. We know these answer. Whether they check who is asking is the next test, and that is where the reach is decided.

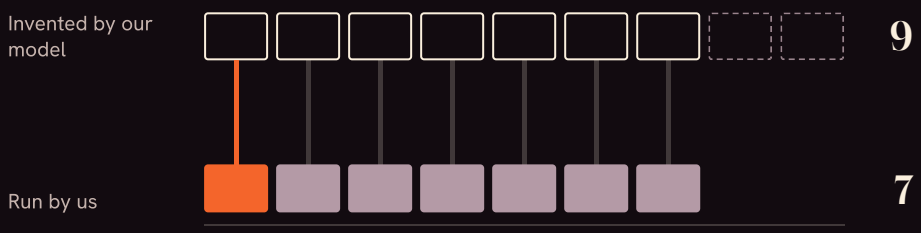
Why the severity. Info. A schema file or a health check is meant to be there. The severity comes from what an endpoint does once someone tests it.

How we know. We saw the signature of the hole. We did not push on it.

05 Attacks our model invented

We ran 7 attacks our model invented for your site. 1 was confirmed.

Our attacker model read what this scan collected about your site and invented 9 attacks written for it. We turned each one into a check and ran 7 of them against your site ourselves, as real read-only requests. 1 was confirmed and 6 came back clear. The other 2 could not be run on this scan, so they are not shown.



Each square is one attack our model invented for your site. A line down means we ran it.

- 1 confirmed and in your findings
- 6 came back clear
- 2 could not be run on this scan

The 1 confirmed attack is in your findings, in section 04 under novel frontier checks. It is there because our own check proved it.

How the attacks were invented

Our attacker model read what our passive pass recorded about your site, including the pages it fetched, the software it recognised, the admin paths that answered, the headers that were missing and the forms it found. From that it wrote attacks for this site alone.

How we ran them

Each attack became a check in this scan. It passed our safety validator first and then ran against northwind-shopfront.example as a real request. It reads and never writes, and it only reaches the server you proved you own.

What this report shows

A confirmed attack is a finding, with what our check found. One that came back clear is counted here. One we could not run is counted here and nothing more, because only our own check can settle it.

Why we invent attacks

The attacks that reach a site are more and more often written for that site. An attacker with a frontier model reads your pages, your software and your forms the way ours does, and invents a way in that no published list holds yet. Our model does that first, on every scan, and we run what it writes against your site ourselves. A pen test is one team on one day, and it cannot do this between visits.

Every scan makes the next one stronger

Every attack our model invents is kept as a check and runs again on every later scan of your site. Once one has fired, been confirmed and raised no false alarm across real scans, it joins the set every site we protect is tested with. What we keep is the method, a bounded read-only request, and your data stays yours. A check that raises false alarms or stops finding anything is retired, so the set stays sharp as it grows.

In all, **13** checks our model has written ran against your site on this scan, these 7 among them, and **12** passed. Their method stays inside FCD under our published guardrails, so this report gives their numbers.

06 Fixing from your repository

What connecting your repository does for these findings

3 of the 14 findings in this scan can be changed from your code. Connect your repository and we write those changes, check them and open them as pull requests for you to review. Give us access to the code your developers work on and we make the developer changes too, so the whole list closes without waiting on anyone's backlog.

1. You connect your repository. Your token is used for the run and dropped when it ends.
2. We prove the repository is the code serving northwind-shopfront.example.
3. We build each change from a fixed template and quote it first.
4. Our gate checks every line, 5 files and 60 added lines at most.
5. A pull request opens for you to review and merge.
6. Your next full scan proves the hole closed.

Where each finding in this scan is fixed



- Fixed in your code by pull request 2
- Fixed at your server, host or DNS 2
- Developer changes 5
- Helped from your code 1
- Needs your decision first 4

Fixed in your code by pull request 2

We write the change from a fixed template, check it, and open a pull request for you to review. The templates cover Node.js sites built on Express, security headers on sites Netlify or Apache serves, and PHP error pages and session cookies on Apache. The fix list carries the steps for every other stack.

`cookie flags 1, security headers 1`

Helped from your code 1

We add the file to .gitignore so it stops spreading, and the fix list gives the steps to take it off your server and rotate anything in it.

`sensitive file exposure 1`

Fixed at your server, host or DNS 2

Set where your site is served. We give you the exact lines for your platform, and your next scan proves them in place.

`api surface 1, source map exposed 1`

Needs your decision first 4

The right answer is your call, such as which sites may read your API or which password to set. The fix list gives the steps once you have decided, and your next scan proves the change is in place.

`cors misconfiguration 1, cross-site request forgery 1, default credentials 1, secret exposure 1`

Developer changes 5

These change what your application does with what somebody sends it. Give us developer access to your repository and our engineers make these changes, as pull requests you review.

`crlf injection 1, cross-site scripting 1, novel frontier checks 1, path traversal 1, sql injection 1`

Code hosts we connect to

GitHub
GitLab and Bitbucket follow once each has been proved on a real repository.

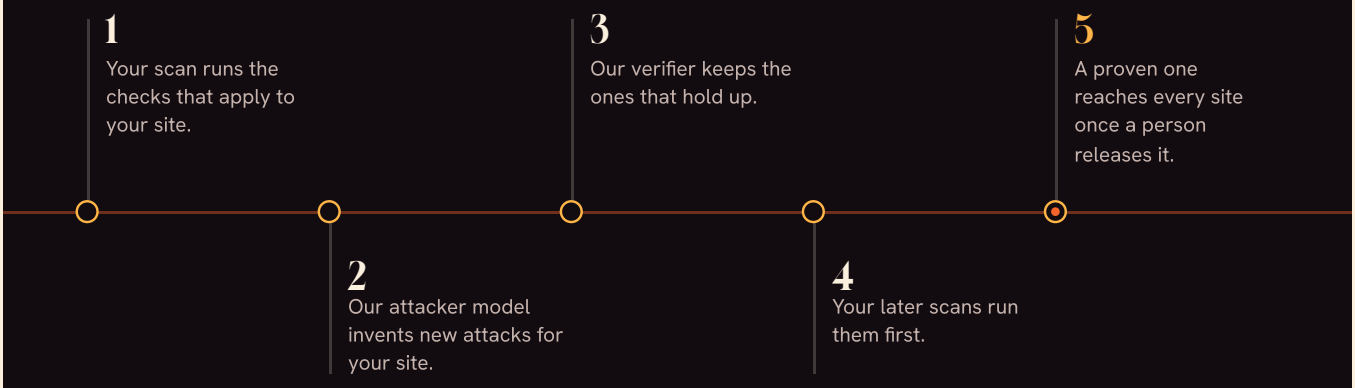
Sites with no repository

The fix list on your dashboard gives the exact steps for WordPress, Drupal, Joomla, Magento, Shopify, Wix, Squarespace and Webflow, and for sites behind Cloudflare, nginx, Apache or Microsoft IIS. Your next scan proves each one closed.

07 Recursive self-improvement

Invented attacks we keep and run again

Recursive self-improvement means our protection gets better each time it runs. Our attacker model invents attacks for your site as it goes. The ones our verifier confirms are kept with your account, and your later scans run them again first, up to 400 a scan. An invented attack reaches other sites only once it has proved itself on real scans and a person on our team has reviewed it.



How FCD applies it

An invented check reaches other sites only after it has fired and been confirmed on its own site with no false alarms, over at least 3 runs across 2 separate scans, and a person releases it, 5 a day at most. A check that raises 2 false alarms, or runs 6 times without finding anything, is retired. What a scan finds also changes the next scan of that site. Our attacker model looks first at the areas where it turned something up. Every release of FCD is attacked by our own red team. When the attack runs before a release goes live, a critical hole it confirms stops the release, and so does a high one unless the release changes nothing that ships. Some releases go live first and are attacked afterwards on the same build, where a hole like that fails the run and tells us to fix it, so the release you are using may not have been attacked yet.

What it means for you

Your next scan starts with what the last one learned about your site, and with the checks FCD has proved on other sites since, without you doing anything.

Why it matters

Attackers use the same frontier models we do, so each improvement in those models reaches your scans as well.

08 Method and terms

Method and terms

You proved you control `northwind-shopfront.example` before we started, and we scanned only that server. The whole assessment is non-destructive: we read, we probe, and we stop the moment a hole is proven, leaving your data and your site exactly as we found them.

Every finding is either a potential hole, a confirmed one, or one we exploited. Each word means something different.

potential

We saw a sign of the hole: its signature, a version known to carry it, or a lead our attacker model raised. We did not push on it.

confirmed

A check proved it is real and reachable, or our frontier verifier reviewed the evidence and judged it real. Neither touches your data.

exploited

We performed the attack ourselves and recorded what it gave us access to.

Each finding card and the summary say how far an attack gets, in five steps.

recon

It tells an attacker about your site, or makes a later attack easier, and opens nothing by itself.

foothold

It lets an attacker act in your name or on your visitors' connections, short of your data: sending mail that passes as yours, sending your visitors somewhere else, or reading a response meant for them.

data

It reaches data: what your site keeps, or what a visitor sees and types on your pages.

control

It lets an attacker change what your site does, such as signing in as its administrator.

root

It runs commands on the server itself.

How we count missing headers changed on 16 November 2026. A missing Content-Security-Policy and a missing Strict-Transport-Security are now each their own finding, with their own fix. A report from before that date lists them

inside "Missing security headers", so the same site can show up to two more findings than it did, with nothing new on the site.

Behind the scan, an autonomous attacker probes your site and a separate frontier model reviews every claim against the raw evidence before it reaches this page. That review can confirm a finding; only our own check can mark one exploited, and it does that solely when it performed the attack and captured what came back. Where you see exploited, it happened.

09 Appendix

Appendix

This run's parameters, and every kind of hole FCD looks for on every scan.

Scan parameters

target northwind-shopfront.example

scan id sample-0000-0000-0000-000000000000

started 12 November 2026

ownership verified before the scan started

mode non-destructive, single origin

What we check for

`security headers`
The browser safety instructions your server should send.

`sensitive file exposure`
Private files and open directories readable from the web.

`admin surface`
An admin login or control panel reachable from the open internet.

`cross-site scripting`
Pages that run a visitor's code in someone else's browser.

`sql injection`
Visitor input that rewrites your database queries.

`template injection`
Input that runs as code inside your page templates.

`cors misconfiguration`
A server that lets any website read its responses.

`open redirect`
A link on your domain that forwards visitors anywhere.

`path traversal`

Requests that step out of a folder to read other files.

`info disclosure`

Error pages that leak how your site is built.

`cookie flags`

Cookies set without their safety flags.

`api surface`

Endpoints that answer requests directly, outside your pages.

`graphql introspection`

A GraphQL API that hands out its full schema.

`prompt injection`

An AI feature that obeys instructions hidden in its input.

`injected instructions in your content`

Text on one of your pages that gives instructions to automated readers.

`http methods`

Unsafe methods like TRACE or open write verbs.

`crlf injection`

Input that forges extra response headers.

`host header injection`

A trusted Host header an attacker can set.

`source map exposed`

Published maps that rebuild your front-end source.

`secret exposure`

Keys and passwords left in the code your site sends to browsers.

`default credentials`

Accounts still on their factory password.

`cross-site request forgery`

A form that runs without proving the request came from your own site.

`email authentication`

Whether your domain can be impersonated in email.

`TLS and certificates`

What your server accepts when a browser connects.

`published versions`

The software versions your own responses give away.

`support status`

Whether the software you run still gets security updates.

`third-party code`

Other people's code running on your pages.

`browser surface`

What your pages let the browser do.

`login form`

How your login form handles the password.

`tokens and sessions`

How your site carries and hands out tokens.

`access control`

Whether a rule that refuses a stranger really refuses them.

`cloud exposure`

Storage and spare servers of yours that answer to strangers.

`injection surface`

Parameters that go straight into a query engine.

`AI agent`

What your AI agent is allowed to do, in your own words.